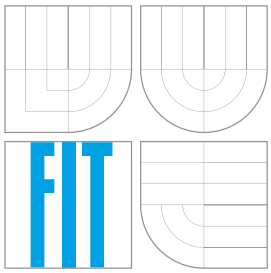
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘÍPRAVA DOMÁCÍCH ÚLOH PRO PŘEDMĚT ALGORITMY

PREPARATION OF EXERCISES FOR A SUBJECT ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PAVLA BROMOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ROMAN LUKÁŠ, Ph.D.
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/5724/2006/xbromo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Bromová Pavla**
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 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 upravte či rozšiřte stávající systém pro zadávání a hodnocení domácích úloh.
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 paměťovém médiu (disketa, CD-ROM), 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 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Slečna

Jméno a příjmení: **Pavla Bromová**
Id studenta: 84448
Bytem: Zahradní 410, 747 69 Pustá Polom
Narozena: 25. 09. 1984, Jindřichův Hradec
(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

Tato bakalářská práce se zabývá návrhem a vytvořením nové sbírky úloh pro předmět Algoritmy. Na začátku jsou vysvětleny pojmy algoritmus a algoritmická složitost, jsou popsány všeobecné vlastnosti a druhy algoritmů. Dále jsou podrobně popsány jednotlivé řadicí a vyhledávací aloritmy, které byly zpracovány v rámci sbírky úloh. Je vysvětlen princip činnosti, definována algoritmická složitost, do jaké kategorie algoritmus spadá, oblast použití, výhody a nevýhody. Další část popisuje principy systému pro automatizované zadávání a hodnocení úloh. Poslední kapitola se zabývá popisem implementace jednotlivých částí systému.

Klíčová slova

Algoritmus, řadicí algoritmy, vyhledávací algoritmy, jazyk C, domácí úlohy, automatizovaná kontrola, testování.

Abstract

This bachelor's thesis deals with a new set of exercises for a subject Algorithms. There are explained terms such as algorithm and complexity, described general properties and types of algorithms. Further, particular sort and search algorithms covered in the set of exercises are described in more details. There is explained how these algorithms work, described their complexity, category, field of use, advantages and disadvantages. Next part presents principles of a system for automated generation and rating of exercises. The implementation of particular parts of system is described in the last chapter of this paper.

Keywords

Algorithm, sort algorithms, search algorithms, language C, exercises, automated checking, testing.

Citace

Pavla Bromová: Příprava domácích úloh pro předmět Algoritmy, bakalářská práce, Brno, FIT VUT v Brně, 2007

Příprava domácích úloh pro předmět Algoritmy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením Ing. Romana Lukáše, Ph.D.

.....

Pavla Bromová

14. května 2007

Poděkování

Tímto bych chtěla poděkovat Ing. Romanu Lukášovi, Ph.D. za pomoc, kterou mi věnoval při konzultacích této práce. Také bych chtěla poděkovat Lukáši Vrábelovi za pomoc a nápady při návrhu aplikace.

Obsah

Kapitola 1

Úvod

Tématem této bakalářské práce je příprava domácích úloh pro předmět Algoritmy. V tomto předmětu se studenti seznamují mimo jiné s tvorbou již dokázaných programů a se základními principy algoritmů. Učí se rekurzivní a nerekurzivní zápisy základních algoritmů, vytvářet a analyzovat algoritmy vyhledávání a řazení. Vyhledávacími a řadicími algoritmy se také zabývá tato práce.

Významnou částí výuky jsou domácí úlohy, kde mají studenti za úkol naimplementovat několik konkrétních algoritmů a mohou tak lépe pochopit, jak fungují. Aby bylo možné tyto úlohy správně ohodnotit, je nutné jejich řešení s něčím porovnat. K porovnání slouží výstupy těchto programů. Výstupem řadicích algoritmů jsou postupné výpisy obsahu pole, u vyhledávacích algoritmů jsou to výpisy indexů, ke kterým se přistupuje. Každý vyhledávací nebo řadicí algoritmus má většinou jen jedno správné řešení, tzn. výstup.

Hlavní náplní praktické části této práce je vzorová implementace vybraných algoritmů v jazyce C, návrh na rozhraní pro práci s potřebnými datovými strukturami a vytvoření vhodného testovacího prostředí pro ověření správnosti implementace. Důraz je kladen na styl a přehlednost kódu, aby pomohl studentům ke správným programovacím návykům.

Tato část práce je zaměřena zejména na teoretický popis implementovaných algoritmů. Dále obsahuje popis systému pro automatizované zadávání a hodnocení domácích úloh a popis vlastní implementace a průběhu návrhu jednotlivých částí systému.

Struktura písemné práce

Na začátku je vysvětlen pojem algoritmus a algoritmická složitost, jsou popsány všeobecné vlastnosti algoritmů, způsoby zápisu a implementace a druhy algoritmů.

Dále jsou popsány řadicí algoritmy, nejdříve všeobecně, následuje klasifikace řadicích algoritmů podle různých kritérií. Hlavní část kapitoly tvoří popis jednotlivých algoritmů, které byly zpracovány v praktické části bakalářské práce. U každého algoritmu je vysvětlen princip činnosti, který je popřípadě znázorněný na obrázku, je definována algoritmická složitost a do jakých kategorií algoritmus spadá, jeho výhody a nevýhody, popř. jiné význačné vlastnosti.

Další kapitola popisuje podobným způsobem vyhledávací algoritmy. Nejprve všeobecně a potom už se věnuje pouze vyhledávání v seznamu. Do této kategorie patří mimo jiné lineární a binární vyhledávání, kterým se zabývá praktická část. Opět je uveden princip činnosti jednotlivých algoritmů, kdy a jak algoritmus skončí, pokud je vyhledání úspěšné a pokud je neúspěšné. Je definována časová složitost, oblast použití, výhody a nevýhody.

V další části už jsou popisovány principy systému pro zadávání a hodnocení úloh. Jsou uvedeny jednotlivé části úlohy, určené studentům a vysvětlení, jak dohromady fungují. Dále jsou popsány další součásti tvořící spolu s předchozími celý systém. Na konci jsou uvedeny použité prostředky.

Na závěr je ponechán popis implementace jednotlivých částí systému. Značná část je věnována popisu rozhraní pro práci s polem a jeho postupného vývoje, jelikož procházelo několika změnami ještě v průběhu zpracování, takže změny a také zamítnuté návrhy byly součástí implementace. Pro přehlednost jsou znovu uvedeny zpracovávané algoritmy a také je zmíněno mírné vylepšení kódu zajišťujícího správné generování zadání. Nakonec je popsáno testovací prostředí - co obsahuje testovací soubor, jak vypadají testovací funkce, jaký je rozdíl mezi standardními a pokročilými testy.

Kapitola 2

Algoritmus

Algoritmus je přesný návod či postup, kterým lze vyřešit daný typ úlohy. Pojem algoritmu se nejčastěji objevuje při programování, kdy se jím myslí teoretický princip řešení problému (oproti přesnému zápisu v konkrétním programovacím jazyce). Obecně se ale algoritmus může objevit v jakémkoli jiném vědeckém odvětví. Jako jistý druh algoritmu se může chápat i např. kuchyňský recept. V užším smyslu se slovem algoritmus rozumí pouze takové postupy, které splňují některé silnější požadavky:

2.1 Vlastnosti algoritmů

Konečnost

Každý algoritmus musí skončit v konečném počtu kroků. Tento počet kroků může být libovolně velký (podle rozsahu a hodnot vstupních údajů), ale pro každý jednotlivý vstup musí být konečný. Postupy, které tuto podmínku nesplňují, se mohou nazývat výpočetní metody. Speciálním příkladem nekonečné výpočetní metody je reaktivní proces, který průběžně reaguje s okolním prostředím. Někteří autoři však mezi algoritmy zahrnují i takovéto postupy.

