VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

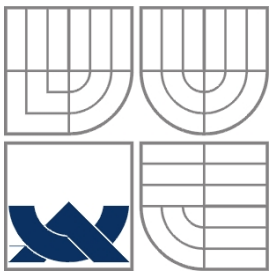
PŘÍPRAVA DOMÁCÍCH ÚLOH PRO PŘEDMĚT ALGORITMY

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

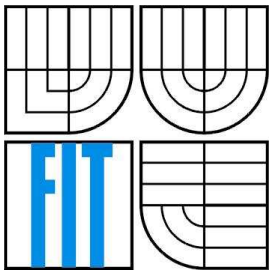
AUTOR PRÁCE
AUTHOR

TOMÁŠ ADÁMEK

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘÍPRAVA DOMÁCÍCH ÚLOH PRO PŘEDMĚT ALGORITMY

PREPARATION OF HOMEWORKS IN THE COURSE ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ ADÁMEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ROMAN LUKÁŠ, Ph.D.

BRNO 2008

Zadání bakalářské práce/6554/2007/xadame27

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Adámek Tomáš**
Obor: Informační technologie
Téma: **Příprava domácích úloh pro předmět Algoritmy**
Kategorie: Alg. a datové struktury

Pokyny:

1. Seznamte se se systémem pro automatizované zadávání a hodnocení domácích úloh v předmětu Algoritmy.
2. Po dohodě s vedoucím práce vyberte pokročilé tři příklady z původních příkladů v jazyce Pascal nebo navrhněte příklady nové.
3. Vybrané příklady vzorově implementujte v jazyce C. Součástí příkladů budou i testy pro ověření správnosti implementace.
4. V případě potřeby rozšiřte stávající systém pro zadávání a hodnocení domácích úloh podle pokynů vedoucího.
5. Zhodnoťte dosažené výsledky a navrhněte možné směry dalšího vývoje.

Literatura:

- Honzík, J. M., Hruška, T., Máčel, M.: Vybrané kapitoly z programovacích technik, Vysoké učení technické v Brně, Brno, 1991. ISBN 80-214-0345-4.
- Herout, P.: Učebnice jazyka C, 3. upravené vydání, Kopp, České Budějovice, 1998. ISBN 80-85828-21-9.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Lukáš Roman, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Tomáš Adámek**
Id studenta: 79124
Bytem: Revoluční 1816/37, 591 01 Žďár nad Sázavou
Narozen: 13. 06. 1986, Nové Město na Moravě
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Příprava domácích úloh pro předmět Algoritmy
Vedoucí/školitel VŠKP: Lukáš Roman, Ing., Ph.D.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

Tématem této bakalářské práce je problematika přípravy komplexní sbírky domácích úloh pro předmět Algoritmy. Úvodem práce je rozebrána problematika počítačových algoritmů. Znalost počítačových algoritmů patří k základním znalostem počítačového programátora. Stručně je zde popsána úloha algoritmů, datových struktur a jejich vlastností. Hlavní část práce se zabývá analýzou a vytvářením úloh na vyhledávání podřetězců v řetězcích. Nalezení vzorku v textu je oblastí, využívanou v široké škále počítačových programů, od prohlížečů až po textové editory. Implementací těchto úloh získá student rozšiřující znalosti o této problematice a dokáže je použít nejen v jazyce C. Součástí rozboru implementace úloh je také vytváření testovacích úloh pro ověření správnosti implementace a jejich charakteristiky. Další část této práce je věnována systému pro automatické hodnocení a zadávání domácích úloh. Je zde rozebrána implementace jednotlivých částí systému a možnosti použití.

Klíčová slova

Vyhledávání vzorku v textu, Knuth-Morris-Pratt algoritmus, Boyer-Moore algoritmus, elementární algoritmus, algoritmus, vyhledávání, jazyk C, předmět Algoritmy, domácí úloha, systém automatického opravování, testovací systém.

Abstract

The main theme of this bachelor project is creating complex homework collection for subject Algorithms. First part of this project deals with problems of computer algorithm. The knowledge of the computer algorithm belongs to basic knowledge of the computer programmer. There is brief description of the algorithm and data structure. The main part of this project deals of analyzing and creating new exercises for searching patterns in the text. Pattern matching in the text is used in the wide range of computer program. The implementation of this homework gets student a new knowledge about this theme. The next part of this implementation is also creating testing homework's for verification test. Last part of this project describes design and implementation automatic system for preparing and checking of homeworks in the course Algorithms.

Keywords

Pattern matching in the text, Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm, basic algorithm, algorithm, searching, C language, course Algorithms, homework, system for automatic correction, test system.

Citace

Adámek Tomáš: Příprava domácích úloh pro předmět Algoritmy. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Příprava domácích úloh pro předmět Algoritmy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Romana Lukáše, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Tomáš Adámek
1.května 2008

Poděkování

Především bych rád poděkoval mému vedoucímu Ing. Romanu Lukášovi, Ph.D. za poskytnutou pomoc, odborné vedení a čas věnovaný při konzultacích k tématu mé bakalářské práce.

Obsah

Úvod

Již z názvu mé úlohy je patrné, čím se bude téma této bakalářské práce zabývat. Jedná se především o přípravu domácích úloh pro předmět Algoritmy. Tento předmět je vyučován ve druhém ročníku bakalářského studia na Fakultě informačních technologií Vysokého učení technického v Brně. Jeho hlavním úkolem je naučit studenty základní principy počítačových algoritmů. Znalost počítačových algoritmů je nezbytnou součástí vybavení každého studenta, jež se chce pohybovat v oblasti počítačových technologií. Úkolem mé bakalářské práce je doplnit sbírku algoritmů tak, aby byl studentům i profesorům poskytnut co největší výběr při jejich implementaci či jejich zadávání. Implementační jazyk je již v zadání omezen na programovací jazyk C. Jedná se o univerzální programovací jazyk, který se po jazyce Pascal stal základem pro výuku počítačových algoritmů.

V úvodní části se zabývám popisem algoritmu a jeho vlastností. Pojem algoritmu je úzce spojen s datovými strukturami, proto je vhodné na úvod provést jejich stručný rozbor. Datová struktura může vznikat po provedení algoritmu nebo s algoritmem úzce spolupracovat. Jak je známo, pojem algoritmus a datová struktura dávají dohromady počítačový program. Před zanořením do podrobných problémů algoritmů je dobré vědět, jaké výhody a nevýhody jejich použití přináší a jaké jsou jejich základní vlastnosti.

V další části je nastíněna problematika vyhledávání podřetězců v řetězcích. Přestože se s tímto tématem studenti prozatím seznamují pouze okrajově, je toto téma velice zajímavé a vhodné pro rozšíření sbírky úloh pro tento předmět. V dnešní době se vyhledávání používá v celém rozsahu počítačových programů od prohlížečů až po operační systém. Hlavní částí praktické úlohy je vzorová implementace algoritmů Knuth-Morris-Pratt, Boyer-Moore pro vyhledávání v jazyce C. Tyto úlohy musejí být vzorově okomentovány a musí být vytvořeno testovací prostředí pro ověření správnosti implementace. Jedná se o testovací prostředí poskytnuté studentům a profesorům. Důležité je, aby při implementaci bylo dodrženo to, co je uvedeno v zadání zdrojového kódu a došlo tak k dodržení předepsaného algoritmu. Několik podobných algoritmů může vést ke shodnému výsledku, proto je třeba toto striktně kontrolovat a donutit tak studenta tento algoritmus pochopit a implementovat. Při vytváření testů je tedy potřeba analyzovat způsob vyhledávání a podle toho zařídit testovací rozhraní tak, aby bylo splněno zadání příslušného algoritmu.

Bakalářská práce není však zaměřena pouze na vyhledávání podřetězců v řetězcích a jejich implementaci, ale také na již vytvořený systém pro automatické hodnocení a zadávání domácích úloh. Tento systém má za úkol odlehčit pří přípravě domácích úloh a jejich hodnocení. Systém je v neustálém vývoji a je zde prostor pro jeho modifikaci a jeho zdokonalení.

V závěru práce jsou shrnuty možnosti dalšího směru vývoje systému pro automatické zadávání a hodnocení domácích úkolů pro předmět Algoritmy. Práce je zakončena zhodnocením výsledků mé práce.

1 Algoritmy a datové struktury

Při psaní počítačových programů potřebujeme znát metodu a postup, jakým budeme danou úlohu implementovat. Tato metoda je velice často nezávislá na určitém počítači a je tedy třeba zjistit, jak máme danou úlohu vyřešit obecně a jak bude organizována v rámci datových struktur. Algoritmy a datové struktury je tedy třeba prostudovat podrobněji, protože při vytváření počítačových programů jsou na sobě závislé. Podrobnější informace o vlastnostech algoritmů lze nalézt v [2, 3, 4].

1.1 Algoritmy

Algoritmus[4] je konečná uspořádaná množina úplně definovaných pravidel pro vyřešení nějakého problému. Intuitivně algoritmem rozumíme postup, který nás dovede k řešení úlohy. Formálněji vyjádřeno se jedná o přesně definovanou konečnou posloupnost kroků (příkazů), jejichž prováděním pro každé přípustné vstupní hodnoty získáme po konečném počtu kroků odpovídající hodnoty výstupní.

Hlavní snahou při vzniku algoritmů je zrychlení řešení úloh a také úspora ohledně prostoru, na kterém operují. Pro řešení jednoho typu úlohy může být k dispozici několik druhů algoritmů, které nás dovedou ke shodnému výsledku. Tyto algoritmy se však liší ve způsobu provedení a ve stylu jejich implementace. V mnoha případech volba správného algoritmu dokáže ovlivnit chod celého systému. Dalším důvodem vzniku algoritmů je potřeba často vytvářet či používat postupy, které již někdo před námi vymyslel a není tedy třeba znovu vytvářet již vytvořené. Dochází tak k úspoře času programátora, který může tyto algoritmy analyzovat, vylepšit a zlepšit tak jejich chod.

1.1.1 Základní vlastnosti algoritmů

Abychom při tvorbě algoritmů věděli, jak je vytvářet a měli tak alespoň základní pravidla, je třeba definovat si několik základních vlastností. Dostane-li algoritmus na vstup korektní data, musí nám poskytnout požadovaný výsledek.

Každý algoritmus musí tedy splňovat tato pravidla:

- **Determinovanost:** Znamená vlastnost algoritmu, při které je přesně stanoveno, jak má program v daném místě pokračovat. Jsou určena přesná pravidla jednotlivých kroků programu. Při splnění těchto pravidel musí program po každém spuštění poskytnout stejný výsledek.
- **Obecnost:** Tato vlastnost udává, že jedním algoritmem lze řešit více úloh, které jsou stejného typu. Program musí dokázat vyřešit úlohu pro libovolné vstupní hodnoty, pokud splňují vstupní podmínky programu.

- **Konečnost:** Vlastnost algoritmu, která nám říká, že výsledek algoritmu musí být znám po konečném počtu kroků. Počet kroků se může pro jednotlivé algoritmy lišit, ale pro každý vstup musí být konečný. Program se tedy při splnění této podmínky nemůže dostat do stádia zacyklení.
- **Efektivnost:** Vlastnost závislá na použitém algoritmu. Nemá vliv na správný výpočet programu. Zajišťuje pouze to, aby program byl co nejrychlejší a zbytečně nezatěžoval systém při výpočtu.
- **Rezultativnost:** Algoritmus při zadání korektních vstupních dat musí vždy vrátit nějaký výsledek. Může to být například pouze jednoduchý výpis na výstup.

