Translation and Analysis: An Introduction to Algorithms

Petr Petrák

Bachelor Thesis
2015

Tomas Bata University in Zlín
Faculty of Humanities

Univerzita Tomáše Bati ve Zlíně
Fakulta humanitních studií
Ústav moderních jazyků a literatur
akademický rok: 2014/2015

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr Petrák**
Osobní číslo:	**H11406**
Studijní program:	**B7310 Filologie**
Studijní obor:	**Anglický jazyk pro manažerskou praxi**
Forma studia:	**prezenční**
Téma práce:	**Překlad a analýza textu: Úvod do algoritmů**

Zásady pro vypracování:

Studium odborné literatury zabývající se teorií překladu
Identifikace použitého stylu a terminologie ve zvoleném materiálu
Překlad vybraných pasáží z Úvodu do algoritmů
Analýza překladu
Vytvoření slovníku odborných pojmů

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

Baker, Mona. 2011. In Other Words: A Coursebook on Translation. New York: Routledge.
Byrne, Jody. 2010. Technical Translation: Usability Strategies for Translating Technical Documentation. Softcover ed. publication place: Springer.
Gentzler, Edwin. 1993. Contemporary Translation Theories. London: Routledge.
Knittlová, Dagmar, Bronislava Grygová, and Jitka Zehnalová. 2010. Překlad a překládání. Olomouc: Univerzita Palackého v Olomouci, Filozofická fakulta.
Mounin, Georges. 1999. Teoretické problémy překladu. Praha:Karolinum.

Vedoucí bakalářské práce: **Mgr. Petr Vinklárek**
Ústav moderních jazyků a literatur

Datum zadání bakalářské práce: **28. listopadu 2014**

Termín odevzdání bakalářské práce: **7. května 2015**

Ve Zlíně dne 23. ledna 2015

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta humanitních studií
L.S.

doc. Ing. Anežka Lengálová, Ph.D.
děkanka

PhDr. Katarína Nemčoková, Ph.D.
ředitelka ústavu

PROHLÁŠENÍ AUTORA BAKALÁŘSKÉ PRÁCE

Beru na vědomí, že

- odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby [1)];
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k nahlédnutí;
- na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3 [2)];
- podle § 60 [3)] odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 [3)] odst. 2 a 3 mohu užít své dílo – bakalářskou práci - nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům.

Prohlašuji, že

- elektronická a tištěná verze bakalářské práce jsou totožné;
- na bakalářské práci jsem pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně 3.5.2015 ..... Bělák

1) zákon č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, § 47b Zveřejňování závěrečných prací:

(1) Vysoká škola nevýdělečně zveřejňuje disertační, diplomové, bakalářské a rigorózní práce, u kterých proběhla obhajoba, včetně posudků oponentů a výsledku obhajoby prostřednictvím databáze kvalifikačních prací, kterou spravuje. Způsob zveřejnění stanoví vnitřní předpis vysoké školy.

(2) Disertační, diplomové, bakalářské a rigorózní práce odevzdané uchazečem k obhajobě musí být též nejméně pět pracovních dnů před konáním obhajoby zveřejněny k nahlížení veřejnosti v místě určeném vnitřním předpisem vysoké školy nebo není-li tak určeno, v místě pracoviště vysoké školy, kde se má konat obhajoba práce. Každý si může ze zveřejněné práce pořizovat na své náklady výpisy, opisy nebo rozmnoženiny.
(3) Platí, že odevzdáním práce autor souhlasí se zveřejněním své práce podle tohoto zákona, bez ohledu na výsledek obhajoby.

2) zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, § 35 odst. 3:
(3) Do práva autorského také nezasahuje škola nebo školské či vzdělávací zařízení, užije-li nikoli za účelem přímého nebo nepřímého hospodářského nebo obchodního prospěchu k výuce nebo k vlastní potřebě dílo vytvořené žákem nebo studentem ke splnění školních nebo studijních povinností vyplývajících z jeho právního vztahu ke škole nebo školskému či vzdělávacího zařízení (školní dílo).

3) zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, § 60 Školní dílo:
(1) Škola nebo školské či vzdělávací zařízení mají za obvyklých podmínek právo na uzavření licenční smlouvy o užití školního díla (§ 35 odst. 3). Odpírá-li autor takového díla udělit svolení bez vážného důvodu, mohou se tyto osoby domáhat nahrazení chybějícího projevu jeho vůle u soudu. Ustanovení § 35 odst. 3 zůstává nedotčeno.
(2) Není-li sjednáno jinak, může autor školního díla své dílo užít či poskytnout jinému licenci, není-li to v rozporu s oprávněnými zájmy školy nebo školského či vzdělávacího zařízení.
(3) Škola nebo školské či vzdělávací zařízení jsou oprávněny požadovat, aby jim autor školního díla z výdělku jím dosaženého v souvislosti s užitím díla či poskytnutím licence podle odstavce 2 přiměřeně přispěl na úhradu nákladů, které na vytvoření díla vynaložily, a to podle okolností až do jejich skutečné výše; přitom se přihlédne k výši výdělku dosaženého školou nebo školským či vzdělávacím zařízením z užití školního díla podle odstavce 1.

ABSTRAKT

Tato bakalářská práce se zaměřuje na překlad odborného textu z oblasti výpočetních technologií - konkrétně na odbornou publikaci Introduction to Algorithms. Teoretická část se zabývá všeobecným popisem pojmu překlad, zmiňuje jednotlivé druhy ekvivalence a představuje rysy vědeckého textu.Praktická část obsahuje přeložený úryvek zdrojového textu včetně analýzy pro snadnější orientaci v jeho charakteristice. Ta je také doplněna o slovník pojmů nacházející se společně s přílohou za přeloženým textem. Práce si vytyčuje za cíl analyzovat text, přeložit ho a představit výčet použitých odborných výrazů.

Klíčová slova: překlad, ekvivalence, vědecký text, analýza textu, terminologie

ABSTRACT

This bachelor thesis focuses on a translation of a scientific text from the field of information technology, in particular the publication Introduction to Algorithms. The theoretical part deals with a general description of the term translation, mentions individual types of equivalence and introduces features of scientific style. The practical part contains a translated excerpt of the source text including its analysis for better interpretation of its characteristics. In addition, a terminological glossary is included in the appendix at the end of the translated text. The aim of the thesis is to analyse the text, translate it and introduce a list of technical terms used in the translation.

Keywords: translation, equivalence, scientific text, text analysis, terminology

ACKNOWLEDGEMENTS

I would like to thank Mgr. Petr Vinklárek for his guidance, valuable advice and persistent support during the writing of bachelor thesis. I also want to appreciate Adéla Benová for a great support and thought-provoking observations and Jakub Karel for developing my knowledge regarding the computer science.

CONTENTS

INTRODUCTION

Many scientific texts, guides and instructions regarding computer science have been written in English and only some of them were chosen to be translated. In my opinion, it seems a pity and at the same time thought provoking. However, such lack of translated publications could be justified by difficulties the translating process presents and by the needs of the target audience who should be capable of speaking, writing and reading in English well enough to comprehend the original meaning and apply it in practice.

Nevertheless, reading this text in the source language revealed that meaning could be understood and applied incorrectly. Even though authors who write such texts use a clear style and aim to provide exact information, at times an interpretation in the target language may bring additional questions regarding exactness and credibility of the transferred information, findings or instructions.

The bachelor thesis therefore consists of a theoretical part describing the nature of translation, the process of translating, known types of equivalence for translating and a chapter which summarizes information about scientific style. These findings are subsequently applied in the practical part, which contains an excerpt of the source text translated into Czech language. The terminological glossary will help to sum up all important terms appearing in the text. Readers will have a chance to compare the effort of transferring the meaning and evaluate the quality of the result to see whether it corresponds with their expectations with regard to features of a scientific text.

The result of the translation should answer the question to what extent does the translation of such voluminous and complex publications preserve the intended meaning of the authors and the usefulness of the translation in itself. The analysis also provides information about its structure and requirements expected from the reader in order to say whether translating of scientific texts in a wider extent is or is not a meaningful activity.