Determinovanost

Každý krok algoritmu musí být jednoznačně a přesně definován; v každé situaci musí být naprosto zřejmé, co a jak se má provést, jak má provádění algoritmu pokračovat. Protože běžný jazyk obvykle neposkytuje naprostou přesnost a jednoznačnost vyjadřování, byly pro zápis algoritmů navrženy programovací jazyky, ve kterých má každý příkaz jasně definovaný význam. Vyjádření výpočetní metody v programovacím jazyce se nazývá program.

Vstup

Algoritmus obvykle pracuje s nějakými vstupy - veličinami, které jsou mu předány před započetím jeho provádění nebo v průběhu jeho činnosti. Vstupy mají definované množiny

hodnot, jichž mohou nabývat.

Výstup

Algoritmus má alespoň jeden výstup - veličinu, která je v požadovaném vztahu k zadaným vstupům, a tím tvoří odpověď na problém, který algoritmus řeší. (Algoritmus vede od zpracování hodnot k výstupu - resultativnost)

Efektivita

Obecně požadujeme, aby algoritmus byl efektivní - každá operace by měla být dostatečně jednoduchá na to, aby mohla být alespoň v principu provedena v konečném čase pouze s použitím tužky a papíru (tj. byla elementární).

Obecnost (hromadnost)

Algoritmus neřeší jeden konkrétní problém (např. ,,jak spočítat 3x7"), ale obecnou třídu obdobných problémů (např. ,,jak spočítat součin dvou celých čísel").

2.2 Algoritmická složitost

V praxi jsou předmětem zájmu hlavně takové algoritmy, které jsou v nějakém smyslu kvalitní. Takové algoritmy splňují různá kritéria, měřená např. počtem kroků potřebných pro běh algoritmu. Problematikou efektivity algoritmů, tzn. metodami, jak z několika známých algoritmů řešících konkrétní problém vybrat ten nejlepší, se zabývají odvětví informatiky nazývané algoritmická analýza a teorie složitosti.

Např. existuje jednoduchý algoritmus, který dokáže určit, zda v dané šachové pozici může hráč na tahu vynutit vítězství a zároveň dokáže určit nejlepší možný tah. Tento algoritmus se však nedá použít, protože by na svou činnost potřeboval ohromné množství času, jakkoli je toto množství konečné. Mimoto by takový algoritmus spotřeboval ohromné množství paměti, což je další praktický zřetel, který se uplatňuje při volbě algoritmu. I když průměrná počítačová paměť stále narůstá, pro některé algoritmy jí nebude nikdy dost.

Složitost algoritmu tedy vyjadřuje spotřebu výpočetních prostředků (např. paměti nebo výpočetního času, vyjádřeného jako počet provedených kroků či instrukcí) v závislosti na velikosti vstupních dat. Jako krok algoritmu se obvykle považuje činnost, kterou lze provést v konstantním časovém úseku (např. porovnání dvou čísel, přiřazení do jednoduché proměnné, atd., tedy nikoli operace s celými poli čísel, řazení čísel a další náročnější operace).

Obvykle se zapisuje jako $O(f(n))$.

Tento zápis znamená, že náročnost algoritmu je menší než $a + b * f(n)$, kde a a b jsou vhodně zvolené konstanty a n je veličina popisující velikost vstupních dat. Zanedbáváme tedy multiplikativní i aditivní konstanty, zajímá nás jen chování funkce pro velké hodnoty n.

2.3 Zápis algoritmů

Algoritmy mohou být vyjádřeny různými druhy zápisu, včetně přirozeného jazyka, pseudokódu[1], vývojových diagramů[2] a programovacího jazyka. Způsoby vyjádření algoritmů v přirozeném jazyce bývají často nejednoznačné a užívají se jen zřídka na složité nebo odborné algoritmy. Pseudokód a vývojové diagramy jsou strukturované zápisy algoritmů, které se vyhýbají mnoha nejasnostem a mnohoznačnostem běžným ve výrazech přirozeného jazyka, a přitom jsou nezávislé na konkrétním implementačním jazyce. Programovací jazyky jsou primárně určeny k vyjádření algoritmů ve tvaru, který může být zpracován počítačem, ale jsou často používány k definici nebo dokumentaci algoritmů.

2.4 Implementace

Většina algoritmů je určena k implementaci na počítači. Mohou být ale také implementovány jinými prostředky, jako v biologické neuronové síti (např. lidský mozek provádějící aritmetiku, nebo hmyz hledající potravu), v elektrickém obvodu, nebo v mechanickém zařízení.

2.5 Druhy algoritmů

Algoritmy můžeme klasifikovat různými způsoby. Mezi důležité druhy algoritmů patří:

Rekurzivní algoritmy: algoritmy, které využívají (volají) samy sebe.

Hladové algoritmy: k řešení se propracovávají po jednotlivých rozhodnutích, která, jakmile jsou jednou učiněna, už nejsou dále revidována.

Algoritmy typu rozděl a panuj: dělí problém na menší podproblémy, na něž se rekurzivně aplikují (až po triviální podproblémy, které lze vyřešit přímo), po čemž se dílčí řešení vhodným způsobem sloučí.

Algoritmy dynamického programování: pracují tak, že postupně řeší části problému od nejjednodušších po složitější s tím, že využívají výsledky již vyřešených jednodušších podproblémů. Mnoho úloh se řeší převedením na grafovou úlohu a aplikací příslušného grafového algoritmu.

Pravděpodobnostní algoritmy: provádějí některá rozhodnutí náhodně či pseudonáhodně.

Paralelní algoritmy: v případě, že máme k dispozici více počítačů, můžeme úlohu mezi ně rozdělit, což nám umožní ji vyřešit rychleji; tomuto cíli se věnují paralelní algoritmy.

[1] Pseudokód je zápis algoritmu podobný programovacímu jazyku, ale je určen pro člověka a ne ke kompilaci; mohou se v něm objevovat slovní spojení i celé věty.

[2] Vývojový diagram je grafické znázornění nějakého algoritmu nebo procesu.

Genetické algoritmy: pracují na základě napodobování biologických evolučních procesů, postupným ,,pěstováním" nejlepších řešení pomocí mutací a křížení. V genetickém programování se tento postup aplikuje přímo na algoritmy (resp. programy), které jsou zde chápány jako možná řešení daného problému.

Heuristické algoritmy: nekladou si za cíl nalézt přesné řešení, ale pouze nějaké vhodné přiblížení; používá se v situacích, kdy dostupné zdroje (např. čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné vhodné exaktní algoritmy vůbec známy).

Přitom jeden algoritmus může patřit zároveň do více skupin.