1.1.2 Algoritmická složitost

Další podstatnou vlastností algoritmů je jejich složitost. Pro každý algoritmus nás zajímá jeho náročnost na čas, paměť a prostor. Proto lze složitost algoritmů rozdělit do kategorií časové, paměťové (prostorové) složitosti. Tyto vlastnosti závisejí především na velikosti a rozsahu vstupních dat. Tato vlastnost nám umožní porovnat jednotlivé algoritmy bez ohledu na systém, na kterém pracují. Časovou či paměťovou složitostí se tedy snažíme vyjádřit efektivitu jednotlivých algoritmů.

Obvykle je třeba rozlišovat složitost algoritmu při provádění nejlepšího, nejhoršího a průměrného případu. Mezi typicky používané případy složitosti se řadí: O(1) – konstantní, O(log N) – logaritmická, O(N) – lineární, $O(N^2)$ – kvadratická, $O(N^3)$ – kubická, $O(2^N)$ – exponenciální, O(N!) – faktoriálová. Tyto složitosti jsou seřazeny od té časově nejrychlejší po tu nejpomalejší.

Časová složitost

Časová složitost nám udává počet kroků provedených od začátku po konec provádění algoritmu. U této složitosti sledujeme vstupy, výstupy, aritmetické operace, logické operace, operace porovnání, výměnu prvků a dobu trvání jednotlivých operací. Příklady časových složitostí: O(1) = přístup k prvkům statického pole, $O(\log_2 N)$ = vyhledávání v již seřazeném poli metodou půlení intervalů, O(N) = vyhledávání prvků lineární metodou v seřazeném poli.

Z předchozích příkladů je vidět, že při zpracování různých typů dat a při použití různých metod se složitosti liší. Časovou složitost můžeme určit, jako počet jednoduchých operací, které můžou být nad daty provedeny. Tuto složitost pak můžeme odvodit již na základě rozsahu jednotlivých dat, bez konkrétní znalosti přesných hodnot dat. Určením této složitosti získáme velikost spotřebovaného času nad použitými daty. Při zvětšování rozsahu vstupních dat je vhodné použít funkci s pomaleji rostoucí funkcí pro snížení časové složitosti. U algoritmů s lineární časovou složitostí dochází například při deseti násobném zvýšení rychlosti výpočtu k deseti násobnému zvýšení rozsahu zpracovaných dat. Při kvadratické časové složitosti to je přibližně pouze trojnásobné zvětšení rozsahu použitých dat. Pro značně rozsáhlá data musíme tedy hledat přijatelnější časové složitosti, vyhledáním lépe provedeného algoritmu.

Paměťová složitost

Udává nám velikost paměti potřebné pro daný výpočet. Mezi paměťovou a časovou složitostí je možno nalézt závislost. Nelze však na základě znalosti jedné složitosti odvodit tu druhou. Obecně však lze za použití většího rozsahu paměťového prostoru vylepšit časovou složitost a naopak.

Horní odhad složitosti

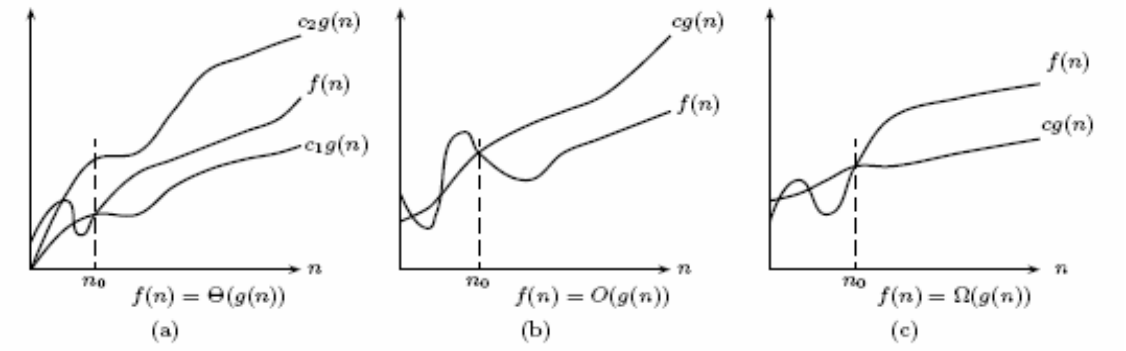
Jedná se o vlastnost udávající složitost algoritmu v nejhorším případě. Obecně nás tato vlastnost zajímá nejvíce. Je označována písmenem O jako omikron. Typicky jde o případ s nejhorším vstupem. Tuto složitost pak lze označit funkcí f(n) = O(g(n)). Tato funkce udává, že funkce f(n) roste maximálně tak rychle jako g(n). Funkci g(n) označíme jako horní mez množiny funkcí. Grafické znázornění je na obrázku 1.1 varianta b). Funkce f(n) musí nabývat hodnot menších nebo rovných hodnotě cg(n) a zároveň větších než 0, aby patřila do množiny O(g(n)).

Dolní odhad složitosti

Je označována písmenkem Ω (omega). Jedná se o složitost algoritmu pro nejlepší vstup, například při řazení již seřazené posloupnosti hodnot. Pokud tuto hodnotu pro algoritmus známe, pak je zřejmé, že nikdy nedosáhne lepšího času provedení než je tato hodnota. Funkci g(n) lze označit jako dolní mez množiny funkcí při zápisu f(n) = Ω(g(n)). Grafické znázornění je na obrázku 1.1 varianta c). Funkce f(n) musí nabývat hodnot větších nebo rovných hodnotě cg(n), aby patřila do množiny Ω(g(n)).

Průměrná složitost

Jedná se také o střední hodnotu náhodné složitosti pouze při určitém vstupu dat. Průměrnou složitost označujeme jako Θ (theta). Jedná se o složitost pro standardní vstup. Počítá se jako pravděpodobnost výskytu jednotlivých vstupů dat a složitostí. Grafické znázornění je na obrázku 1.1 varianta a). Funkce f(n) musí nabývat hodnot v intervalu c_1g(n) a c_2g(n), aby patřila do množiny Θ(g(n)).



Obrázek 1.1 – Grafická ukázka **a)** Průměrná složitost Θ, **b)** Horní odhad složitosti O, **c)** Dolní odhad složitosti Ω.

1.2 Datové struktury

Při tvorbě algoritmů je důležité vhodně vybrat datovou strukturu, se kterou budeme pracovat. Správný výběr datové struktury při tvorbě dokáže výrazně ovlivnit celý postup implementace. Na základě tohoto požadavku je třeba uvážlivě volit datovou strukturu, kterou si pro naše řešení vybereme. Zde uvedu pouze nejzákladnější rozdělení datových struktur, protože se jedná o značně rozsáhlé téma. Podrobnější informace o datových strukturách lze nalézt v literatuře [2, 3, 4].

1.2.1 Základní dělení datových struktur

Datové struktury se rozdělují na:

- **Homogenní:** Proměnné v této struktuře musejí být stejného typu. Jedná se například o pole.
- **Heterogenní:** Proměnné této datové struktury nemusejí být stejného typu. Tímto datovým typem je například struktura složená z několika datových typů (např. int, double, char).
- **Statické:** Počet proměnných je znám již v době překladu. Nelze je následně měnit.
- **Dynamické:** Je opakem statických datových struktur. Počet proměnných i způsob uspořádání se může měnit v průběhu výpočtu. Za běhu programu se tyto struktury dokáží automaticky zvětšovat dle potřeby a nastavení programu.

1.3 Postup řešení problémů

Při tvorbě programů potřebujeme znát postup, který nás dovede k požadovanému řešení. Zajímá nás tedy množina vstupních, výstupních hodnot a podmínky nutné k vyřešení úlohy. Musíme specifikovat a analyzovat daný problém tak, aby naše výsledné řešení bylo co nejlepší a nejefektivnější.
Postup řešení úloh při tvorbě programů lze rozdělit do několika kroků:

- **Specifikace problému:** Musíme si určit požadavky, které budou kladeny na úlohu. Určíme vstupy a výstupy, na které bude program reagovat. Na kvalitní specifikaci problému může záviset celý následující postup při tvorbě.
- **Analýza:** Zvolíme si metodu vhodnou pro řešení daného problému. Složitější problémy máme možnost rozložit na jednodušší.
- **Návrh algoritmu:** V této části vytvoříme posloupnost na sebe navazujících kroků.
- **Implementace:** Vytvořený algoritmus zapíšeme v jazyce, kterému rozumí překladač a který jsme si pro implementaci vybrali.
- **Testování:** Snažíme se zjistit, zda implementovaný algoritmus odpovídá předem stanoveným podmínkám. Ve fázi testování zjišťujeme výkonnost, bezpečnost, správnost a jiné předem stanovené vlastnosti.
- **Udržování:** Závěrečná fáze nasazení vytvořené úlohy do provozu a její údržba.

2 Vyhledávání podřetězců

Jak již jsem v úvodu nastínil, vyhledávání podřetězců v textu je jednou ze základních součástí počítačových programů a patří mezi typické programátorské úlohy. Dá se říci, že se v dnešní době bez nich neobejde jediný textový editor, webový prohlížeč, adresářová služba a dokonce také operační systém. Velikost prohledávaných dat dokáže výrazně ovlivnit chod celého systému. Nastává tedy potřeba rychle vyhledat zadaný vzorek textu, což vyžaduje neustálé vylepšování těchto úloh a snahu o jejich zrychlování. Z tohoto důvodu nás u těchto úloh obvykle nejvíce zajímá časová složitost algoritmu.

2.1 Vyhledávací algoritmy

Vyhledávací algoritmy lze rozdělit do několika skupin podle principu na jakém pracují. Všechny tyto skupiny mají vztah ke hledanému vzorku a textu, ve kterém hledáme.

Dělení vyhledávacích algoritmů

Rozdíl mezi jednotlivými vyhledávacími algoritmy je v tom, zda dochází k předzpracování zadaného vzorku nebo textu. Vyhledávací algoritmy pak na základě předzpracování vzorku či textu dokáží doplnit datové struktury tak, aby se tyto struktury daly použít pro urychlení vyhledávání.
Tyto vyhledávací algoritmy lze rozdělit do několika skupin:

- **Skupina I** - nepoužívá předzpracování vzorku ani textu, jedná se například o elementární vyhledávací algoritmus.
- **Skupina II** - předzpracuje vzorky a vytvoří pro ně datovou strukturu, která se využije pro vyhledávání v textu. Jedná se o algoritmy jako jsou Knuth-Morris-Pratt a Boyer-Moore.
- **Skupina III** - vytvoří uspořádaný seznam slov textu a poskytne informace o jejich pozici. Dochází zde k předzpracování textu nikoliv vzorku.
- **Skupina IV** - pro text i vzorek vytvoří seznam, který je následně porovnáván. Algoritmy této skupiny používají předzpracování vzorku i textu.