I. THEORY

1 NATURE OF TRANSLATION

1.1 Term translation

The notion of translation can be described as the process of transforming a source text into target text or as the product of a translator´s effort (Hatim and Munday 2004, 4–5). Translation is attempting to produce a true transformation of original text in order to provide the access to the resource of information, thoughts or feelings inserted by the author of original text. Translation itself helps readers to understand many texts in foreign languages, who are incapable of reading and understanding the text in original language partially or at all.

According to Dagmar Knittlová (2010, 8), translation deals with essential issues of equivalence in the lexical and the grammatical level. Therefore, translation accesses gently to semantic and pragmatic aspects of a text in order to bring the best result in a target language and allow to understand a text with fewer obstacles.

1.2 Process of translating

The process of translating a text represents several phases dealing with identification of all aspects, the aim and leading to finished work. Christiane Nord (2005, 34) distinguishes two basic and frequently used models, i.e. two phase model or three phase model. This distinction means that two phase model presents idea of the focus on analysis whereas translator is collecting the useful information in order to be able interpret it through the verbalization. In this case model works as code switching operation.

However, three phase model also contains the transferring component essential for delivering the meaning of source text to the intention of target text. The rest of the model stays the same and it does not modify previously introduced model any more (Nord 2005, 34). It is possible to define process of translating as a summary of logically following steps, where translator considers all the aspects (from macro- and micro- view) and creates a text with proper equivalents (with respect to source text focus). Knittlová suggests determining a proper processing of text for translation. This means to set a logical order of analysis, i.e. from ground to core with aim on genre that is determined by further elements as register, field, the level of formality and the form of message (2010, 27–36).

1.3 Approach to translation

Translators can select from two ways how to approach to the translating. The first way, how to access to the translation, describes approach based on collecting feelings from several translated sentences, the second way offers more analytical access to text because it deals with observing the register, tone of text, difficulty of vocabulary and seems more logical if it is used in technical translations. The approach is also determined by type of a translation, with which translator has to work.

1.3.1 Types of translation

The people, who are translating texts on regular basis, have to work with several types of translations in order to assess which one is the suitable one for their work. Depending on various aspects, translators have to choose between three basic types of translations: intralingual, interlingual and intersemiotic (Hatim and Munday 2004, 4–5).

Interlinear translation provides a source information transferred in the source language regardless grammatical rules, even though it transfers grammatical unit from the source language to corresponding target language. The translation fulfils its function in descriptive linguistics and helps to understand specific information. It may be thus comprehensible if there is close relation between two languages.

But in case of literal translation translators tend to convert every lexical unit without contextual engagement however it is adhered to the grammar system of the target language. Text may become a true rewritten copy in the target language but with feel that text is not written by native speaker (Knittlová et al. 2010, 17).

Free translation lacks aesthetic qualities and does not focus significantly on connotative parts of meaning. It also ignores register or stylistic features and gives reader an inaccurate translation possibly with missing details. That is why it appears for instance in amateur translation. (Knittlová et al. 2010, 17)

Communicative way of translating, as the last one, factors a pragmatic aspect and adheres to the meaning of sayings, proverbs, headlines, etc. Peter Newmark (1988, 45–47) adds several more types of translation, namely literal, faithful, semantic, adaptation, idiomatic and communicative translation, based on source or the target text.

Hence the literal translation is searching for nearest equivalents, while a true translation tries to reproduce the contextual meaning together with preserving of grammar (more dogmatically) and semantic one focuses on aesthetic value of source language and admitting creativity. In addition it must be distinguished between adaptation, idiomatic

translation and communicative translation to evaluate the most suitable option for translating process.

Adaptation, for instance, enables to rewrite plot and convert source language culture to target language culture, while idiomatic translation allows to reproduce the meaning, but with deliberate changing of used idioms for collocations and idioms in not existing in original. Communicative translation is attempting to sustain the contextual meaning and selects comprehensible language for reader.

2 EQUIVALENCE

This chapter provides particular distinction of equivalence used between the source and target text. The main goal of translator is to identify and select the best option for translating the text. Mona Baker (2011) poses a distinction of equivalence, which is possible to encounter in the process of translating. She describes equivalence on word level, above word level, grammatical, textual and pragmatic. The other theorists, contributing to notion equivalence, will be included in following subchapters as well.