Kapitola 3

Řadicí algoritmy

Algoritmus řazení je algoritmus zajišťující seřazení daného souboru dat podle specifikovaného pořadí. Nejčastěji se řadí podle numerické velikosti čísel, případně abecedně. Řazení je velmi častá úloha, která je také částí mnoha dalších algoritmů; vývoji co možná nejefektivnějších algoritmů řazení se proto věnuje velké úsilí.

V informatice se obvykle místo slova *řazení* používá slovo *třídění*, což pochází z prehistorie výpočetní techniky, kdy se děrné štítky seřazovaly pomocí opakovaného třídění. Stejná záměna slov je v angličtině (*sort* místo *order*).

Z hlediska řazení se vstupní data chápou jako soubor dvojic klíč-hodnota, přičemž po seřazení je posloupnost klíčů monotónní, zatímco na připojené hodnoty se při řazení nebere zřetel a pouze se přesouvají vždy s odpovídajícím klíčem. Při existenci několika položek se stejným klíčem se však podle pořadí odpovídajících hodnot rozlišují stabilní a nestabilní algoritmy.

Klasifikace algoritmů

Podle různých kritérií se algoritmy řazení dají dělit do různých skupin. Dvě základní skupiny algoritmů jsou tzv. **vnitřní** a **vnější** řazení. Vnitřní řazení vyžaduje, aby všechna řazená data byla uložena v operační paměti, kde k nim má algoritmus možnost libovolně přistupovat. Pokud je dat tak velké množství, že v jednu chvíli může být v operační paměti jen nějaká část dat (a zbytek je ve vnější paměti, např. na pevném disku), je třeba použít vnější řazení.

Největší část algoritmů řazení je založena na **porovnávání** dvojic prvků. Jedná se o univerzální metodu, kterou lze seřadit libovolná data v libovolné reprezentaci (stačí příslušná relace uspořádání). Pro některé konkrétní reprezentace nějak vymezené množiny dat lze sestrojit algoritmy, které fungují na jiném principu, např. na základě reprezentace řazených čísel v poziční číselné soustavě.

Kromě samotných řazených dat také algoritmus zpravidla potřebuje nějakou dodatečnou pracovní paměť. Pokud je velikost této paměti konstantní (nezávislá na množství řazených dat), algoritmus se označuje jako **řazení na původním místě** (*angl.* in-place, *lat.* in situ),

jiné algoritmy však potřebují dodatečnou paměť, například místo o velikosti původních dat, ve kterém generují seřazený výsledek.

Vstupní data mohou obsahovat několik prvků se shodným klíčem. Podle vzájemné polohy těchto prvků před a po seřazení (kterou lze detekovat podle přidružených dat, která nejsou součástí klíče) se rozlišují tzv. **stabilní** a **nestabilní** řadicí algoritmy: stabilní algoritmus zachovává vzájemné pořadí položek se stejným klíčem, u nestabilního není vzájemné pořadí prvků se stejným klíčem zaručeno.

Podle chování na částečně seřazených souborech dat se rozlišují algoritmy **přirozené** a **nepřirozené**: přirozený algoritmus rychleji zpracuje seřazenou množinu než neseřazenou.

Dále jsou blíže popsány a vysvětleny konkrétní algoritmy, které byly zpracovány v rámci praktické části bakalářské práce.

3.1 Bubble sort

Bubble sort je implementačně jednoduchý řadicí algoritmus. Algoritmus opakovaně prochází seznam, přičemž porovnává každé dva sousedící prvky, a pokud nejsou ve správném pořadí, prohodí je. Porovnávání prvků běží do té doby, dokud není seznam seřazený. Pro praktické účely je neefektivní, využívá se hlavně pro výukové účely či v nenáročných aplikacích.

Algoritmus je univerzální (pracuje na základě porovnávání dvojic prvků), pracuje lokálně (nevyžaduje pomocnou paměť), je stabilní (prvkům se stejným klíčem nemění vzájemnou polohu), patří mezi přirozené řadicí algoritmy (částečně seřazený seznam zpracuje rychleji než neseřazený).

Průměrná i nejhorší asymptotická složitost je $O(N^2)$.

Tento algoritmus řazení je jedním z nejpomalejších, oproti jiným algoritmům se stejnou složitostí vyžaduje velké množství zápisů do paměti a tím neefektivně pracuje s cache[1] procesoru. V praxi se nepoužívá, často slouží jako algoritmus používaný pro výuku programování.

3.2 Quicksort

Quicksort je jeden z nejrychlejších známých algoritmů řazení založených na porovnávání prvků. Jeho průměrná časová složitost je pro algoritmy této skupiny nejlepší možná - $O(NlogN)$, v nejhorším případě (kterému se ale v praxi jde obvykle vyhnout) je však jeho časová náročnost $O(N^2)$. Další výhodou algoritmu je jeho jednoduchost.

Základní myšlenkou quicksortu je rozdělení řazené posloupnosti čísel na dvě přibližně stejné části (quicksort patří mezi algoritmy typu rozděl a panuj). V jedné části jsou čísla větší a ve druhé menší, než nějaká zvolená hodnota (nazývaná pivot). Pokud je tato hodnota

[1] Cache je označení pro vyrovnávací paměť používanou ve výpočetní technice. Je zařazena mezi dvě zařízení s různou rychlostí a vyrovnává tak rychlost přístupu k informacím.

zvolena dobře, jsou obě části přibližně stejně velké. Pokud budou obě části samostatně seřazeny, je seřazené i celé pole. Obě části se pak rekurzivně řadí stejným postupem.

Největším problémem celého algoritmu je volba pivota. Pokud se daří volit číslo blízké mediánu[2] řazené části pole, je algoritmus skutečně velmi rychlý. Pokud ne, je jeho paměťová i časová náročnost horší než u všech používaných řadicích algoritmů. Medián můžeme tedy vypočítat a zvolit jej za pivota. Toto je ale velmi neefektivní metoda, protože hledání mediánu běží v čase $O(N)$. Výsledkem by byl velmi pomalý algoritmus. Proto existuje velké množství způsobů, které se snaží vybrat pivota co nejbližšího mediánu. Zde je seznam pouze několika metod:

- První prvek - popřípadě kterákoli jiná fixní pozice. Velmi nevýkonná především na částečně seřazených množinách.
- Náhodný prvek - často používaná metoda. Jde dokázat, že pokud je pozice pivotu skutečně náhodná, algoritmus poběží v $O(NlogN)$. Skutečně náhodná čísla generují ale pouze hardwarové generátory, které nemusí dodávat data dostatečně rychle. V praxi mnohdy stačí použít pseudonáhodný algoritmus.
- Metoda mediánu tří - případně pěti, či libovolné jiné konstanty. Pomocí pseudonáhodného algoritmu (používají se i fixní pozice) se vybere X prvků z množiny, ze kterých se použitím některého primitivního řadicího algoritmu najde medián a ten je zvolen za pivota.

Rekurzi je možné se vyhnout použitím iterativní verze quicksortu, která nahradí rekurzivní volání použitím zásobníku. I když se tím nesníží nároky na čas a paměťový prostor samotného algoritmu, praktickou výhodou je, že se tím zrychlí běh programu, jelikož se vyhne spotřebě času a paměti při vstupu do podprogramu a výstupu z něj, k čemuž dochází během obyčejného rekurzivního volání. Nevýhodou ale je podstatně větší složitost kódu.