Postup vyhledávání podřetězců

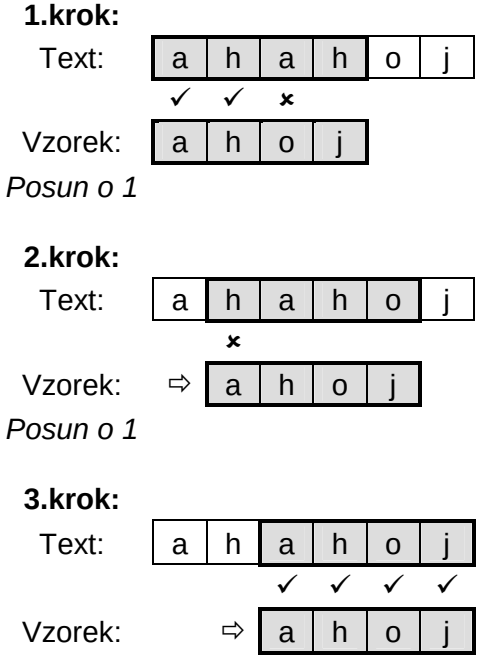
Při vyhledávání nás obvykle zajímá, jestli se daný vzorek v textu nachází, na jakém místě se nachází a kolikrát se v textu opakuje. Při vyhledávání potřebujeme znát hledaný vzorek, jeho délku, dále text, který budeme prohledávat a nakonec délku prohledávaného textu. K porovnání dochází dokud nenarazíme na konec prohledávaného textu, postupným porovnáváním vzorku s textem. Přesný postup vyhledávání bude uveden v souvislosti s použitým algoritmem. Důležitý při porovnávání je také směr porovnávání textu se vzorkem. Jsou zde dvě možnosti a to procházení vzorku zleva

doprava ve směru toku textu a opačný, zprava doleva. Úkolem těchto algoritmů musí být při prohledávání také nalezení z části se překrývajících vzorků v textu. Tyto algoritmy musí efektivně vyhledávat v rozsáhlých textech, proto je k dispozici mnoho různě modifikovaných algoritmů.

2.2 Elementární vyhledávací algoritmus

Mezi klasické vyhledávací algoritmy patří elementární algoritmus vyhledávání „hrubou silou“. Tato úloha prochází postupně prostor všech možných řešení, aby nalezla očekávané řešení. Jeho obrovskou nevýhodou je paměťová i kvadratická časová složitost při vyhledávání. Úkolem tohoto algoritmu je najít opravdu ideální řešení, ale za cenu obrovské režie algoritmu. V žádném případě zde nedochází k předzpracování vzorku. Čas potřebný k nalezení zadaného vzorku v textu narůstá v některých případech exponenciálně. Časová složitost tohoto jednoduchého algoritmu pro nejhorší případ je O(m*n), kde m je délka vzorku a n délka textu. Pro vyhledávání v obyčejném textu má tento algoritmus složitost O(m+n).

Text je zpracováván postupně zleva doprava. V případě, že dojde k neshodě v porovnání s prohledávaným textem, musí se vzorek posunout o jednu pozici vpravo a od prvního znaku vzorku pokračovat v prohledávání. Text je znak po znaku porovnáván s hledaným vzorkem. Pokud se index ve vzorku dostane na hodnotu délky vzorku, pak je vyhledávaný vzorek nalezen a pokračuje se v dalším prohledávání. Vyhledávání je znázorněno na obrázku 2.1. Nevýhodou tohoto algoritmu je nutnost vracet se v již prohledaném textu zpět.



Obrázek 2.1 *– Ukázka postupu vyhledávání elementárním algoritmem.*

2.3 Specializované algoritmy

Nutnost urychlit vyhledávání si vynutila vytváření dokonalejších algoritmů. Většinou se jedná o algoritmy, které si dokáží hledaný vzorek předzpracovat. Prohledají hledaný vzorek a na základě hodnot tohoto vzorku způsobí vylepšení samotného algoritmu vyhledávání. Mezi tyto algoritmy se řadí algoritmus Knuth-Morris-Pratt a algoritmus Boyer-Moore, probírané pouze okrajově v předmětu Algoritmy. Oba tyto algoritmy umožňují předzpracování vzorku a to za pomoci rozdílných postupů při jejich vytváření. Stejně tak využívají rozdílné postupy při samotném prohledávání textu. Snaha těchto algoritmů je vylepšit kvadratickou časovou složitost u vyhledávání elementárním algoritmem. Více informací o těchto algoritmech je možné nalézt v literatuře [1, 5].

2.3.1 Knuth-Morris-Pratt (KMP) algoritmus

Původní implementace vycházela z elementárního algoritmu vyhledávání popsaného v kapitole 2.2. Knuth-Morris-Pratt algoritmus vznikl jako rozšíření Morris-Prattova algoritmu. Základní vlastností Knuth-Morris-Prattova algoritmu je možnost předzpracování vzorku. Vylepšením Morris-Prattova algoritmu však došlo ke zvětšení kroku při posunu vzorku. Na základě předzpracování lze vzorek posunout o více než jeden krok a urychlit tak celé vyhledávání.

Předzpracování vzorku

Fáze předzpracování vzorku má časovou i prostorovou složitost O(m), kde m je délka vzorku. Analýzou základního vyhledávacího algoritmu dospěl tento algoritmus k možnosti v některých případech posunout hledaný vzorek o více než jeden znak.

Dojde-li k neshodě znaku na určité pozici, můžeme si pro tuto pozici předem spočítat posun, který je možný. Lze si tedy vypočítat KMP tabulku o velikosti m+1. Hledaný vzorek je ukončen znakem konce řetězce ‘\0‘. Velikost KMP tabulky je dána tím, že pokud narazíme na znak ‘\0‘ ve vzorku na pozici m, pak je hledaný vzorek nalezen. Pro každou pozici ve vzorku bude v této tabulce k dispozici hodnota, o kterou se můžeme posunout.

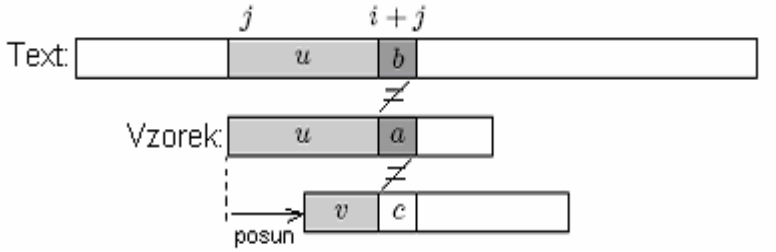
Tabulka KMPtab[i] udává délku nejdelší předpony, která je současně příponou vzorku [0..i-1], kde i je pozice ve vzorku. Rozšiřující podmínkou Morris-Prattova algoritmu je, že znak následující po předponě, tzv. následník, se musí lišit od znaku ve vzorku na pozici [i]. Tato podmínka zabrání neshodě ihned po posunu vzorku. KMPtab[0] se vždy rovná hodnotě -1. Ukázka hodnot KMP tabulky pro jednotlivé indexy vzorku je v tabulce 2.1.

Index:	**0**	**1**	**2**	**3**	**4**
Vzorek:	a	b	a	b	\0
KMPtab[index]:	-1	0	-1	0	2
Předpona(následník):		(a)		(a)	ab(a)
Přípona(následník):		(b)		(b)	ab(b)

Tabulka 2.1 *- Tabulka vytvoření KMPtab předzpracovávaného vzorku abab.*

Vyhledávání vzorku v textu

Knuth-Morris-Prattův algoritmus vyhledávání má časovou složitost O(m+n). Vzorek je v textu vyhledáván zleva doprava. Prohledávání textu není nijak závislé na velikosti použité abecedy. Při vyhledávání je použita předem vytvořená KMP tabulka. Porovnání provádíme v textu y[0..n] znaků se vzorkem x[0..m] znaků, postupným posunem vzorku. Dojde-li k neshodě mezi znakem na pozici x[i] a y[i+j], kde i je hodnota ve vzorku od 0<i<m a j je pozice v textu, pak je z tohoto zřejmé, že znaky na pozicích x[0..i-1] a y[j...i+j-1] se musejí rovnat. Z ukázky posunu vzorku lze usoudit, že část přípony textu *u* se bude rovnat předponě vzorku *v*, viz. obrázek 2.2. Při neshodě se posouvá vzorek tak, aby se na této pozici objevil znak z tabulky KMPtab pro daný index.



Obrázek 2.2 *– Ukázka posunu Knuth-Morris-Prattova algoritmu.*

Hodnota -1 v tabulce KMPtab na indexu 0 udává posun v textu o jednu pozici a porovnání s prvním znakem vzorku. Pokud dojde u porovnání textu se vzorkem k neshodě například na pozici vzorku = 4, pak je z tabulky KMPtab viz. tabulka 2.1. pro tento index zřejmé, že vzorek byl nalezen. Na tomto místě se totiž ve vzorku nachází znak '\0' a vzorek můžeme podle tabulky 2.1 následně posunout o 4 – 2 znaky, tedy o 2 znaky dopředu. Ukázka vyhledávání podřetězce (vzorku) abab za použití tabulky 2.1 je znázorněna na obrázku 2.3.

1.krok:

Text:	**a**	**b**	**a**	**c**	a	b	c	a	b	a	b
	✓	✓	✓	×							
Vzorek:	**a**	**b**	**a**	**b**							

Posun o 3 (index - KMPtab[index] = 3-0)

2.krok:

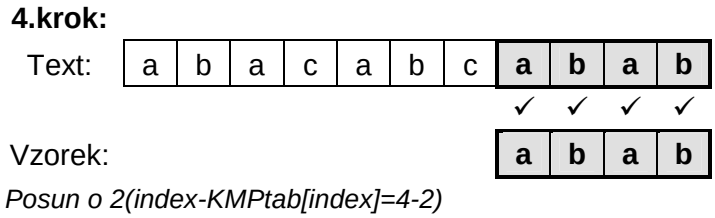
Text:	a	b	a	**c**	**a**	**b**	**c**	a	b	a	b
				×							
Vzorek:				**a**	**b**	**a**	**b**				

Posun o 1(index - KMPtab[index]=0-(-1))

3.krok:

Text:	a	b	a	c	**a**	**b**	**c**	**a**	b	a	b
					✓	✓	×				
Vzorek:					**a**	**b**	**a**	**b**			

Posun o 3(index - KMPtab[index]=2-(-1))



Obrázek 2.3 *– Postup při vyhledávání vzorku v textu KMP algoritmem.*

Testováním bylo zjištěno, že při vyhledávání za pomoci Knuth-Morris-Prattova algoritmu je porovnáváno maximálně (2*n–1) znaků. Jeho nespornou výhodou je to, že se nevrací v textu zpět. Při porovnání obyčejného textu není však použití elementárního algoritmu o moc horší než použití KMP algoritmu, jehož silnou stránkou je porovnání často se vyskytujících částečných shod.

2.3.2 Boyer-Moore (BM) algoritmus

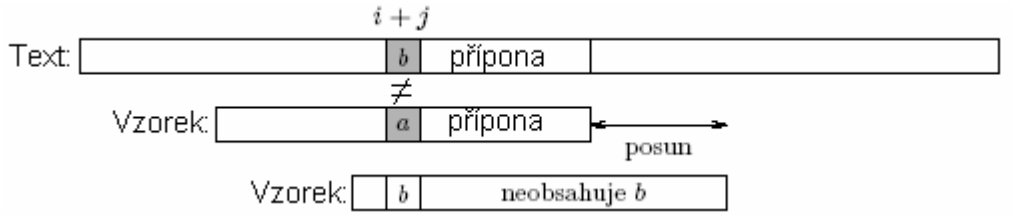
Tento algoritmus využívá pro předzpracování vzorku dvojici heuristik a z výsledku obou vybírá tu s větším krokem posunu vzorku při neshodě pro daný index. Je považován za jeden z nejlepších vyhledávajících algoritmů. Tento algoritmus je používán v jednoduchých textových editorech k prohledávání textu.