2.1 Equivalence on word level

Baker (2011, 10) describes this type of equivalence as operation with basic unit creating the meaning by itself, i.e. word. Together with morpheme it underlies the minimal unit of the meaning and the main goal of translator is to translate the text with respect to cultural differences, the importance of paraphrase, relevance for each language and unequal relationship between word and meaning. For instance term *exponentiation* can be paraphrased as increasing the value by multiplying itself according to value of exponent. This refers to fact that word consists of two morphemes - *exponent*, a mathematical notation, and *iation* signalling a process. A word for word translation brings difficulties caused by morphemes being able to change their properties, for instance class, tense, gender, etc., or words whose counterpart does not correspond in amount of words.

Therefore, in order to of making the flawless translation, it is vital to pay attention to semantic fields or lexical sets covering words or expressions and hence for translator is essential to know the value of each word in the source language and evaluate the size of contrast between languages and their elements.

2.2 Equivalence above word level

Collocations and idioms become the point of discussion in this part. The scope applies to more complex units, i.e. words connected with each other giving a clause or sentence and respecting the grammatical rules.

In case of collocations they represent a reflection of the material, social or moral environment in which they appear. ‘This term also connects with term range meaning the extent of ability of the word to be connected with others. It depends mainly on level of specificity and number of senses which it can bring’. Translators can observe several issues related to collocation translating. Firstly it can be connected with the attempt to copy a source text collocation to target one while trying to express the same meaning. Sometimes

it can happen that translator may misinterpret the meaning of the source language collocation – it comes from correspondence in form between collocations of the source and target language (2011, 54–60).

On the other hand idioms do not offer so many possibilities for variation and decrypting of the meaning from individual parts of the collocation is not executable every time. Fixed expressions and proverbs also add more transparent meaning than idioms. According to Baker (2011, 68–70), it is vital to be aware of 'ability to recognize and interpret an idiom correctly and the difficulties involved in rendering the various aspects of meaning that an idiom or a fixed expression conveys into the target language'. Translators may find decision whether they are dealing with an idiomatic expression or not as first obstacle. All it can be caused because of not following the grammar rules of the language or inducing to translate not the idiom not literally (simile). Therefore it is possible to state that the more difficult idiom to be translated, the less sense it can make to readers. It happens that translator can easily misinterpret the meaning of idiom just because of literal translation seems reasonable and it sounds quite well.

In second case it happens that 'source language idiom may have really close counterpart in the target language on the surface however a different meaning'. So it is vital to be aware of idiom environment surrounding any expression whose meaning is hard to access and collocation patterns (2011, 68–70).

2.3 Grammatical equivalence

Baker (2011, 92–94) poses grammar as 'a set of rules for combining words and phrases in a language and makes notions like gender, number or reference more explicit'. It works with two fundamental components, i.e. morphology covering the structure of words, and syntax covering the grammatical structure of groups, clauses and sentences including their position and restrictions regarding the interchangeability while every selecting of choice, whether grammatical or lexical, brings more or less limitations, i.e. grammatical ones are obligatory, on the other hand lexical ones are optional. It is considered for very difficult to make changes in the grammatical rules compared to lexical level enabling the acceptance of new words, phrases, expressions, collocations, etc.

A grammatical system of each language differs from each grammatical system has unequally complex categories. For instance, suffix *–s* or *–es* indicates a morphological representation of the number in English, but on the other hand Czech operates with more complicated system of plural where the suffix varies according to gender, animateness and

it distinguishes between one, two and more than two, see example: *one variable, two variables and five variables* vs. *jedna proměnná, dvě proměnné a pět proměnných* (2011, 95–96).

English works with simple distinction. It means that three possible persons in two modes – singular or plural - the same distinction applies to Czech, too. Slavic languages, for instance, add pronominal adjectives to refer to the subject with proper referent, compared to English that works with four persons (‘one’). In addition, all the languages use addressing or deference in various ways, e.g. Czech makes distinction based on choice second person singular (*Ty* - informal / *Vy* - formal).