Přestože quicksort nemá zaručenou časovou složitost $O(NlogN)$, reálné aplikace a testy ukazují, že na pseudonáhodných datech je vůbec nejrychlejší ze všech obecných řadicích algoritmů (tedy i rychlejší než heapsort a mergesort, které jsou formálně rychlejší). Maximální časová náročnost $O(N^2)$ ho však diskvalifikuje pro použití v kritických aplikacích.

3.3 Heapsort

Heapsort je jeden z nejlepších obecných algoritmů řazení, založených na porovnávání prvků. I když je v průměru o něco pomalejší než dobře napsaný quicksort, je jeho zaručená časová náročnost $O(NlogN)$ a dokáže řadit data na původním místě (má pouze konstantní nároky na paměť).

[2]Medián je hodnota, jež dělí řadu podle velikosti seřazených výsledků na dvě stejně početné poloviny.

Základní myšlenkou tohoto kroku je využití datové struktury označované jako halda (angl. heap). Všechny prvky určené k seřazení jsou vloženy do haldy, která je uspořádá tak, že buď největší anebo nejmenší prvek lze rychle vyjmout. Kromě toho, jelikož tato operace zachovává strukturu haldy, je možné největší/nejmenší prvek vyjímat opakovaně, dokud v haldě žádný nezůstane. Tak získáme seřazené prvky.

Dodatečná paměť je potřeba pouze k uložení haldy. Heapsort využívá dvě standardní operace haldy: *vkládání* a *rušení uzlu*. Pokaždé, když zrušíme (vyjmeme) maximum, umístíme ho na poslední dosud neobsazenou pozici pole a použijeme zbylý začátek pole jako haldu, která uchovává zbývající neseřazené prvky:

Halda zbývajících neseřazených prvků	Seřazené prvky

3.4 Mergesort

Mergesort je řadicí algoritmus, jehož průměrná i nejhorší možná časová složitost je $O(NlogN)$. Mergesort je snadné naprogramovat tak, aby byl stabilní. Algoritmus je velmi dobrým příkladem programátorské metody rozděl a panuj. Je to porovnávací algoritmus. Velkou nevýhodou oproti algoritmům stejné rychlostní třídy (např. heapsort) je, že mergesort pro svou práci potřebuje navíc pole o velikosti N. Existuje sice i modifikace mergesortu, která toto pole nepotřebuje, ale její implementace je velmi složitá a pomalá.

Pro představu, mergesort pracuje následovně:

1. Rozdělí neseřazený seznam na dvě poloviny (dva podseznamy).
2. Řadí oba podseznamy rekurzivně, dokud nemáme seznamy o délce 1 - v tom případě se vrátí celý seznam (samotný prvek).
3. Spojí dva seřazené podseznamy do seřazeného seznamu.

Mergesort zahrnuje dvě hlavní myšlenky pro zlepšení doby běhu programu:

1. K seřazení menšího seznamu stačí menší počet kroků než k seřazení většího seznamu.
2. K vytvoření seřazeného seznamu ze dvou seřazených seznamů je potřeba menší počet kroků než ze dvou neseřazených seznamů.

3.5 Selection sort

Selection sort (zkráceně Selectsort) je jednoduchý řadicí algoritmus s časovou složitostí $O(N^2)$. Pro svou jednoduchou implementaci bývá často používán pro uspořádávání malých množství dat. Pro větší objem dat se používají algoritmy s nižší časovou složitostí ($O(NlogN)$) jako quicksort nebo mergesort.

Selection sort pracuje následovně:

1. Najde prvek s nejmenší hodnotou v posloupnosti dat.
2. Zamění ho s prvkem na první pozici.
3. Na první pozici se nyní nachází správný prvek, zbytek posloupnosti se uspořádá opakováním těchto kroků (počínaje druhou pozicí).

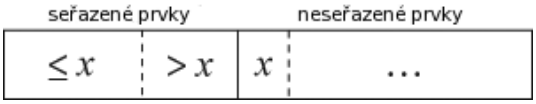
3.6 Insertion sort

Insertion sort je jednoduchý řadicí algoritmus založený na porovnávání. Na velké seznamy jsou mnohem efektivnější pokročilé algoritmy jako quicksort, heapsort nebo mergesort, ale insertion sort má své výhody:

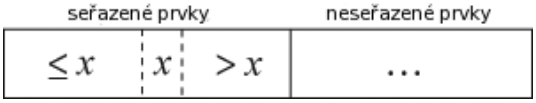
- Jednoduchý na implementaci
- Efektivní pro malé množství dat
- Efektivní na data, která už jsou z větší části seřazena
- V praxi efektivnější než většina jiných jednoduchých ($O(N^2)$) algoritmů jako selection sort nebo bubble sort: průměrná doba je $N^2/4$ a v nejlepším případě je lineární
- Stabilní (nemění původní pořadí prvků se stejným klíčem)
- Pracuje bez dodatečné paměti (na místě)

Insertion sort funguje tak, že v každé iteraci vezme jeden prvek ze vstupních dat a vloží ho na správnou pozici v už seřazeném seznamu, dokud ve vstupních datech nezbudou žádné prvky. Ze vstupu můžeme brát libovolný prvek a na jeho výběr můžeme použít téměř kterýkoli algoritmus.

Mějme pole:



Vložením x na správnou pozici dostaneme:



přičemž se všechny prvky $> x$ posunou o jednu pozici doprava.

3.7 Radix sort

Radix sort je algoritmus, který dokáže přeuspořádat čísla na základě zpracování jednotlivých cifer tak, že čísla jsou buď ve stoupajícím nebo klesajícím pořadí. Celočíselné zobrazení lze použít také k řazení dat v lexikografickém pořadí. Radix sort se dělí na dva druhy: LSD - least significant digit a MSD - most significant digit. LSD radix sort zpracovává čísla od nejpravější číslice a pokračuje směrem k nejlevější. MSD radix sort zpracovává čísla od nejlevější číslice a pokračuje směrem k nejpravější.

Princip si blíže vysvětlíme na LSD radix sortu:
Každé číslo se nejdřív ,,hodí" do jednoho kbelíku přiřazeného hodnotě nejpravější číslice každého čísla. V každém kbelíku se zachovává původní pořadí čísel tak, jak jsou postupně házena do každého kbelíku. Každé číslici přísluší jeden kbelík. Potom se postup opakuje s další sousední číslicí, dokud nezbývají žádné číslice na zpracování. Jinými slovy:

1. Vezme se nejpravější číslice každého čísla.
2. Vytvoří se skupiny čísel podle této číslice, ale přitom se zachovává původní pořadí čísel. (Proto je LSD radix sort stabilní.)
3. Postup se opakuje s každou levější číslicí.

Příklad:

1. Původní neseřazené pole:
   170, 45, 75, 90, 2, 24, 802, 66
2. Seřazením podle nejpravější číslice (jednotky) dostaneme:
   170, 90, 2, 802, 24, 45, 75, 66
3. Seřazením podle následující číslice (desítky) dostaneme:
   2, 802, 24, 45, 66, 170, 75, 90
4. Seřazením podle nejlevější číslice (stovky) dostaneme:
   2, 24, 45, 66, 75, 90, 170, 802

Je důležité si uvědomit, že každý z výše uvedených kroků vyžaduje pouze jeden průchod polem, jelikož každý prvek lze umístit do správného kbelíku, aniž by musel být porovnáván s ostatními prvky.