Heuristika

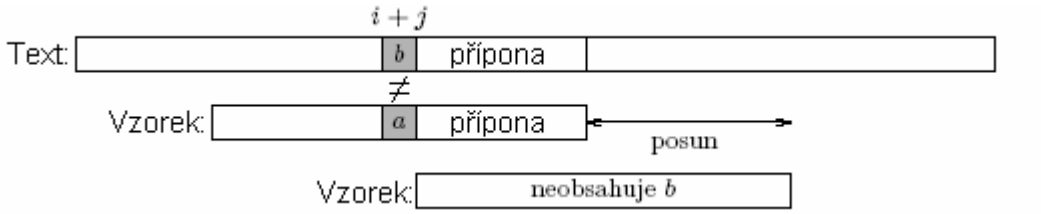
Obecně se jedná o postup vhodný k nalezení řešení bez znalosti či existence konkrétního algoritmu. Použitím heuristiky dosáhneme nalezení řešení, ale nemusí k tomu dojít vždy. Při nevhodných vstupních datech si s těmito daty nemusí heuristický algoritmus vůbec poradit nebo se jejich použitím razantně zvýší čas výpočtu programu. Obecná heuristická funkce neovlivní přímo výsledek programu, ale dokáže zkrátit čas výpočtu. U Boyer-Moorova algoritmu pro vyhledávání jsou obě heuristiky tak dobře provedeny, že jejich použití dokáže výrazně urychlit prohledávání v textu.

První heuristika BM algoritmu

Tato heuristika je závislá na rozsahu použité abecedy. Je nazývána též jako charakteristika „špatného“ znaku. Úkolem této heuristiky je nalezení nejpravějšího výskytu znaku v hledaném vzorku. Při předzpracování první heuristikou mohou nastat dvě situace. Při porovnání znaku v textu na pozici i + j se snažíme nalézt nejpravější výskyt tohoto znaku ve vzorku (viz. obrázek 2.4) a uložit do tabulky hodnotu tohoto nejpravějšího znaku. Druhá situace nastává, pokud se hledaný znak v celém vzorku nenachází. V tom případě můžeme vzorek posunout tak, aby další porovnání začínalo na pozici i+j+1 v textu (viz. obrázek 2.5), tedy až za porovnávaným znakem.



Obrázek 2.4 *– Hledaný znak b se ve vzorku nachází.*

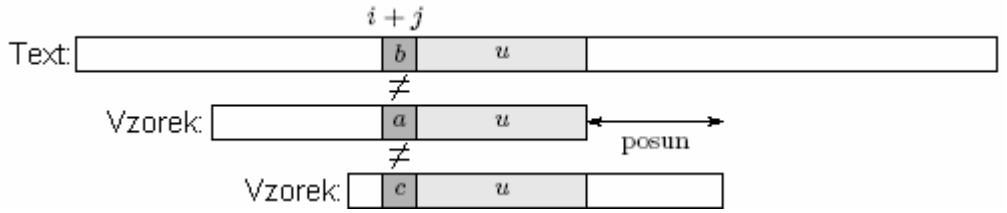


Obrázek 2.5 *– Hledaný znak b se ve vzorku nenachází.*

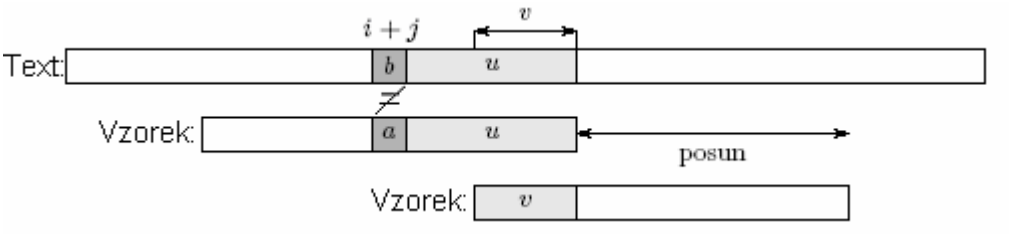
Pro tyto situace si tedy předem vypočítáme tabulku „špatného“ znaku pro všechny znaky použité abecedy (viz. tabulka 2.2). Každý znak použité abecedy bude mít tedy předem přidělenou hodnotu na základě pozice ve vzorku. Tato hodnota odpovídá minimální vzdálenosti znaku k pravému okraji vzorku. Znaky, které se ve vzorku nenacházejí, budou mít přidělenu hodnotu odpovídající délce vzorku. Nevýhodou této heuristiky je to, že při výpočtu posunu vzorku může nabývat hodnot menších jak nula. K tomuto dochází, je-li nejpravější hodnota ve vzorku hledaného znaku již za prohledaným indexem v textu. Tímto by došlo k posunu vzorku zpět v textu. Z tohoto důvodu je k dispozici druhá heuristika, která tento problém řeší a při vyhledávání se použije větší z těchto hodnot. Další nevýhodou je paměťová složitost pro značně rozsáhlé abecedy znaků, kde se musí pro každý znak uchovávat jeho hodnota. Výhodou však je značné zvetšení kroku v případech, kdy znaky obsažené v textu ve vzorku vůbec nejsou.

Druhá heuristika BM algoritmu

Hlavní myšlenkou této heuristiky je nalezení nejdelší vhodné přípony. Je vhodná v případech překrývajících se shodných částí vzorku s textem. Dojde-li k neshodě s textem na pozici i+j a vzorkem na pozici i, pak můžeme prohlásit, že vzorek na pozici [i+1..m-1] je shodný s textem na pozici [i+j+1..j+m-1], kde m je délka vzorku. Hledáme tedy nejpravější výskyt vzorku se shodnou částí textu. Za této situace jsou opět k dispozici dvě možnosti nalezeného řešení. První možností je nalezení shody přípony vzorku *u* s nejpravější shodnou částí vzorku, za předpokladu, že znaky na pozici *a* v textu a *c* ve vzorku jsou od sebe odlišné (viz obrázek 2.6). Druhou možnosti je shoda nejdelší části přípony *u* s její předponou ve vyhledávaném vzorku (viz obrázek 2.7). Předpona vzorku *v* se tedy srovná s částí přípony textu *u*.



Obrázek 2.6 *– Posun na nejpravější výskyt shody přípony vzorku u.*



Obrázek 2.7 *– Nalezení shody části přípony s předponou ve vzorku a jeho posun.*

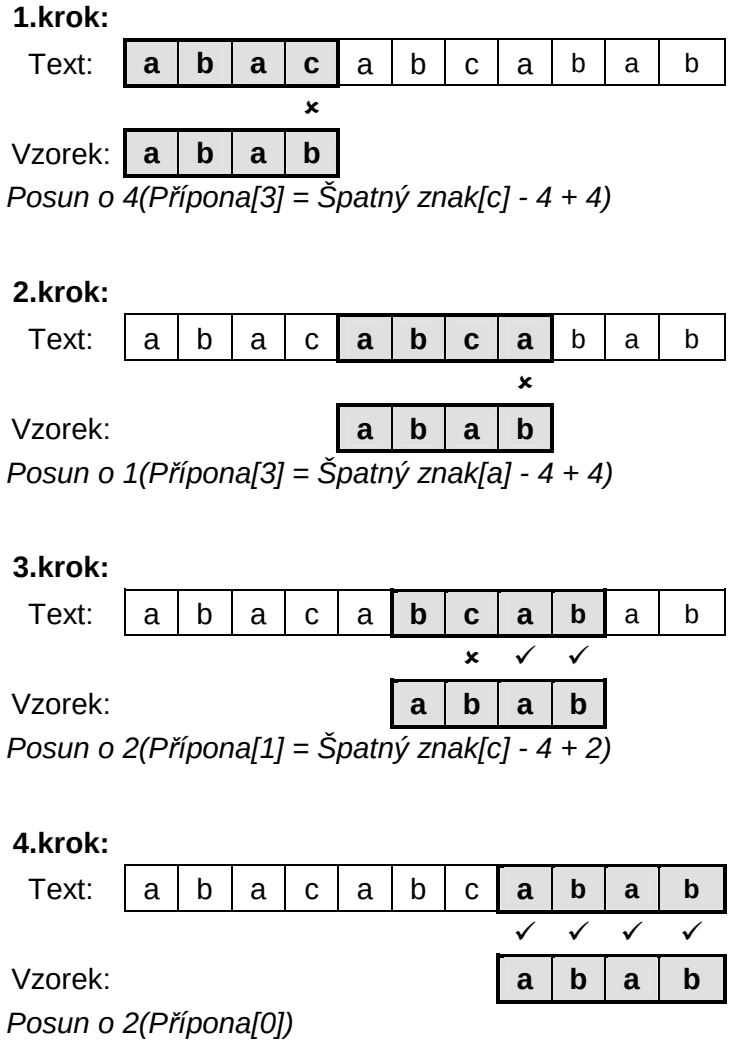
V přepočítané tabulce Good[i] tedy budeme mít uloženy hodnoty, o které chceme daný vzorek posunout při neshodě na dané pozici vzorku (viz. tabulka 2.2). Výhodou je, že tato heuristika nikdy nemůže nabývat záporných hodnot. Proto vždy při prohledávání můžeme posunout vzorek v textu o minimálně jednu pozici vpřed oproti první heuristice, která mohla po přepočítání hodnoty z tabulky podle aktuální pozice nabývat záporných hodnot.

Index:	**0**	**1**	**2**	**3**	
Znaky vzorku:	a	b	a	b	ostatní
Špatný znak[znak]:	1	2	1	2	4
Good[index]:	2	2	4	1	

Tabulka 2.2 *– Ukázka tabulky po předzpracování vzorku první a druhou heuristikou.*

Vyhledávání vzorku v textu

Boyer-Moore algoritmus se řadí do skupiny algoritmů prohledávajících vzorek v opačném směru toku textu, tedy zprava doleva. Vzorek je posouván doprava, ale porovnáván je s textem vždy odzadu. Při vyhledávání se při neshodě na daném indexu použije lepší z hodnot předzpracovaných tabulek první a druhé heuristiky vzorku. V obou tabulkách je spodní hodnota kroku, o který se může vzorek posunout. Vybírá se tedy větší z hodnot obou tabulek. Před samotným porovnáním obou tabulek se musí přepočítat hodnota kroku první heuristiky v závislosti na pozici ve vzorku. Ukázka vyhledání vzorku abab podle předzpracované tabulky 2.2 je na obrázku 2.8. K nalezení vzorku dojde v případě posunu ve vzorku na hodnotu indexu = 0. Časová i prostorová složitost předzpracování vzorku je O(m+σ), kde m je délka vzorku a σ je rozsah použité abecedy. Pro nejhorší případ vyhledávání je časová složitost tohoto algoritmu O(m*n), kde n je délka textu. Při vyhledávání s neperiodickým vzorkem, dochází u BM algoritmu nejhůře k 3*n porovnání znaků.



Obrázek 2.8 *– Ukázka postupu při vyhledávání vzorku abab v textu pomocí BM algoritmu.*

Z obrázku 2.8 je vidět, že bylo provedeno 9 porovnání vzorků s textem. U KMP algoritmu z obrázku 2.3 bylo porovnáváno 12 znaků. Je tedy vidět urychlení u Boyer-Moore algoritmu již pro takto krátký a jednoduchý vyhledávaný vzorek.

2.3.3 Shrnutí efektivity BM a KMP algoritmu

Pro porovnání obou těchto vyhledávacích algoritmů je vhodné si na závěr uvést jejich časovou a prostorovou složitost. Pro porovnání zde uvedu také složitost elementárního algoritmu viz. tab. 2.3.