Tense usually serves to indicate three basic states: past, present and future. This distinction further spreads more or less according to particular language. Translator has to pay attention to correct conversion of tense from one language to another one in order not to corrupt a timeline of events in the text. In case of aspect, it is needed to be aware of temporal distribution of an event and verbal aspect (completion or non-completion of the action). Particular case – academic writing – shows using present perfect or past simple if the translator wants to refer other author´s work and attempts to bring consistency in the target text (2011, 108–111).

In case of voice it is distinguished between active and passive. In selected languages it happens that using passive voice considers for obligatory in particular contexts, for instance agent is a third person acting upon a first or second person in American Indian language (2011, 112).

Most languages can produce sentences without agent or with dummy subject in order to avoid mentioning of the origin of the action. A frequency of using the passive depends on every language and it is given just by pure convention. For instance, technical writing in English uses more often passive voice than German, Russian or French. This concept serves to bring an impression of objectivity however, for instance, intentional using passive voice in Japanese means conveying the unfortunate new(2011, 113–117).

Mark Herman (Wright and Wright 1993, 13) also adds that translated text may require recasting of sentences in order to support clarity. It could mean that, for instance, the Czech language, as highly inflected language, requires to rearrange of elements in the clause if translator attempts to create a coherent text. Later it will be shown in the Czech text that more repetition of terms and denotations supported the clarity, whereas English text reduced fairly the repetition of these elements.

2.4 Textual equivalence

This type of equivalence deals with connecting the meaning based on lexical or grammatical links in the text. Jan Firbas (Hajičová 1996, 226) presupposes that foundation of clause is created by theme used by other elements in order to express a new information (topic of communication), transition carrying elements with connecting attribute and the rheme completing a sentence (giving information about speaker´s attitude).

Halliday and Hasan (1976, 4) describe textual equivalence as five fundamental devices creating a cohesive text: reference (personal, demonstrative and comparative), substitution (nominal, verbal and clausal), ellipsis (nominal, verbal and clausal), conjunction (additive, adversative, causal and temporal) and lexical cohesion (proper selection of vocabulary – reiteration and collocation). A cohesive text works ideally with all these devices in order to produce a logical and aesthetically acceptable flow of information or ideas in the text. Translators and readers may see later that cohesion does not stand for coherence and cohesion represents the image or surface of the text visible for readership.

Knittlová also adds distinction of three types of equivalence for lexical meaning. These are full, partial and null equivalence.

In case of absolute equivalence, counterparts create lexical units corresponding with each other as to meaning in the context, stylistics (register included) and vocabulary (anthropocentric approach). In addition, its feature is symmetry (e.g. title consists of one word or two words, etc.) and motivation. Words, which are created by process of making the mono semanteme by the context, can be considered for equivalence, too. If adjectives are inspected closer, there prefers to mark objective properties and speaking about adverbs, the equivalent counterpart comes from lingual environment.

Partial equivalence seems based on evidence that languages have got different origin, culture, geography, etc. It means that it can different in case of formality, meaning (denotation), meaning (connotation) or pragmatic. Speaking of formality, English works with more explicit terms and analytical, polyword expressions than Czech. Polyword expressions represent a negative attitude towards the specific object. Constituting noun can be removed denotative components but negative connotation can stay. Czech counterparts may be more complex than English ones (three words may equal to one word and vice versa). Rational and emotional evaluation can be included in the translated Czech term while in original one is more described (Knittlová et al. 2010, 39–42).

Last type describes null equivalence where translator has to solve out absence of counterpart by borrowing the term from source language or give it to Czech nature: it means generalization or periphrastic processing or so called functional analogy. This might be a typical situation for terminology mostly from technology, science or quickly developing industrial branches.

2.5 Pragmatic equivalence

This is the last part dealing with equivalence from viewpoint of the comprehensibility of the text. Like in previous paragraph, coherence and implicature cover certain network of relations, however the main scope focuses on perception of surrounding world and reader´s expectations.

High rate of coherence appears in texts with dense informative value and therefore the aesthetic function may present less significant component than coherence. The translator has to approach text carefully and works variety of sources, differ between referential or relational linking, and also with readers´ expectations.