LSD radix sort je stabilní algoritmus, který pracuje v čase $O(nk)$, kde n je počet čísel a k je průměrný počet cifer. Jednou nevýhodou LSD radix sortu je, že nepracuje na místě - potřebuje dodatečnou paměť k uložení čísel.

Více podrobností o jednotlivých algoritmech lze najít na internetu [2].

Kapitola 4

Vyhledávací algoritmy

Vyhledávací algoritmus, všeobecně řečeno, je algoritmus, který má jako vstup nějakou úlohu, a vrací řešení k dané úloze. Většina algoritmů, které řeší nějakou úlohu, patří mezi vyhledávací algoritmy. Soubor všech možných řešení dané úlohy se nazývá vyhledávací prostor.

Existuje mnoho druhů vyhledávacích algoritmů, které mají různé využití. Zde se budeme zabývat pouze vyhledáváním v seznamu, které je také mimo jiné předmětem praktické části této práce.

Algoritmy vyhledávání v seznamu jsou asi nejzákladnějším druhem vyhledávacích algoritmů. Cílem je najít jeden prvek množiny podle nějakého klíče. Protože v informatice je to častý problém, složitost těchto algoritmů byla důkladně studována. Nejjednodušším takovým algoritmem je **lineární vyhledávání**, které jednoduše prověří postupně každý prvek seznamu. Má náročnou dobu běhu $O(N)$, kde n je počet položek v seznamu, ale lze ho použít přímo na jakýkoli nezpracovaný seznam. Dokonalejším algoritmem vyhledávání v seznamu je **binární vyhledávání** - běží v čase $O(logN)$. Pro velké soubory dat je podstatně lepší než lineární vyhledávání, ale vyžaduje, aby byl seznam před vyhledáváním seřazený.

4.1 Lineární vyhledávání

Lineární vyhledávání (také známé jako sekvenční vyhledávání) je vyhledávací algoritmus vhodný k nalezení určité hodnoty v seznamu.

Funguje na principu procházení všech prvků seznamu, dokud nenalezne hledaný prvek. Lineární vyhledávání má časovou složitost $O(N)$. V případě náhodného rozložení je průměrně potřeba $N/2$ porovnání. Nejlepší případ nastane tehdy, když se hledaná hodnota nachází na prvním místě v seznamu, v tomto případě je potřeba pouze jedno porovnání. Nejhorší případ nastane tehdy, když se hodnota v seznamu vůbec nevyskytuje (nebo je poslední položkou v seznamu), v tom případě je potřeba N porovnání.

Výhoda lineárního vyhledávání spočívá v tom, že pokud je potřeba najít jen pár prvků,

je to jednodušší než u složitějších metod jako například binární vyhledávání, které vyžadují přípravu seřazením prohledávaného seznamu.

4.1.1 Lineární vyhledávání v seřazeném poli

V případě, že prohledáváme seřazený seznam, dochází k jediné modifikaci algoritmu: prověřujeme postupně každý prvek v seznamu tak dlouho, dokud nenajdeme hledanou hodnotu, nebo dokud nenarazíme na prvek větší než je hledaná hodnota. Výhodou je, že v případě neúspěšného vyhledání nemusíme procházet seznam až do konce. V případě úspěšného vyhledání se nic nemění oproti prohledávání neseřazeného seznamu.

4.1.2 Lineární vyhledávání se zarážkou

Vyhledávání se zarážkou (tzv. rychlé sekvenční vyhledávání) používá zarážku, což je přidaná položka za poslední prvek seznamu, do níž se vloží hledaný klíč (hodnota). Vyhledání tak vždy skončí nalezením, a úspěšnost se pozná podle indexu, na němž vyhledání skončilo. Rychlost spočívá ve zkrácení booleovského výrazu.

Zarážka (*sentinel*, *guard*, *stop-point*) je často používaná technika, která umožňuje vynechat test na konec seznamu, ale nutnost rezervovat místo pro zarážku snižuje efektivní kapacitu tabulky o jednu položku.

4.1.3 Lineární vyhledávání s adaptivní rekonfigurací pole

V případě, že jsou určité položky vyhledávány častěji než ostatní, je výhodné použít vyhledávání s adaptivní rekonfiguraci pole. To je založeno na myšlence, že výhodou při vyhledávání by bylo, kdyby nejčastěji vyhledávané položky byly na začátku seznamu. To lze zajistit přeuspořádáním položek podle četnosti přístupů: po každém přístupu k položce se položka vymění se svým levým sousedem, pokud sama již není na první pozici.

Tato metoda je velmi elegantní a účinná všude tam, kde se spokojíme se sekvenčním vyhledáváním a lineární složitostí.

4.1.4 Lineární vyhledávání se zarážkou s adaptivní rekonfigurací pole

Jde o kombinaci vyhledávání s adaptivní rekonfigurací a vyhledávání se zarážkou. Používá se zarážka, vyhledání tak vždy skončí nalezením a podle indexu poznáme, zda bylo úspěšné. Pokud ano, tak vrátíme index a položku vyměníme s levým sousedem, pokud není na první pozici.

4.2 Binární vyhledávání

Binární vyhledávání pracuje nad seřazeným seznamem. Algoritmus připomíná metodu půlení intervalu pro hledání jediného kořene funkce v daném intervalu. Hlavní vlast-

ností binárního vyhledávání je jeho složitost, která je v nejhorším případě logaritmická ($O(logN)$).

Algoritmus pracuje následovně: najde medián a porovná ho s hledanou hodnotou. Pokud je medián menší než hledaná hodnota (hledaná hodnota se nachází v horní polovině seznamu), nastaví se medián jako nový začátek seznamu. Pokud je medián větší než hledaná hodnota (hledaná hodnota se nachází v dolní polovině seznamu), nastaví se medián jako nový konec seznamu. Pokud je medián rovný hledané hodnotě, vrátí se jeho index.

Iterativním opakováním této strategie se velmi rychle zúží vyhledávací prostor a najde se hledaná hodnota. Binární vyhledávání je příklad algoritmu typu rozděl a panuj.

4.2.1 Dijkstrova varianta binárního vyhledávání

E.W.Dijkstra byl významný teoretik programování druhé poloviny minulého století. Dijkstrova varianta binárního vyhledávání vychází z předpokladu, že v seznamu může být více položek se shodným klíčem. Potom je otázkou, polohu které z více položek se shodným klíčem má algoritmus vrátit. Obvyklým požadavkem je některý z krajních prvků. Tomuto požadavku odpovídá algoritmus, který nekončí tím, že najde shodu s klíčem, ale tím, že se dalším dělením dostane až na dvojici sousedních prvků, z nichž jeden je krajní z více shodných a druhý už má jinou hodnotu.

Příklad:
Mějme pole: 1,2,3,4,5,5,6,6,6,8,9,13
a hledanou hodnotu 6.
Algoritmus Dijkstrovy varianty vrátí prvek buď na 7. nebo na 9. pozici.

Některé informace byly čerpány ze studijní opory k předmětu Algoritmy [1]. Více podrobností o jednotlivých algoritmech lze najít na internetu [2].

Kapitola 5

Systém pro hodnocení úloh

Tato kapitola popisuje principy systému pro automatizované zadávání a hodnocení domácích úloh v předmětu Algoritmy.