Složitost: / *Algoritmus:*	**předzpracování**	**vyhledávání**	**prostorová**
Elementární	O(1)	O(m*n)	O(1)
KMP	O(m)	O(m+n)	O(m)
BM	O(m+σ)	O(m*n)	O(m+σ)

Tabulka 2.3 *– Porovnání jednotlivých algoritmů pro vyhledávání vzhledem ke složitosti*

Nutno říci, že tabulka je uvedena pro nejhorší případ vyhledávání, například pro text aaaaa a vyhledávaný vzorek a. Je zde vidět, že pro tento případ dosahuje BM algoritmus hodnoty časové složitosti vyhledávání stejné jako elementární algoritmus. Nedochází tak k výraznému urychlení. U vzorku aaab a textu aaaaab je naopak BM algoritmus výrazně efektivnější než elementární algoritmus, kde se hodnota času vyhledání nezmění. Dalším zajímavým údajem je závislost BM algoritmu na rozsahu abecedy u prostorové složitosti. Tato vlastnost je nepříjemná při značném rozsahu abecedy např. Unicode, kde musíme uchovat značné množství dat a je třeba se pokusit tuto hodnotu zkorigovat.

2.3.4 Ostatní specializované algoritmy

Existuje mnoho dalších vyhledávacích algoritmů uvedených v literatuře [1]. Rád bych zde stručně uvedl alespoň některé z nich, u kterých je styl vyhledávání či předzpracování odlišný od předchozích uvedených algoritmů a nejsou probírány v předmětu Algoritmy.

Karp-Rabinův algoritmus

Tento algoritmus využívá k vyhledávání hashovací funkce[1]. Namísto porovnávání přesné části textu je porovnáváno na základě výsledku hashovací funkce. Prohledávají se ty části textu, které jsou podobné uvedenému vzorku a jsou uloženy na místě hashovací tabulky. Snahou tohoto algoritmu je omezení kvadratické časové složitosti vyhledávání. Aby se dosáhlo efektivního vyhledávání musí být hashovací funkce jednoduše vypočitatelná, snadno měnitelná, odolná proti kolizím a musí umožnit snadno vypočítat následující hodnotu hashovací funkce z předchozí hodnoty.

Shift-Or algoritmus

Činnost tohoto algoritmu je založena na použití bitových polí. Velikost bitového pole je dána délkou vzorku a uvnitř tohoto pole jsou uloženy hodnoty předpony vzorku končící na aktuální pozici v textu. Tato hodnota je aktualizována po každém přečtení nového znaku v textu. Časová i paměťová složitost předzpracování vzorku je závislá na velikosti abecedy a délce vzorku a je určena jako $O(m+\sigma)$. Vyhledávání však už není závislé na velikosti vzorku, ale pouze na velikosti prohledávaného textu.

Horspoolův algoritmus

Jedná se o zjednodušený Boyer-Moore algoritmus. Používá se v případech, kdy délka vzorku je výrazně menší než rozsah použité abecedy. Je zde tedy málo pravděpodobné, že by se mohla část vzorku překrývat. Proto Horspoolův algoritmus pracuje s vynecháním tabulky vypočítané pomocí druhé heuristiky u BM algoritmu bez velkého snížení efektivity při vyhledávání.

---

[1] Hashovací funkce: Převádí vstupní posloupnost bitů různé délky na posloupnost bitů pevné délky, tzv.hash.

3 Implementace domácích úloh

Tato kapitola bude orientována na popis praktické části implementace jednotlivých domácích úloh a testovacích souborů. Jednotlivé zdrojové kódy úloh jsou uloženy na přiloženém CD. Tyto zdrojové kódy jsou velice podrobně okomentovány. Zde se budu zabývat problematikou, na kterou jsem narazil při vytváření a stručným rozborem postupu při implementaci těchto úloh.

3.1 Rozdělení souborů a jejich obsah

Z důvodu tvorby ucelené sbírky úloh bylo nutné podle toho také zařídit tvorbu zdrojových kódů. Každá domácí úloha je složena z několika souborů potřebných pro správnou činnost. Toto rozdělení zdrojových kódů musí být zachováno také pro systém automatické kontroly a zadávání úloh.

Každá složka se vzorovým řešením domácích úloh musí obsahovat alespoň tyto zdrojové kódy:

- **Soubor.c:** Soubor, kde jsou implementovány jednotlivé funkce domácích úloh deklarované v souboru s rozhraním. Do tohoto souboru má student za úkol doimplementovat těla potřebných funkcí podle zadaného algoritmu a dle komentářů v hlavičkách funkcí.
- **Soubor.h:** Jedná se o soubor s rozhraním, ve kterém definujeme datové struktury a deklarujeme funkce použité pro manipulaci s těmito datovými strukturami. Jsou zde navíc umístěny datové struktury potřebné pro testování úloh.
- **Soubor-test.c:** Program, ze kterého jsou volány jednotlivé funkce deklarované v rozhraní pro ověření správnosti implementace. Dochází zde ke spouštění jednotlivých testovacích úloh. V tomto souboru také kontrolujeme, zda byly funkce implementovány či nikoliv.
- **Soubor-advanced-test.c:** Soubor s rozšířenou verzí testovacích úloh. Testy okrajových podmínek, které nejsou obsaženy v souboru soubor-test.c.
- **Soubor.output:** Soubor s přesným vzorem řešení jednotlivých testů domácích úloh.
- **Makefile:** Soubor potřebný pro sestavení a překlad domácích úloh.

Soubor s rozhraním nám umožňuje spojit testovací soubor se souborem implementací jednotlivých domácích úloh. Student je při implementaci donucen použít datové struktury uvedené v tomto rozhraní a zároveň je nucen implementovat funkce deklarované v tomto rozhraní, aby mohl obdržet požadované bodové hodnocení z těchto úloh. Pro novou sbírku úloh vyhledávání podřetězců jsem navrhnul zařadit vyhledávací funkce za pomoci elementárního a Knuth-Morris-Prattova algoritmu do souborů s názvem začínajícím kmp a pro algoritmus Boyer-Moore soubory začínající názvem bm. Pro porovnání všech těchto algoritmů jsem vytvořil navíc složku s názvem one, kam jsem umístil všechny tyto algoritmy do jediného zdrojového kódu.

3.2 Implementace vyhledávacích metod

K vytváření nových typů domácích úloh pro vyhledávání podřetězců v řetězcích bylo třeba navrhnout datové struktury a pro jednotlivé úlohy vytvořit vzorově jejich implementace tak, aby bylo možné automaticky porovnávat řešení studentů a naše vzorová řešení.

3.2.1 Implementace KMP algoritmu

Při implementaci Knuth-Morris-Prattova algoritmu je nutné implementovat dvě funkce pro korektní činnost tohoto algoritmu. První je funkce pro předzpracování vzorku. Druhou funkcí je samotné vyhledávání v textu. Pro implementaci obou těchto funkcí jsem zvolil soubor kmp.c. Do tohoto souboru jsem navíc přidal studentovi k dopracování elementární algoritmus. Mojí snahou bylo, aby byl ve výsledku testu vidět rozdíl mezi kroky elementárního a Knuth-Morris-Prattova algoritmu. Mělo by zde být také vidět, že délka zdrojového kódu jednotlivých algoritmů není nejdůležitějším faktorem při volbě efektivních vyhledávacích algoritmů.

V úvodu implementace bylo třeba vytvořit rozhraní s jednotlivými datovými strukturami, které budou použity v těchto funkcích. Návrh tohoto rozhraní je uložen v souboru kmp.h. Pro fázi předzpracování vzorku bylo nutné studentovi poskytnout strukturu, do které bude ukládat nastavení po předzpracování pro konkrétní vzorek. Součástí této struktury je také informace o vyhledávaném vzorku a prohledávaném textu. Poslední položkou této datové struktury je pozice v textu, na které byl daný vzorek nalezen. Pro jednoduchost jsem zvolil k uložení textu a vzorku statické pole znaků zakončené znakem ‘\0‘. Velikost statického pole je možno změnit na jinou či změnit na dynamicky implementované, ale pro pochopení, jak vyhledávací algoritmy fungují, studentům pro tyto úlohy postačuje staticky nastavená velikost řetězců.

```
typedef struct {
  char text[MAXTEXT];       /* obsahuje prohledávaný text */
  int delkatext;            /* uložena délka textu */
  char vzor[MAXVZOR];       /* obsahuje vyhledávaný vzorek */
  int delkavzor;            /* obsahuje délku vzorku */
  int next[PREFIX];         /* tabulka pro předzpracování vzorku */
  int nasel;                /* sem se ukládá pozice nalezeného v textu */
}tKMP;
```

Obrázek 3.1 *– Ukázka datové struktury pro Knuth-Morris-Pratt algoritmus.*

```
void tabulkaNext(tKMP* );          /* hlavička funkce pro předzpracování vzorku */
void KMPalgoritmus(tKMP* );     /* hlavička funkce pro vyhledávání vzorku v textu */
```

Obrázek 3.2 *– Hlavičky funkcí pro implementaci Knuth-Morris-Pratt algoritmu.*

Úkolem studenta je doplnit funkce, jejichž hlavičky jsou znázorněny na obrázku 3.2. Jediným parametrem těchto funkcí je ukazatel na datovou strukturu znázorněnou na obrázku 3.1. Podrobný komentář k jednotlivým funkcím je součástí zdrojových kódů. Pro předzpracování vzorku má student za úkol dopracovat funkci tabulkaNext( ). Úkolem této funkce je doplnit datovou strukturu tKMP tak, aby se mohla zcela využít ve fázi vyhledávání vzorku. Tato datová struktura poskytuje studentovi tabulku next[], kterou musí student naplnit podle principu algoritmu popsaného v kapitole 2.3.1 tak, aby postup při vyhledávání vzorku pomocí tohoto algoritmu byl správný. Pro snadné pochopení, jak s touto tabulkou pracovat, jsem zvolil pole prvků. Pole prvků umožňuje snadný přístup k jednotlivým indexům tabulky a umožňuje efektivně zkontrolovat správné nastavení po předzpracování. Bez správné implementace funkce pro předzpracování nebude správný ani postup vyhledávání vzorku.

Implementace funkce KMPalgoritmus( ) pro vyhledávání vzorku se řídila postupem vytváření KMP algoritmu popsaném v kapitole 2.3.1. Problém nastal při potřebě kontrolovat indexy, které student při porovnávání vzorku s textem prošel. Bylo by snadné nechat studenta implementovat funkci pro vyhledávání způsobem, kterým on sám chce. Nastal by však problém v nemožnosti kontroly použitého algoritmu. Na rozdíl od fáze předzpracování, kdy nás zajímal pouze výsledek naplnění tabulky po předzpracování a nebylo tak třeba kontrolovat, jakým postupem student tuto tabulku doplnil. Ve fázi prohledávání textu je naopak nutné zkontrolovat postup vyhledávání vzorku.

Problém implementace vyhledávání

Ve fázi vyhledávání vzorku bylo třeba vyřešit problém kontroly indexů při průchodu prohledávaným textem. Bez použití speciálních operací (funkcí) se tento problém jevil jako neřešitelný. Objevilo se tedy několik způsobů možných řešení, které zde podrobněji rozeberu:

- **1.způsob:**

Prvním řešením, které mě napadlo, bylo vytvoření funkce, které by student jako parametry předal index vyhledávané pozice ve vzorku a v textu. Tato funkce by měla za úkol v každém průchodu zkontrolovat a vytisknout na standardní výstup informace o procházených indexech a posunu získaném z tabulky předzpracování vzorku. Toto řešení by bylo možné, ale znemožnilo by při následném testování vracet se k prošlým indexům a s těmito indexy pracovat v různých testovacích funkcích. Kontrolní výpisy by však musely být také součástí samotné vyhledávací funkce.