Albrech Neubert and Gregory M. Shreve (1992, 94–95) describes the notion of coherence as the guide for reader, who is led by logical structure of the text. This explains why authors such publications thoroughly keep structure of scientific works and so makes sense for readers. Peter D. Fawcett (1997, 3:124–125) adds and claims that authors of any texts have to work with readers´ expectations to set clear presuppositions which serve as contextual triggers. The problem may be seen in the case if the target audience do not recognize the intended meaning and the translator has to adapt it to their knowledge or previous experience. Compare *Here, we use ∞ as the sentinel value.* and Czech counterpart *Použijeme ∞ jako sentinel value.*, where the original term was kept, however Czech readers may infer the meaning according to function of the value securing the certain limit, and therefore in Czech coined as *mezní hodnota*.

3 SCIENTIFIC STYLE

This chapter will focus on identifying of the text that belongs to scientific style. Firstly, however, generalization will help to understand the whole range of text variety. Every single text have own specific features and each type brings a different function. It means, for instance, that can be informative, expressive, operative or in form of audio-medial text (Munday 2001, 75).

3.1 Features of scientific style

Scientific style of writing is being characterized as brief, clear, useful (Ad rem) and specific (using of the specialized terms, collocations, structuring of findings, making of conclusions and further motivation of reader to think about the topic more critically). This style deals with description of research or function of the procedure, verifying of the facts and, inferring the new findings and creating of the conclusions summarizing that research. The excerpt for translation, even though represent a textbook or guide, contains these features because it meets reader with description of algorithms, their function and makes a proper conclusion at the end of the chapter.

3.2 Characteristics of text & function

Knittlová (2010, 148–168) claims that scientific text mainly works with the informative function enriched by thoughtful order and clear expressive devices. Therefore texts present information without redundant and distractive elements, e.g. developing adjectives, with often inclination to convey the content in passive voice. Text itself attempts to deliver as much new and important information as possible and therefore ellipsis become an usual device, however text does not have to be got rid of metaphors providing encoding the meaning. But in this particular case, excerpt from publication belongs to academic style of writing and more complex syntax prevails upon simple statements of facts, especially in explanations or footnotes.

Function of each work is determined by the purpose which means some of them serve as tutorials or further motivation to learn more about the topic. Readers also appreciate the simplicity, clear structure of text (correct indentation, alignment, font type, etc.) and supporting illustrations (Byrne 2010, 63–76).

3.3 Terminology

Scientific or technical texts work frequently with terms expressing for example activity, state, process, etc. Terms enrich the text and condensates the explanation. Their specificity requires the previous knowledge or experience with the topic in order to understand the whole text.

Using of original term instead of Czech equivalent has become common matter for readers such publications and therefore terms, for instance names of algorithms (*merge sort, insertion sort, selection sort,* etc.) appears frequently used in the English language. Knittlová (2000, 152) adds that problem may appear in question of semantic condensing at English terms causing difficulties while translating into Czech and extending the target text.

Newmark (1988, 152–154) offers further distinction between technical and descriptive terms and set varieties of styles typical for each text: academic, professional and popular. This particular excerpt uses an academic style with descriptive terms such as *exponentiation, caller, ceiling* or *floor* which describes every function, procedure or feature in expressive way. These terms may be often related to the biology or nature, in order to express their shape or function, see notion ordered or unordered *tree* describing the data structure having so called *nodes* containing values and are differentiated as *parent* or *children* according to relationship between *nodes*[1].

Phillip Rubens (2001, 56–57) recommends that text should work with a minimum synonyms in order not to change the concept for idea, compare choice of Czech between *řada* and *posloupnost* for English counterpart of word *sequence* - picture of row with numbers may be described by both expressions but the second one is usually used in math terminology.

[1] See https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/trees/lecture.html

II. ANALYSIS

4 ANALYSIS OF TRANSLATION

4.1 Methodology of analysis

Analysis of the source text and translation was described according to Nord (2005, 43 – 139) and was found as the appropriate solution how to summarize all the known facts. Her analysis covers two basic aspects: extratextual factors comprising of the analysis a sender, intention, recipient, medium/channel, place of communication, time aspects, motive and text function, and intertextual factors comprising of the analysis a subject matter, content, presuppositions, text composition, non-verbal elements, lexis, sentence structure and suprasegmental features. This was adopted in order to sum up details especially about original text and apply these findings in the translator´s attempt. Analysis also contains some findings from translated excerpt, too.