Úlohy pro předmět Algoritmy jsou koncipované jako implementace různých datových typů a funkcí. Řešitelé dostanou k dispozici pevně definované rozhraní, do kterého mají za úkol doplnit vlastní kód tak, aby fungovalo správně podle zadání.

5.1 Jednotlivé části úlohy

Úloha, kterou student dostane, obsahuje následující soubory:

priklad.h - hlavičkový soubor, který obsahuje deklarace použitých datových struktur a funkcí, a který slouží jako spojovací soubor mezi souborem se zadáním a testovacím souborem.

priklad.c - tento soubor obsahuje samotné zadání. Jsou v něm uvedeny funkce, do kterých studenti doplňují kód, jejich popis a instrukce týkající se použití předepsaných funkcí apod.

priklad-test.c - standardní testovací soubor, který obsahuje pomocné funkce, testovací funkce, které volají funkce z řešené části a poskytují výpisy komentující výstup programu, a hlavní program, který volá testovací funkce.

Makefile - soubor pro program GNU Make.

Aby se mohlo generovat zadání a vzorové řešení jednotlivých příkladů z jednoho souboru, používá se k tomu speciálních značek, kterými se označí kód ze vzorového řešení, který má být studentům odstraněn. Tyto značky se v syntaxi jazyka C jeví jako komentáře, takže překladač jazyka C je ignoruje. Ale skripty, které ze vzorového řešení generují zadání, odstraní veškerý kód mezi dvěma takovými značkami, takže studentům se předloží hotové zadání, do kterého už můžou doplňovat vlastní kód.

Pro každou úlohu existují dva testovací soubory, jeden pro standardní a druhý pro pokročilé testování. Standardní testovací soubor se poskytuje studentům už se zadáním, aby si mohli sami monitorovat svou činnost, zkontrolovat výstupy pro své řešení a podle nich upravit svůj program. Mají také možnost si poskytnuté testy rozšířit. Soubor pro pokročilé testování (**priklad-test-advanced.c**) mají k dispozici pouze cvičící a lektoři. Rozdíl mezi standardními a pokročilými testy je v množství a kombinaci volání jednotlivých funkcí a výpisů.

Pro překlad příkladů a sestavení testů byl použit program GNU Make. Formát souboru Makefile je pro všechny příklady stejný, pouze na začátku je třeba nadefinovat název konkrétního příkladu do proměnné. Makefile obsahuje několik sekcí, které se provedou standardním voláním *make nazev_sekce*:

make, make all - kompilace příkladu a sestavení standardního testu

make priklad - kompilace a spuštění testu

make test - kompilace a sestavení pokročilého testu

make clean - vymazání dočasných souborů a sestavených programů

5.2 Další součásti systému

Jelikož domácí úlohy odevzdává velké množství studentů a manuální opravování odevzdaných úloh by bylo časově velmi náročné, nezbytnou součástí sbírky úloh je systém automatické kontroly. Základem je testovací program, jehož výstup je možné automaticky zpracovat a udělit studentovi příslušný počet bodů. Nejjednodušším řešením by bylo mít jednoznačně definovaný správný výstup, sestavit odevzdané řešení s testovacím programem a pokud by se oba výstupy shodovaly, přidělit body. Při tomto řešení by se ovšem mohlo stát, že i drobná chyba by studentovi způsobila ztrátu všech bodů. Proto jsou testovací skripty rozděleny na části, které se porovnávají samostatně, a body jsou pak přidělovány na základě počtu shodných částí.

Tímto už je proces opravování částečně automatizován. Nicméně ten, kdo by projekty opravoval, by musel překládat jedno řešení po druhém, což samozřejmě není ideální. Toto řeší skripty, které v pevně dané adresářové struktuře vybírají jeden odevzdaný úkol za druhým, kompilují je, spouštějí a porovnávají výstupy se vzorovými. Skript musí být napsán tak, aby nikdy nezhavaroval - musí ošetřit případy, kdy program z nějakého důvodu nejde sestavit nebo se zacyklí, a dále také veškerou činnost zaznamenávat pro kontrolu a případné reklamace ze strany studentů.

Aby se studenti snadno dostali k detailním informacím o svých opravených projektech, je systém rozšířen o skripty pro rozesílání hodnocení opravených úloh prostřednictvím elektronické pošty. Vedle hodnocení se případně rozesílají další dodatečné informace, ze

kterých je patrný důvod případné bodové ztráty. Tímto se v prvé řadě umožňuje studentům poučit se z vlastních chyb a také se tak předchází zbytečným reklamacím.

Poslední částí celého systému, zvyšující jeho užitečnost, je soubor skriptů pro odhalování plagiátů. Odhalování pracuje na jednoduchém principu porovnávání zdrojových souborů, kde se vynechávají mezery a jiné čistě formátovací znaky, a odstraní se komentáře. Skript umí rozpoznat i rozdíly způsobené pouze prohozením řádků a prohledává i výstupy z testovacích programů (hledání stejné chyby). Výstupem kontroly je textový soubor, na jehož základě může cvičící vybrat podezřelé řešení a rozhodnout o tom, zda se jedná o plagiáty.

Informace o principu systému byly čerpány z loňských diplomových prací[3] a [4], které se zabývaly návrhem tohoto systému.

5.3 Použité prostředky

Programovací jazyk byl pro vytvoření úloh pevně daný v zadání - jazyk C.

Návrh i samotná implementace byla přizpůsobena tomu, že veškeré kontroly a hodnocení probíhají na školním serveru *eva.fit.vutr.cz*, na kterém běží operační systém FreeBSD. Náš systém byl ale primárně vyvíjen v prostředí GNU/Linux a byly použity standardní UNIXové nástroje, které jsou na serveru k dispozici.

Kapitola 6

Implementace

Praktická část této práce se zabývá vytvořením základních částí sbírky úloh, což jsou následující soubory, které byly popsány v předchozí kapitole:

- priklad.h
- priklad.c
- priklad-test.c
- priklad-test-advanced.c
- Makefile

Všechny soubory byly vytvořeny dvakrát, jednou pro vyhledávací algoritmy a jednou pro řadicí algoritmy. Všechny algoritmy byly implementovány pro práci nad polem čísel.

Následující části se věnují detailnímu popisu jednotlivých souborů, používaných datových struktur a funkcí.

6.1 Rozhraní pro práci s polem

Největším problémem při návrhu rozhraní bylo, jakým způsobem se bude detekovat, zda student použil správný algoritmus a správně ho naimplementoval. Protože na prvním místě byla implementace řadicích algoritmů, vysvětlíme si postupný vývoj návrhu rozhraní na řadicích algoritmech. U těch je klíčový způsob procházení pole, kterým se liší jeden od druhého.

Návrh číslo 1

Původní návrh vycházel z toho, že většina řadicích algoritmů využívá pomocnou funkci *Swap()* na prohození dvou čísel. Ta měla být součástí rozhraní a kromě samotné záměny čísel měla zajišťovat výpis obsahu celého pole. Tak se mělo detekovat, která čísla jsou zaměňována, a podle jejich pozic určit, jakým způsobem student prochází pole. Výstupem

by tedy byly posloupnosti čísel v poli, které by se vypsaly po každém použití funkce *Swap()*, takže na začátku by se vypsalo původní neuspořádané pole a na konci by mělo být toto pole seřazené.