- **2.způsob:**

Dalším možným řešením by bylo vytvořit například globální pole prvků či struktury, které by byly přístupné z testovacích i vytvářených funkcí. V komentářích k jednotlivým funkcím doplnit postup, který po studentovi vyžadujeme, aby provedl při jednotlivých průchodech textem. Toto řešení by bylo také možné, ale případné nepochopení studenta, co po něm v zadání požadujeme, by vedlo k jeho ztrátě bodového hodnocení. Navíc by bylo krajně nepříjemné a neefektivní při každém průchodu ukládat jednotlivé indexy do pole či do struktury. Student by při každé změně musel uložit všechny potřebné informace. V případě, že by nedoplnil jedinou hodnotu, vedl by výsledek jeho

snažení ke špatnému výsledku při kontrole algoritmu. Z tohoto důvodu jsem tento způsob řešení zamítnul.

- **3.způsob:**

Konečný způsob řešení, který jsem nakonec při implementaci úloh využil. Tento způsob se velice podobá prvnímu způsobu možného řešení, ale jsou zde některé odlišnosti. Implementace spočívá ve vytvoření speciální funkce vratPozici( ) pro Knuth-Morris-Pratt algoritmus. Tato funkce vyžaduje jako parametry aktuální pozici ve vzorku a ukazatel na datovou strukturu tKMP. Návratovou hodnotou této funkce je nová pozice ve vzorku na základě informací z tabulky předzpracování vzorku pro daný index. Úkolem této funkce je naplnění seznamu prošlých indexů. Pro tento seznam je k dispozici speciální datová struktura testKMP, kam se ukládají jednotlivé informace nutné pro otestování správnosti implementace algoritmu. Jsou zde uloženy informace o pozici v textu, pozici ve vzorku a pozici nalezeného vzorku v textu. Výhodou tohoto seznamu je snadné přidávání dalších položek a snadné procházení položek při testování. Jediným naším úkolem je studenta donutit v komentářích k funkci k použití této pomocné funkce. Student tuto pomocnou funkci využije při potřebě získat informace z tabulky next[] (předzpracování). Informace z této tabulky je nutné získat při každé změně pozice ve vzorku v závislosti na aktuální pozici vzorku. Bez použití této pomocné funkce však student nezíská bodové hodnocení, což je pro něho velmi silnou motivací.

Pro elementární algoritmus jsem také vytvořil pomocnou funkci, kterou student musí využít při každé změně pozice v textu. Jedná se o funkci nazvanou norm( ). Úkolem této funkce je uložit indexy vzorků, na kterých došlo k neshodě s prohledávaným textem. Tento index musí být znám pro každý index pozice v textu. Nemá smysl zatěžovat studenty implementací těchto funkcí, které nejsou podstatné pro pochopení algoritmů vyhledávání podřetězců v řetězcích, ale jsou podstatné pro testovací fázi. Implementaci těchto pomocných funkcí si student má možnost prohlédnout v souborech s testy domácích úloh a nejsou pro něho tedy nijak skryté. Po přeložení zdrojových souborů je možné otestovat oba tyto algoritmy zadáním příkazu přesměrování ./kmp-test > kmp.txt.

3.2.2 Implementace BM algoritmu

Boyer-Moore algoritmus je vytvářen za pomoci první a druhé heuristiky a fáze vyhledávání. Bylo tedy nutné vytvořit nejméně tři funkce, aby implementace odpovídala použitému algoritmu. Ukázka datové struktury pro Boyer-Moore algoritmus a hlavičky funkcí, které je nutné implementovat jsou znázorněny na obrázku 3.3 a 3.4. Pro tento algoritmus jsem zvolil soubory s předponou bm. Jediným parametrem těchto funkcí je ukazatel na strukturu tBM.

Ve fázi předzpracování bylo třeba doplnit tabulku znak[] za pomoci první heuristiky a tabulku Good[] za pomoci heuristiky druhé. Počet naplněných hodnot u tabulky znak[] je roven rozsahu abecedy, který jsem pro naše řešení použil jednoduchou ASCII tabulku tzn. 255 znaků. Naopak

u druhé heuristiky byl počet naplněných hodnot roven délce vzorku. Není tedy problém zkontrolovat správnost nastavení obou těchto tabulek. Při implementaci tabulky pomocí druhé heuristiky jsem se rozhodl využít pomocné tabulky přípona[], která usnadní následné naplňování tabulky Good[]. Pro každou pozici ve vzorku si předem vypočítám délku nejdelší předpony pro daný index, která je současně příponou vzorku a uložím si hodnotu do pole přípona[]. Tuto funkci poté využiji pro výpočet tabulky posunu na základě druhé heuristiky, kde pro každý index znám délku nejdelší předpony z tabulky přípona[] a z této hodnoty není problém vypočítat bezpečný posun vzorku.

Ve fázi vyhledávání ve funkci BMalgoritmus( ) je třeba vybrat větší z obou hodnot tabulek znak[] a Good[] pro daný index vzorku, na kterém došlo k neshodě. V tabulce znak[] je však uložena hodnota posunu vzorku pro daný znak textu a tuto hodnotu je třeba přepočítat před samotným porovnáním na hodnotu posunu vzhledem k pozici ve vzorku. Pro ověření správnosti implementace Boyer-Moore algoritmu bylo třeba zjistit pozice, které student při vyhledávání prošel a kterou heuristiku použil. Bylo tedy nutné analyzovat algoritmus a nabídnout řešení tohoto problému, obdobné způsobu z kapitoly 3.2.1. Byl tedy opět vybrán třetí způsob řešení s mírnou úpravou. Byla tedy vytvořena pomocná funkce novaPozice( ) s parametry pozice v textu, posun, pozice ve vzorku a ukazatelem na strukturu. Tato funkce vrací hodnotu nové pozice v textu, na kterou se máme posunout na základě druhého parametru posunu. Tento druhý parametr je hodnota určená na základě informace získané z větší z obou heuristik. Funkce má navíc tento parametr, aby se zjistilo, zda student použil správnou heuristiku a zda správně přepočítal posun první heuristiky. Součástí této funkce je opět uložení všech potřebných informací do seznamu, aby se mohl způsob implementace v testovací části zkontrolovat. Tuto funkci musí student použít, vyžaduje-li změnu pozice v textu na základě informací z tabulky znak[] nebo Good[]. Bez použití této funkce by nebylo možné ověřit správnost použité heuristiky. Otestovat tento algoritmus je možné přesměrováním ./bm-test > bm.txt.

```
typedef struct {
  char text[MAXTEXT];        /* obsahuje prohledávaný text */
  int delkatext;             /* uložena délka textu */
  char vzor[MAXVZOR];        /* obsahuje vyhledávaný vzorek */
  int delkavzor;             /* obsahuje délku vzorku */
  int Good[PREFIX];          /* tabulka pro předzpracováni druhou heuristikou */
  int pripona[PREFIX];       /* pomocná tabulka pro uložení délky přípony */
  int znak[ABECEDA];         /* tabulka pro uložení informaci první heuristikou */
}tBM;
```

Obrázek 3.3 *– Ukázka datové struktury pro Boyer-Moore algoritmus.*

```
void tabulkaZnak(tBM* st);      /* Funkce první heuristiky */
void tabulkaGood(tBM* st);      /* Funkce druhé heuristiky */
void BMalgoritmus(tBM* st);     /* Funkce pro vyhledávání v textu */
```

Obrázek 3.4 *– Hlavičky funkcí pro implementaci Boyer-Moore algoritmu.*

3.3 Testovací úlohy vyhledávání

Při tvorbě testovacích úloh pro vyhledávání podřetězců v řetězcích bylo třeba brát v úvahu složitost vypracování jednotlivých domácích úloh. V porovnání s implementací lineárního seznamu, operací s frontou či zásobníkem se jedná o mnohem složitější úlohy. Této skutečnosti bylo třeba přizpůsobit složitost a názornost jednotlivých testů vyhledávacích úloh. Názornost však byla omezena použitím jazyka C, kde pro tisk výsledků na výstup lze použít libovolné znaky z ASCII tabulky.

Po zjištění, zda danou úlohu student řešil pomocí použití proměnné solved rozebrané v kapitole 4.1.1, je možno přistoupit k testování jednotlivých úloh. Jednotlivé testy pro ověření správnosti implementace vyhledávacích metod pomocí algoritmu Knuth-Morris-Pratt a Boyer-Moore lze rozdělit do několika základních skupin:

- **Předzpracování vzorku:** U těchto úloh je předzpracování vzorku základním faktorem pro úspěšné nalezení vzorku v textu. Proto jsem se rozhodl vytvořit testy pro ověření správnosti nastavení datových struktur pro jednotlivé algoritmy po předzpracování vzorku. Součástí těchto testů jsou výpisy jednotlivých tabulek (datových struktur) tak, aby si student mohl ověřit jejich správnost. Ukázku výpisu v testovacím prostředí jazyka C vytvořeného pro testování korektního předzpracování vzorku naleznete na obrázku 3.5.
- **Vyhledání vzorku:** Pokud by student ve fázi implementace předzpracování vzorku udělal chybu, či pokud by se pokusil vytvořit implementaci bez předzpracování vzorku, testovací úlohy by tento fakt odhalily. V případě, že studentova implementace proběhla úspěšně, je k dispozici několik testů pro samotnou funkci vyhledávání vzorku.
- **Postup vyhledávání:** U postupu vyhledávání nás obvykle nezajímá obsah prohledávaného textu, ale indexy, které student při vyhledávání vzorku prošel. Tímto dokážeme ověřit správnost implementace zadaného algoritmu ze zadání úlohy. Bylo tedy nutné donutit studenta k použití funkcí předem připravených pro přístup k jednotlivým prvkům struktury. Bez použití těchto funkcí by nebylo možné zjistit postup, jakým student danou úlohu řešil. Tato problematika při vytváření a funkce, které musí student při řešení jednotlivých vyhledávacích metod použít, jsou podrobněji popsány v předchozí kapitole 3.2. Do testů jsem tedy přidal výpisy počátečních indexů, na kterých zadaný vzorek začínal prohledávání. Ukázky textových výpisů testů vyhledávání vzorku algoritmem Boyer-Moore naleznete na obrázku 3.6 a 3.7.
- **Počet nalezení:** Postup prohledávání textu závisí na použitém algoritmu, ale počet nalezených vzorků ve shodném textu musí být stejný pro všechny algoritmy vyhledávání. Postupným prohledáváním textu lze nalézt několik částí textu, které odpovídají hledanému vzorku, proto je vhodné mít k dispozici testy pro ověření počtu nalezených vzorků v textu.
- **Index nalezeného:** Dalším testem je index nalezeného vzorku. Bylo třeba zjistit, na kterém místě v textu se nalezený vzorek nalézá. Je tedy vypisována počáteční pozice v textu.

Předem popsané typy testů jsou společně se vzorovým řešením poskytnuty studentům k ověření správnosti implementace. Tyto testy patří mezi ty základní. Jsou uloženy v souboru s názvem-test.c. Musím však přiznat, že již shoda základních testů se vzorovým řešením je ukázkou pochopení dané problematiky a na výsledku bodového hodnocení se to značně projeví. Profesorům však musí být k dispozici testy pokročilé. Testy pokročilé se nalézají v souboru název_úlohy-advanced-test.c. Rozhodl jsem se tedy rozšířit testy o okrajové podmínky a další možnosti při vyhledávání. Základním rozšiřujícím testem je pokus o vyhledání prázdného vzorku. Student musí mít tuto část ošetřenu. Dalším pokusem je vyhledání vzorku v prázdném textu. Nemělo by se stát, že dojde k nalezení zadaného vzorku v prázdném řetězci znaků. Posledním rozšiřujícím testem je vyhledání prázdného vzorku v prázdném textu.