4.2 Translation analysis

Initially, it seems convenient to be acquainted with text on general level. This translated excerpt demonstrates only a small part of publication called *Introduction to Algorithms* which aims to provide exhaustive description of many algorithms (however not all of them) with their application. Publication was written for learning purpose in the United States by a group of highly educated scientists, namely Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. The whole publication, published in 2009 as third edition, can be borrowed in most university libraries. Analysis of the text constitutes on identification of variety of details regarding the text. For this purpose the translator has chosen colourful text with specific word stock, collocations and terms. It also deals with issues during translating and describes the translator´s solutions how to overcome them.

For purpose of translation was chosen a part of second chapter where is already expected to know at least basics of programming in order to understand all used concepts, ideas and findings.

4.3 Analysis of extratextual factors

The original text will be screened in order to inspect closer extratextual factors:

4.3.1 Sender

A group of authors fulfils the function of senders who work as computer scientists in the field with long experience. They produce a new text combined with interpretation of

theoretical findings from mathematics and computer science. Translated text sustains this feature and adheres to true transfer of the meaning.

4.3.2 Intention

The intention of publication and authors is to meet readers with very dense overview of algorithms, understand their structure and application in educative way. This type of text, by definition, therefore presents straightforward intention to learn about algorithms.

4.3.3 Recipient

Authors define as recipient students of computer science but textbook can offer fairly useful knowledge base for those who are already experienced in programming and computer security. It was assumed that all recipients might have the same access to preliminary knowledge helping to understand the topic, nevertheless some notions and ideas presented in the publication have not been fully adopted.

4.3.4 Medium

Original excerpt provided details regarding the text formatting if it is needed to inspect medium. This meant to sustain at least some elements in order not to corrupt or change difference in meaning and cause factual error, compare $n_2 + 1$ and $n2 + 1$, which is not the same (applied to indices and signs). Translated sample adhered to the original text in all aspects to provide true copy (indices, signs, bold, italics, font of footnotes). This textbook could be purchased or borrowed as PDF file or hardcover.

4.3.5 Place of communication

Place of communication is represented by universities or institutions where students, graduates or experienced programmers have access to printed or electronic version of publication. Text was produced in Massachusetts Institute of Technology (MIT) in the United States and may be accepted by any group of students in the world capable of reading and understanding the English text (preliminary knowledge of mathematics and programming was mentioned). It could be true that reception by original audience, i.e. English native speakers from the United States and other English speaking countries, would occur less obstacles during the reading and understanding of the book while on the hand other audience may struggle from time to time with complicated word order. Translating process respected this setting about word order and transferred ideas as much non-biased as possible.

4.3.6 Time aspects

Given by nature of computer science and development of this area, time aspects correspond with exact descriptions in the text where authors used mostly simple tenses to express clearly the order of actions. No shift or lag can be recorded in the vocabulary – the content and terminology in the text has not become obsolete and it is still relevant for contemporary study of algorithms. In addition, text refers to knowledge explored several decades ago, however being updated and still relevant in this time.

The only problem of exact expressing of the action (its verbal aspect) can be found in a distinction of the proper Czech equivalent for verb *terminate* (particularly p. 19 in the source text) where was possible to choose between *končí* or *skončí*. The first choice was the core of examination because it stresses what is happening while application of the algorithm finishes.

4.3.7 Motive

Authors had to find a reason or motive for creating such publication. This is an explanation why to bother with algorithms in order to create more complex applications, for instance effective processing of big data, encryption or securing of sensitive data, etc. Textbook is primarily designed as educational tool and that is why readers should access to it repeatedly to understand the gist of each chapter.

4.3.8 Text function

The last part of extra textual factors examines text function. Publication helps understand target audience a topic and serves as the instrumental guide. Informative function is therefore fulfilled by precise structure, helpful explanations, demonstrations of *pseudocode*, etc. Each chapter contains a series of exercises to verify gained knowledge during learning.