To se ale později ukázalo jako nemožné řešení z více důvodů: jednak ne všechny algoritmy pracují na principu výměny dvou čísel, takže u takových by se musel použít jiný způsob. A jednak i některé algoritmy, které využívají této metody, zásadně mění pole i jiným způsobem, např. kopírováním prvků (tak fungují většinou algoritmy, které k řazení potřebují dodatečnou paměť, např. mergesort) nebo pouze vložením na určitou pozici (např. insertion sort). Takže se musel vymyslet jiný způsob, kterým by se důkladně monitorovalo procházení polem.

Návrh číslo 2

Dalším návrhem bylo naimplementovat rozhraní v jazyce C++. Důvodem bylo, že v C++ je možné přetížit operátory tak, aby při každém jejich použití v kódu vykonávaly kromě vlastní funkce to, co potřebujeme. Hlavní myšlenkou bylo, aby se obsah pole vypsal při každém přiřazení do pole. Tak by se dala jednoduše detekovat jakákoli změna v poli, ať už pomocí funkce *Swap()*, kopírování prvků, nebo přímého vložení do pole. Tedy bylo potřeba přetížit operátor ,,=" tak, aby se při každém jeho použití (pouze) u pole vypsal obsah pole.

Při implementaci přetíženého operátoru se však narazilo na problémy, jejichž řešení by bylo komplikované a náročné, a proto se od tohoto řešení ustoupilo. Dalším důvodem pro zamítnutí návrhu bylo neměnit požadovaný implementační jazyk - jazyk C.

Návrh číslo 3 - konečný

Nakonec se jako nejlepší možnost ukázal návrh rozhraní, který si podrobně popíšeme.

Rozhraní funguje na podobném principu jako v předchozím případě s C++: zůstává základní myšlenka, aby se obsah pole vypsal při jakékoli změně v poli, nejen při použití funkce *Swap()*. Řešením je vytvořit pro práci s polem funkci, která nastaví prvek v poli na požadovanou hodnotu, a nějakým způsobem zařídit, aby studenti mohli měnit pole jedině prostřednictvím této funkce.

Toho je dosaženo tak, že pole je ,,schované" do struktury, která je před studenty ,,skrytá" v hlavičkovém souboru. Struktura obsahuje dvě položky - pole a jeho délku. K poli je tak možné přistupovat pouze pomocí této struktury, čemuž musely být přizpůsobeny všechny funkce, které pracují s polem.

Ovšem mohlo by se stát, že někteří vynalézaví studenti by prozkoumali hlavičkový soubor, který mají samozřejmě k dispozici a pro zjednodušení práce by si pole naimplementovali tak, aby s ním mohli pracovat klasickým způsobem, který už mají přece jen zažitý. To by sice bylo pro studenty ideální, ale znemožnila by se tím detekce procházení pole, netiskly by se žádné výpisy s obsahem pole a tudíž by samozřejmě neseděly se vzorovými, takže by student nezískal za úlohu žádné body. Proto bylo třeba do zadání přidat instrukce

upozorňující studenty, že strukturu nemají měnit a pro práci s ní mají použít předem naimplementované funkce, a také návod na jejich použití.

Následuje výčet naimplementovaných funkcí pro práci s polem a jejich popis v případě řadicích algoritmů:

SetValue() - nastaví prvek na indexu i na hodnotu x a vypíše obsah pole

GetValue() - vrátí hodnotu prvku na indexu i

Swap() - prohodí prvek na indexu i s prvkem na indexu j a vypíše obsah pole

CopyArray() - zkopíruje jedno pole do druhého od indexu i po index j a vypíše obsah pole

Funkce *Swap()* a *CopyArray()* je možné naimplementovat i pomocí funkcí *SetValue()* a *GetValue()*, ale jelikož nechceme studenty zdržovat něčím, co není smyslem úloh, a navíc už jen fakt, že s polem nemůžou pracovat klasickým způsobem, je určité omezení, byly předem vytvořeny i tyto dvě užitečné funkce.

U vyhledávacích algoritmů se funkce přizpůsobily způsobu kontroly vyhledávání. U vyhledávání není potřeba vypisovat obsah pole, ten se většinou nemění. (Mění se jen po vyhledání s adaptivní rekonfigurací pole, a na to stačí pole vypsat jednou.) Je třeba kontrolovat, k jakým prvkům v poli se přistupuje, v jakém pořadí, a kdy vyhledávání končí. To se provede vypsáním indexů prvků pole, které byly prověřovány. Výpisem tedy bude posloupnost indexů prvků v pořadí, v jakém k nim bylo přistupováno.

Pro tyto účely se funkce pro práci s polem mírně upravily:

SetValue() - nastaví prvek na indexu i na hodnotu x

GetValue() - vrátí hodnotu prvku na indexu i a vypíše obsah pole

Swap() - prohodí prvek na indexu i s prvkem na indexu j

Není potřeba vypisovat obsah pole ve funkci *SetValue()*, zato potřebujeme vypsat index, ke kterému se přistupuje pomocí funkce *GetValue()*. Funkce *Swap()* byla ponechána pro případ vyhledávání s adaptivní rekonfigurací pole, kdy je po úspěšném vyhledání hledaná hodnota prohozena se svým levým sousedem - posunuta o jedno místo dopředu.

6.2 Vzorové řešení

Soubor se vzorovým řešením, z kterého se bude generovat zadání pro studenty, obsahuje definice jednotlivých funkcí, které mají studenti za úkol naimplementovat.

Řadicí algoritmy

V případě řadicích algoritmů je to sedm algoritmů popsaných výše v kapitole 3:

- Bubble sort
- Quicksort (iterativní verze)
- Heapsort
- Mergesort
- Selection sort
- Insertion sort
- Radix sort

Většinu těchto algoritmů je vhodné naimplementovat tak, aby využívaly jednu nebo i více pomocných funkcí. Kód je tak přehlednější a patří to k dobrým programovacím návykům. Proto jsou pomocné funkce ponechány i ve vygenerovaném zadání spolu se samotnými „sorty", které je volají a realizují tak řazení (pomocí speciálních značek se odstraní pouze jejich těla). Studenti jsou tak nenásilně navedeni k jejich použití, ovšem pokud si chtějí algoritmus naprogramovat po svém, nemusí je vůbec použít, anebo je mohou pozměnit.

Vyhledávací algoritmy

Soubor se vzorovým řešením vyhledávacích algoritmů obsahuje následujících šest algoritmů popsaných v kapitole 4:

- Lineární vyhledávání
- Lineární vyhledávání v seřazeném poli
- Lineární vyhledávání se zarážkou
- Lineární vyhledávání s adaptivní rekonfigurací pole
- Lineární vyhledávání se zarážkou s adaptivní rekonfigurací pole
- Binární vyhledávání (Dijkstrova varianta)

U vyhledávacích algoritmů nejsou žádné pomocné funkce potřeba, protože jsou to vesměs jednoduché algoritmy, takže v této části je realizace ponechána zcela na studentech.

Vylepšení

Ve vzorovém řešení jsou použity speciální značky, které slouží k odstranění kódu ve vygenerovaném zadání, tedy kódu, který mají studenti sami doplnit. Kromě toho je v souboru použita proměnná, která indikuje, zda byla funkce implementována nebo ne. Může nabývat dvou hodnot - *false* anebo *true*. U každé funkce je implicitně nastavena na *false*, což znamená, že funkce nebyla implementována.