Jako první z testů jsem studentům vytvořil test na elementární vyhledávací algoritmus. Pokud studenti implementují funkci pro tento algoritmus, mohou si porovnat výsledek tohoto jednoduchého algoritmu s algoritmem Knuth-Morris-Pratt. Pro názornost urychlení algoritmu pro některé vzorky obsahující částečné shody se počet kroků mezi oběma algoritmy výrazně liší. U některých vyhledávaných vzorků však není poznat rozdíl mezi způsobem prohledávání obou těchto algoritmů. Další testy obsahují výpisy tabulek pro předzpracování vzorku, které student musí doplnit a to buď do struktury tKMP nebo tBM. Samozřejmostí jsou testy pro vyhledání vzorku, který se v textu nachází pouze jednou, vícekrát či vůbec. Pro názornost jsem vytvořil tzv. „upovídané testy“. Tyto testy mají za úkol ukázat studentovi přesně průchod, jakým se podle něho porovnávalo. Pro každý krok je zde ukázka pozice v textu a vzorku, na které se znaky rovnají, a na které se liší. Součástí tohoto je také informace o následujícím vypočítaném posunu vzorku. Tyto testy jsou pouze názorné a nehodí se pro rozsáhlé texty. Naopak pro rozsáhlé testy se hodí zkrácené testy. Tyto testy vypisují na výstup pouze prohledávaný text, vyhledávaný vzorek, počáteční pozici každého nalezeného vzorku v textu a počet nalezených vzorků v celém textu. Dále je vypisován seznam prošlých pozic textu. Ukázky obou typů výsledků testů vytvořených v jazyce C jsou na obrázku 3.6 a 3.7.

Hlavní výhodou základních testů je možnost jejich rozšíření studenty. Tyto testovací úlohy jsou poskytnuty studentům, ale nejsou součástí odevzdání úlohy, proto má student možnost dopracování a libovolné úpravy těchto testů. Může si základní testy doplnit tak, aby ve výsledku jeho domácí úloha prošla i testy pokročilými. Při úpravě však musí mít na paměti, že úlohy budou testovány v rozhraní, které bylo studentovi dodáno. Projdou-li studentovi pouze testy základní a neprojdou mu testy pokročilé, získá pouze poměrnou část bodového hodnocení za úlohu.

```
good[0] = znak vzorku[v] = 4
good[1] = znak vzorku[o] = 4
good[2] = znak vzorku[d] = 4
good[3] = znak vzorku[o] = 4
good[4] = znak vzorku[v] = 7
good[5] = znak vzorku[o] = 7
good[6] = znak vzorku[d] = 1
```

Obrázek 3.5 *– Ukázka výpisu po předzpracování vzorku za pomoci BM algoritmu v jazyce C.*

```
Text -> ovodoovovodovoddvo.
Vzor -> vodovod
??? Nalezen vzorek pomoci BM algoritmu na pozici: 8 ???

Výpis prošlých pozic textu:
0, 2, 4, 8, 12,
Počet prohledávaných znaků za pomoci BM algoritmu = 15

Počet nalezených vzorků -> vodovod v textu za pomoci BM algoritmu = 1
-----------------------------------------------------------------
```

Obrázek 3.6 *– Ukázka verze testovacího výpisu pro delší texty za pomoci BM algoritmu v jazyce C.*

```
krok -> 1.
ovodoovovodovoddvo.
      !
vodovod

Další posun v textu o -> 2 na počáteční pozici -> 2
---------------------------------------
krok -> 2.
ovodoovovodovoddvo.
        !
  vodovod
Další posun v textu o -> 2 na počáteční pozici -> 4
---------------------------------------
krok -> 3.
ovodoovovodovoddvo.
      !====
    vodovod

Další posun v textu o -> 4 na počáteční pozici -> 8
---------------------------------------
krok -> 4.
ovodoovovodovoddvo.
       !=======
        vodovod

??? Nalezen vzorek pomoci BM algoritmu na počáteční pozici: 8 ???

Další posun v textu o -> 4 na počáteční pozici -> 12
---------------------------------------
krok -> 5.
ovodoovovodovoddvo.
                 !
            vodovod

---------------------------------------
Celkový počet porovnaných znaků za pomoci BM algoritmu = 15
Počet nalezených vzorků -> vodovod v textu za pomoci BM algoritmu = 1
-----------------------------------------------------------------
```

Obrázek 3.7 *– Ukázka „upovídané“ verze testovacího výpisu za pomoci BM algoritmu v jazyce C.*

4 Systém automatické kontroly

Dříve byly úlohy pro předmět Algoritmy psány v jazyce Pascal. Vytvoření tohoto systému si vynutila potřeba přepracování těchto úloh do jazyka C a nutnost opravovat ručně velké množství studentských prací. Systém byl vytvořen v jazyce Bash, Perl, PHP a je plně funkční na serveru eva. V této kapitole se pokusím nastínit, jak tento systém pracuje a jaké jsou jeho základní součásti. Prvním jeho úkolem je vytvořit automaticky zadání úloh pro studenty ze vzorového řešení úloh. Druhým úkolem systému je po vypracování těchto úloh studenty jejich následné automatické hodnocení.

4.1 Automatické zadávání úloh

Systém pro automatické zadávání je tvořen tak, aby za použití speciálních značek bylo možné připravit domácí úlohy k dopracování studentovi a to bez jakéhokoliv následného zásahu profesorů do zdrojových kódů. Systém však musí mít přístup ke vzorově vyřešeným úlohám.

4.1.1 Speciální značky ve zdrojovém kódu

Po implementaci funkcí pro jednotlivé domácí úlohy se do vzorových souborů musejí doplnit speciální značky, které jsou při přípravě interpretovány interpretem. Tyto značky je třeba doplnit na místa ve zdrojovém kódu, která mají být označena popř. upravena interpretem. Tento interpret je součástí systému pro automatické zadávání a byl vytvořen pouze pro tento systém. Vzorové řešení vyhledávacích domácích úloh se nachází ve složce s názvem kmp, bm a one.
Rozlišujeme tyto typy značek, které jsou speciálně interpretovány systémem:

- **Solved -** globální proměnná, která nám udává, zda byla daná funkce řešena. Uvnitř každé funkce, kterou má student naimplementovat, je nastavena tato proměnná na hodnotu false. V případě, že student danou úlohu řešil, musí tuto proměnou zakomentovat, smazat nebo nastavit na hodnotu true. Při přípravě zadání interpretem je tato hodnota v původním návrhu odkomentována a nastavena tedy na hodnotu false = neřešeno.
- **/**/ -** mezi těmito značkami je text, který má zůstat vygenerován beze změn. Tyto značky jsou použity např. ve vzorovém souboru Makefile pro vygenerování studentského Makefile souboru. Tyto značky mají svůj význam na začátku a na konci každého zdrojového souboru. Určují části textu, které budou studentům poskytnuty v zadání úloh.
- **/*v*/ -** mezi těmito značkami je vzorové řešení jednotlivých funkcí. Toto vzorové řešení spolu s těmito značkami bude při interpretaci systémem ze zdrojového kódu odstraněno a poskytnuto studentům k dopracování.
- **//** - jednořádkový komentář v jazyce C. Tyto značky jsou při generování úloh odstraňovány ze zdrojového kódu, aby bylo možné rozpoznat nastavení globální proměnné solved.

4.1.2 Generování zadání

Pro tvorbu zadání poskytovaného studentům musí být součástí adresářů vzorového řešení úloh speciální Makefile, který má v sobě zabudován mechanismus pro vygenerování zdrojového kódu. Dále musí být k dispozici interpret, který prochází vzorové zdrojové kódy a na základě použitých značek popsaných v kapitole 4.1.1 vytváří soubory pro studenty.

Potřebné části systému pro vygenerování zadání:

- **Interpret:** Jedná se o interpret vytvořený v jazyce C. Interpret musí být uložen ve složce systém/interpret/. Před samotným použitím musí být vytvořen spustitelný soubor interpretu. Jeho konstrukce je tvořena konečným automatem, který postupně prochází jednotlivé zdrojové kódy a odstraňuje všechny části textu, které student nemá ze vzorového řešení obdržet. Interpret se rozhoduje na základě značek popsaných v předchozí kapitole. Ostatní části ponechává v souboru.
- **Makefile:** Po zadání příkazu *make zadani* dojde k předání souborů vzorového řešení speciálnímu interpretu. Po zpracování interpretem Makefile soubor zajistí vytvoření složky se zadáním pro studenty. Tento Makefile musí být uložen ve složce s názvem vzorového řešení. Součástí složky se zadáním bude poté soubor příklad.c, příklad.h, studentský Makefile a soubor příklad-test.c.
- **Vzorové řešení úlohy:** Jedinou ruční činností, kterou musíme provést při vytváření zadání je zkopírování souboru se vzorovým řešením základních testů. Tento soubor příklad.output je nutné ručně překopírovat ze složky se vzorovým řešením do složky zadání. Po úpravě Makefile souboru popsané v kapitole 4.1.3 však již není třeba tuto ruční činnost provádět.

4.1.3 Modifikace systému

Při práci s interpretem jsem narazil na jednu nepříjemnost, která je při vytváření zadání prováděna. Při přípravě zdrojových textů dochází v původním návrhu interpretu, který jsem měl k dispozici k odstranění jednoduchých komentářů //. Jedná se o jednořádkové komentáře, které v jazyce C studenti hojně využívají a jejich odstranění vedlo k chybovým hlášením při kompilaci souborů a nebylo tak možné některá řešení přeložit. Původní interpret odstraňoval tyto lomítka tak, aby rozpoznal použití speciální značky solved. Usoudil jsem, že tato část interpretu není nutná ani vhodná pro jeho korektní činnost. Dle mého je lepším řešením ponechat jednořádkové komentáře ve zdrojových kódech a modifikovat uplatnění globální proměnné solved.

Odstranil jsem tedy ze zdrojového kódu interpretu mazání jednořádkových komentářů. V této chvíli je tedy bez obav možné použít tyto komentáře kdekoliv ve zdrojovém kódu. Nyní je tedy nutné vyřešit použití globální proměnné solved. V případě vzorového řešení úlohy je nutné tuto proměnnou mít ve všech řešených funkcích na hodnotu true = řešeno. Po odstranění vzorového řešení při přípravě úloh pro studenty je naopak nutné nastavit předem tuto hodnotu na false = neřešeno. Změnil jsem

tedy pozici umístění nastavení proměnné. Ve vzorovém řešení jsem tuto proměnou umístil mezi speciální značky /*v*/, které označují odstranění tohoto kódu. Proměnná nastavená zde na hodnotu true označuje vyřešenou úlohu. Naopak ještě před těmito speciálními značkami jsem tuto proměnnou nastavil na hodnotu false. Což při odstranění vzorového řešení vede ke správnému nastavení této proměnné. V testovacích funkcích je však nutné mít tuto hodnotu nastavenu předem na true. To je nutné z důvodu studentova odstranění či zakomentování proměnné uvnitř jeho řešené funkce, čímž dojde při testování automaticky k nastavení řešené funkce na hodnotu true = řešeno.

Úprava Makefile souboru

Při úpravě systému jsem navíc modifikoval vzorový Makefile soubor popsaný v předchozí kapitole. Přidal jsem do tohoto souboru kopírování souboru s příponou .output se vzorovým řešením úlohy do složky se zadáním. V původním návrhu systému pro automatické zadávání úloh bylo nutné tento soubor při přípravě zkopírovat ručně. Musely se však navíc ručně přeložit a spustit vzorové zdrojové kódy tak, aby byl nejprve k dispozici soubor se vzorovým řešením pro kopii do složky. Nyní je však při zadání příkazu *make zadani* do příkazové řádky toto vyřešeno automaticky. Touto modifikací se tak ještě více vylepšilo a zjednodušilo vytváření zadání pro studenty.