Ve starých úlohách bylo toto řešeno tak, že v testovací funkci se před voláním příslušné funkce tato proměnná nastavila na *true* (aby testy proběhly správně i pro vzorové řešení). Pak v každé funkci mimo kód, který má být studentům odstraněn, se nastavila na *false*. Před tento příkaz se napsaly dvě lomítka, takže pro překladač, který kompiloval vzorové řešení, se jevil jako komentář a příkaz ignoroval. Ale skript, který ze vzorového řešení odstraňuje kód mezi dvěma speciálními značkami, musel odstranit i tyto dvě lomítka, aby bylo implicitní nastavení *false*. Takže pokud student funkci z nějakého důvodu neimplementoval, ponechal příkaz na svém místě a funkce se tak vůbec netestovala. V opačném případě ho smazal, takže proměnná zůstala nastavená na *true*.

To bylo ale poněkud komplikované řešení, protože skript měl potom problémy odlišit zakomentovaný příkaz od obyčejného komentáře, takže odstraňoval všechny komentáře začínající dvěma lomítkami. A navíc musel zjišťovat začátek a konec každé funkce, aby neodstranil komentáře mimo ně, které obsahovaly zadání a instrukce k použití funkcí.

Toto se vyřešilo velice elegantně tím, že se příkaz nastavující proměnnou na *true* vepsal do kódu mezi speciální značky, takže byla proměnná správně nastavena těsně před provedením funkce. Skript potom tento příkaz odstraní i s ostatním kódem mezi značkami, takže v zadání bude proměnná nastavena implicitně na *false*, což je zajištěno na začátku každé funkce ještě před speciální značkou.

6.3 Testovací prostředí

Testovací soubor obsahuje funkce pro práci s polem, které mají studenti k dispozici, dále některé pomocné funkce pro testování, samotné testovací funkce, které volají funkce z řešené části a hlavní program, který volá testovací funkce.

Testovací funkce pracují v principu následovně: nejprve vytisknou výpisy komentující, co se vlastně testuje, dále vytisknou obsah pole, s kterým se bude pracovat a zavolají příslušnou řešenou funkci. Ta už zajistí výpis posloupností čísel v poli v případě řadicích algoritmů a posloupnost indexů u vyhledávacích algoritmů. Nakonec se vytiskne hláška, zda funkce byla implementována nebo ne. Pokud ano, tak se ještě vytiskne informace o úspěšném nebo neúspěšném průběhu funkce - jestli bylo pole správně seřazeno, nebo jestli byl prvek v poli nalezen.

V hlavním programu se nejdříve vytvoří pole potřebná pro testování a potom už se volají samotné testovací funkce. Před každou testovací funkcí se vypíše číslo testu v pevně

daném formátu. Podle tohoto formátu se řídí skript, který porovnává každou funkci zvlášť - vždy zkontroluje zvlášť každý výstup mezi dvěma formáty a porovná ho se vzorovým.

Soubory se standardními a s pokročilými testy se od sebe v principu neliší. Jediný rozdíl je v proměnných, které se předávají testovacím funkcím jako parametry. U řadicích algoritmů jsou to pole, která se liší naplněnými hodnotami, u vyhledávacích algoritmů jsou to vyhledávané hodnoty.

Řadicí algoritmy

U řadicích algoritmů se ve standardních testech pracuje s polem s náhodnými hodnotami, zatímco pokročilé testy testují funkce navíc na poli naplněném stejnými hodnotami a na poli s pouze jedním prvkem. Tak se otestuje, jestli jsou funkce správně ošetřené.

Přehled:

Standardní testy:

- Pole s náhodnými hodnotami

Pokročilé testy:

- Pole s náhodnými hodnotami
- Pole se stejnými hodnotami
- Pole s jediným prvkem

Vyhledávací algoritmy

U vyhledávacích algoritmů ve standardních testech se vyhledává hodnota, která se nachází v poli, zatímco v pokročilých testech se navíc vyhledává hodnota, která v poli není. U vyhledávání s adaptivní rekonfigurací pole se navíc ještě testuje vyhledání hodnoty, která se nachází na prvním místě v poli - v tomto případě by nemělo dojít k výměně prvků na rozdíl od všech ostatních případů, kdy se prvek nenachází na první pozici.

Přehled:

Standardní testy:

- Hodnota, která se v poli nachází

Pokročilé testy:

- Hodnota, která se v poli nachází
- Hodnota, která se v poli nenachází
- Hodnota na prvním místě v poli (pouze u vyhledávání s adaptivní rekonfigurací pole)

Kapitola 7

Závěr

Výsledkem této bakalářské práce je sbírka úloh, která slouží k výuce v předmětu Algoritmy. Sbírka má dvě části - jednu část tvoří úlohy na řadicí algoritmy, druhou část úlohy na vyhledávací algoritmy. Každá část zahrnuje vzorově naimplementované vybrané algoritmy a také testy na ověření správnosti implementace.

Sbírka úloh tvoří hlavní část systému, jehož prostřednictvím jsou studentům zadávány domácí úlohy a po jejich odevzdání automaticky hodnoceny. Tento systém byl nově vytvořen v minulém roce v rámci diplomových prací dvou studentů na základě jejich spolupráce.

Při vytváření vzorového řešení se jednalo o implementaci jednotlivých algoritmů, což proběhlo bez větších problémů. Až v průběhu implementace se objevily komplikace s tím, jak monitorovat změny v poli. Rozhraní pro práci s polem prošlo postupně několika změnami, které už byly výše popsány. Výsledné rozhraní umožňuje detekci jakékoli změny v poli. Testovací prostředí zajišťuje kromě volání řešených funkcí také přehledné výpisy komentující výstup programu, tedy průběh a výsledky testování. V závěrečné fázi bylo ještě třeba všechny soubory důkladně okomentovat, aby studenti věděli, jak s nimi pracovat a hlavně jak pracovat s polem.

Tato práce byla přínosná díky seznámení s různými algoritmy a problémy týkajících se jejich implementace. Také umožnila důkladně proniknout do problematiky zadávání úloh studentům a jejich hodnocení a uvědomit si možná omezení při implementaci úloh (v tomto případě nemožnost pracovat s polem klasickým způsobem) pro účely testování.

Úlohy tvoří celistvou sbírku příkladů, na kterých mají studenti možnost lépe pochopit princip jednotlivých algoritmů. Ovšem v případě potřeby je možné jednoduše přidat další algoritmy, ať už řadicí nebo vyhledávací, které také pracují nad polem čísel. Větší možnosti ale poskytuje vytvoření dalších sbírek na stejném nebo podobném principu, které budou obsahovat jiné druhy algoritmů nebo datových struktur, kterým se předmět věnuje.

Literatura

[1] CSc. Prof. Ing. Jan M. Honzík. Algoritmy - studijní opora. Přístupné na `http://www.fit.vutbr.cz/study/courses/IAL/public/materials`.

[2] WWW stránky. Wikipedia. `http://www.wikipedia.org`.

[3] Václav Topinka. *Příprava domácích úloh pro předmět Algoritmy [Diplomová práce].* 2006.

[4] Martin Tuček. *Systém domácích úloh pro předmět Algoritmy [Diplomová práce].* 2006.

Seznam příloh

A Datový nosič CD s kompletní implementací programu a programovým manuálem