4.2 Automatické hodnocení úloh

Potřeba opravovat velké množství studentských prací ovlivnila vytvoření systému pro automatické hodnocení domácích úloh. Tento systém umožňuje přidělit studentovi bodové hodnocení na základě poměru korektních testů vzorového a studentova řešení.

4.2.1 Způsob kontroly domácích úloh

V principu hodnocení dochází nejprve ke spuštění jednotlivých testů nad vzorovým řešením úlohy. Výstupem těchto testů je soubor se vzorovým řešením. Jeden soubor obsahující textový výstup základních testů a druhý textový soubor s pokročilými testy. Následuje fáze spouštění testů nad řešením studenta. Ze studentovy implementace funkcí jsou opět vytvořeny dva soubory s výsledky jednotlivých testů. Porovnáním obou souborů se vzorovým a studentským řešením obdržíme počet testů, které byly vyhodnoceny jako správné a špatné. Porovnání je prováděno textově pro každý test.

Pro porovnání vzorového řešení a řešení studenta musí být k dispozici skript *checker.php* a *checkall.sh*. Všechny skripty potřebné pro chod celého systému musejí být uloženy ve složce systém/. První skript má za úkol vytvořit soubory s výpisem jednotlivých testovacích úloh a výsledku porovnání pro jednotlivé domácí úlohy. Výsledek porovnání každého testu může nabývat pouze dvou hodnot, OK nebo failed. OK označuje, že test s daným číslem 1..n souhlasil se vzorovým řešením testu. Failed označuje chybný výsledek testu. Druhým skriptem spouštíme celou testovací fázi všech úloh studentů.

Druhý skript checkall.sh psaný v jazyce Bash má za úkol zajistit chod celého systému. Před spuštěním celého systému je nutné v tomto souboru nastavit názvy domácích úloh, které budou opravovány. Je tedy nutné nastavit názvy složek, ve kterých se nachází vzorové řešení jednotlivých úloh. Skript bude porovnávat řešení vzorových úloh s úlohami ve složkách začínajících písmenem x. Nejprve však zkompiluje a poté spustí všechny soubory v těchto složkách a vytvoří vzorová řešení všech úloh do souborů s příponou *.output.

Výsledek po použití prvního i druhého skriptu se uloží do souboru results.txt za použití přesměrování příkazem *bash checkall.sh > results.txt.* Ukázka jednoduchého výpisu po použití obou skriptů je na obrázku 4.1.

```
------------ IAL compile&check script v. 0.9 -----------
COMPILING project TEMPLATES
COMPILING students works in 'x*' directories
#xadame27:
[kmp]Making project ... OK
[kmp]Running 'advanced-test' ... OK
[kmp]Comparing results ... OK: 11 failed: 0
#test-results:xadame27:11
```

Obrázek 4.1: *Ukázka výpisu po spuštění systému pro automatické hodnocení úloh.*

Pro bezpečné spouštění jednotlivých testů máme k dispozici navíc skript *saferun.pl* psaný ve skriptovacím jazyce Perl. Tento skript má za úkol zkontrolovat, zda každý test proběhne v rozumném čase a nedojde v jeho průběhu k zacyklení. Každá testovací úloha je spouštěna jako samostatný proces, aby nedošlo k přerušení celého systému. Dojde-li uvnitř testu k zacyklení či jiné situaci, která přesahuje běžný čas výpočtu algoritmu, je tento test přerušen a je vypsáno chybové hlášení pro tento test.

4.2.2 Bodové hodnocení

Způsob hodnocení domácích úloh je závislý na počtu testů pro danou úlohu. Volba bodového hodnocení pro tyto úlohy nebyla snadná. Jedná se o úlohy, které před samotnou implementací vyžadují detailní prostudování daného problému v literatuře. Nelze tyto algoritmy implementovat pouze na základě komentářů v hlavičkách funkcí ve zdrojovém kódu. Student musí předem proniknout do daného problému a pochopit, jak algoritmus funguje.

Pro automatické bodování je v systému k dispozici skript *points.sh*. Tento soubor má na základě předem vytvořené mapovací funkce za úkol každému studentovi přidělit bodové hodnocení. Musíme určit hranici počtu testů pro jednotlivé bodové ohodnocení. Souboru points.sh stačí předat výsledek porovnání jednotlivých testů každého studenta a on se postará o vytvoření celkového počtu bodů pro jednotlivé studenty. Skriptu se za pomoci příkazu *points.sh < results.txt* předá výsledek porovnání testů vytvořený skriptem checkall.sh. Výsledek bodového hodnocení studentů lze po přesměrování uložit do zvláštního textového souboru.

Hodnocení BM, KMP a elementárního algoritmu

V testovacím souboru jsem se rozhodl vytvořit celkem 25 testů, které ověří správnou implementaci úloh. Pro algoritmus Knuth-Morris-Pratt a elementární jde o 14 testů základních a 11 rozšířených. Algoritmus Boyer-Moore je otestován za pomoci 13 testů základních a 12 rozšířených. Do složky one jsem navíc umístil všechny tři implementované vyhledávací algoritmy do jediného zdrojového souboru. V této složce naleznete testovací soubory obsahující 20 testů základních a 20 pokročilých. Mapovací funkce, která je implementována v souboru points.sh, vyžaduje zapsání mezní hodnoty počtu testů či rozsahu, za které je obdrženo bodové hodnocení. Bylo tedy nutné stanovit tyto hodnoty pro vyhledávací metody.

V polích pro jednotlivé body přidělované v předmětu Algoritmy je nutné mít uloženy meze, ze kterých je celkový výsledek vypočítán. Maximum bodů, které může student obdržet za obě správně vyřešené úlohy, je 10 bodů. Bodové hodnocení vytvořené pro nové vyhledávací domácí úlohy sbírky je znázorněno v tabulkách 4.1 a 4.2. Hodnoty zapsané v tabulce odpovídají hodnotám v hlavičce souboru points.sh. Za celkem 4 body je v tomto souboru navíc zapsáno rozmezí bodového hodnocení pro všechny tři implementované vyhledávací algoritmy ze složky one.

- **KMP algoritmus a elementární algoritmus (soubor kmp.c):**

	Počet OK testů	**Počet bodů**
Základní testy:	0 - 5	0
	6 - 8	1
	9 - 11	2
	12 -13	3
	14	4
Pokročilé testy:	11	+1
Celkem :	25	5

Tabulka 4.1: *Tabulka bodového hodnocení KMP a elementárního algoritmu.*

- **BM algoritmus (soubor bm.c):**

	Počet OK testů	**Počet bodů**
Základní testy:	0 - 5	0
	6 - 8	1
	9 - 10	2
	11 -12	3
	13	4
Pokročilé testy:	12	+1
Celkem :	25	5

Tabulka 4.2: Tabulka bodového hodnocení BM algoritmu.

Závěr

Mojí snahou při tvorbě nových typů domácích úloh pro předmět Algoritmy bylo vytvořit je tak, aby jejich implementací student dokázal lépe pochopit danou problematiku. Za použití jazyka C jsem implementoval tři nové typy domácích úloh pro vyhledávání podřetězců v řetězcích. Úkolem nebyla pouze samotná implementace úloh, ale potřeba zjistit přesně, jak daná úloha pracuje a vytvořit pro ni testovací prostředí. Toto prostředí vyžaduje vytvořit speciální funkce pro kontrolu jednotlivých průchodů prohledávaným textem. Dále bylo potřeba vytvořit kontrolní výpisy tak, aby student dokázal snadno porozumět tomuto textovému výpisu a dokázal si tak ověřit jeho správnost. Nebylo jednoduché vytvořit jednotlivé funkce tak, aby student měl plnou svobodu nad jejich tvorbou a bylo tedy potřeba dát tvorbě těchto úloh stanovený řád. Bylo nutné přinutit studenta použít speciálních funkcí uvnitř tvořených funkcí. Tyto speciální funkce by nebylo nutné použít v případech ručního testování úloh či použití dané úlohy v praxi.

Z mého pohledu bylo při tvorbě důležité a přínosné proniknout do problematiky automatické přípravy a hodnocení domácích úloh. Dle mého se mi podařilo vytvořit testovací úlohy pro vyhledávání podřetězců v řetězcích přehledně tak, aby bylo jasné, kde došlo k neshodě či chybě při vyhledávání. Podařilo se mi navíc vylepšit systém pro automatické zadávání úloh, který již neodstraňuje jednořádkové komentáře a můžeme je tak bez problému ve všech zdrojových kódech využít. Úpravou vzorového Makefile souboru jsem navíc zjednodušil přípravu zadání pro studenty a ještě více tuto činnost zautomatizoval.

Systém pro automatické hodnocení a zadávání úloh má mnoho výhod, ale také několik nevýhod. Dokáže ve velice rychlém čase opravit velké množství studentských prací bez jakéhokoliv zásahu do nich. Nedokáže však posoudit částečnou správnost řešení daného problému. Dokáže pouze kontrolovat úplně správné řešení testovacích úloh. V dalším vývoji tohoto systému po dokončení komplexní sbírky úloh by bylo vhodné zamyslet se nad kontrolou částečně správných řešení úloh. U úloh pro vyhledávání podřetězců v řetězcích je například při špatně implementované funkci pro předzpracování nemožné zkontrolovat funkci pro vyhledávání. Tato funkce může být správně implementována, ale nad konečným výsledkem se to neprojeví. Bylo by tedy vhodné například vložit vzorové řešení této funkce namísto chybného řešení studenta a s touto funkcí následně pracovat. Systém by se tedy mohl při dalším vývoji rozšířit o tento princip automatické kontroly. Musela by se však řádně promyslet efektivita opravování. Bylo by zřejmě nutné zdrojové kódy několikrát překládat, což by výrazně zpomalilo chod celého opravování.

Literatura

[1] DVORSKÝ, J.: *Algoritmy I – studijní opora* [online]. Ostrava, Fakulta elektrotechniky a informatiky VŠB, 28. února 2007. Dokument dostupný na URL: <http://homel.vsb.cz/~kal164/prednasky/ZakladyAlgoritmizace28022007.pdf>(2.3.2008).

[2] HONZÍK, J. M.: *Algoritmy – studijní opora* [online]. Brno, Fakulta informačních technologií VUT v Brně, 2007. Dokument dostupný na URL: <https://www.fit.vutbr.cz/study/courses/IAL/private/> (2.3.2008).

[3] SEDGEWICK, R.: *Algoritmy v C* /části 1 - 4, základy, datové struktury, třídění, vyhledávání. Praha : SoftPress, 2003. 688 s. ISBN 80-86497-56-9

[4] KRESLÍKOVÁ, J.; MARTÍNEK, D.: *Studijní texty Základy programování* [online]. Brno, Fakulta informačních technologií VUT v Brně, 2007. Dokumenty dostupné na URL: <https://www.fit.vutbr.cz/study/courses/IZP/private/> (2.3.2008).

[5] SVOBODA, J.: *Prohledávání textu(I-III)* [online]. Poslední modifikace: 23. ledna 2003. Dokument dostupný na URL: <http://reboot.cz/howto/software/prohledavani-textu-3/articles.html?id=292> (2.3.2008).

Seznam příloh

Příloha 1. CD/DVD obsahující:

- Implementaci domácích úloh pro vyhledávání podřetězců v řetězcích.
- Modifikovaný systém pro automatické zadávání a hodnocení domácích úloh.
- Soubor INSTALL s popisem systému pro automatické zadávání a hodnocení domácích úloh.
- Technickou zprávu bakalářské práce ve formátu PDF v souboru xadame27_bp.pdf