4.4 Analysis of intratextual factors

This part of analysis offers a closer look into what is sender trying to say and what are means of conveying the information:

4.4.1 Subject matter

Firstly, subject matter defines the level of coherence which appears on high level due to one dominant subject. It determines the terminology (mathematics and computer science) and logical order of well-arranged ideas.

4.4.2 Content

Content provides a rich lexical structures using colourful linking devices such as a variety of conjunctions, e.g. *hence, thus, which*, etc., references and repetition of denotations. As original text, the translated excerpt also uses these linking devices, e.g. *a proto, protože, neboť, který,* etc.

4.4.3 Presupposition

Authors establish as presupposition in the introduction - it is necessary to have done basics of computer programming in order to understand description and application of algorithms. However the rate of explicitness appears on high level and helps less experienced readers to understand sophisticated concepts. It means that text contains summary in form of introducing sentences at the beginning of paragraph or chapter. English text works more with ellipsis and avoids repeating of some terms compared to Czech text which repeats some collocations more often to sustain coherent text.

4.4.4 Text composition

Coherence and cohesion of text is being supported by clear text composition which builds up on chapters and paragraphs, all of them separated by head title and dividing line. Authors provide additional explanation in footnotes below the text and embedded information as anaphora in the sentence. Sometimes it seemed difficult to understand fully the intention of authors and transferring the meaning of these sentences with embedded information was problematic.

4.4.5 Non-verbal elements

The publication also works with amount of non-verbal elements in form of illustrations, excerpts of *pseudocode*, tables with development of algorithm, special symbols substituting of the whole word (e.g. denotation for *theta* notion squared *n*). Formulas are also described in mathematical notion which helps understand the meaning even without words. Illustrations, for instance, support explanative function in order to understand the application of algorithms more effectively. Text itself was originally processed in LaTeX[2], which offers more aesthetical and sharper font, however the

[2] See homepage of this document markup language for detailed description, http://latex-project.org/intro.html.

translated excerpt used Microsoft Word instead – it means for instance bigger file size of the output PDF file.

4.4.6 Lexis

Lexis, used in the excerpt, creates a substantial part of analysis. Firstly, Nord (2005, 125) mentions a degree of originality represented by using of new words invented by authors. In this case authors take over all terms used in mathematics and computer science but also define own ones, for instance *pseudocode*, *running time*, *size of input*, etc. (at least in the excerpt). Secondly, text is missing any influence of regional or social dialects and prefers to deliver information in maximally objective and academic style. Authors adhere to address in first person of plural however translated excerpt contains a passage with more personal addressing (second person of singular), particularly in *Divide-and-conquer* algorithm (*Rozděl a panuj*). Many sources dealing with computer programming written in Czech prefer to use original term or adapt more personal addressing in the name of algorithm. Explanation regarding this algorithm in the excerpt therefore uses personal addressing, too.

Terminology of the text works mainly with notions from computer science (*RAM, loop-counter*) and mathematics (*recursion, exponentiation*). It can be observed some relation to the nature, see translation of *merge sort* - *třídění slévání*m instead of *třídění spojováním*. Similar relation appears in programming languages like C, C++, Pascal, etc. Czech equivalents of the English terminology regarding these languages are closely related to liquids, see *underflow, overflow, stream* and Czech equivalents *podtéct, přetéct, proud*. Text is charged with collocations and specific terms; some of them may seem untranslatable or simply have no Czech equivalence, for instance *sentinel card* and *sentinel value*. It was necessary to coin the new meaning for these terms from the context - *mezní karta* and *mezní hodnota*. Repetition of some conjunctions like *which, that, and, but* appears quite frequent however it helps to create a cohesive text. This feature also appears in the translation in order to create cohesive text, too. The padding such as *typically, on average,* etc. opens up mostly new sentence and its using occurs in the text frequently.

In addition, it has been observed that original excerpt contains expressions or collocations which should not appear in such textbooks or guides, however these expression are attempting to make the text obviously less formal. Recall these examples in the source text: *tedious details* (p. 23), *handful of problems* (p. 24), *messy formula* (p. 25). The translated excerpt transferred these expressions without change.

Some terminology opened a question of choice the correct meaning because it was possible to match more than one possible meaning to original term. Distinction between *třídění* and *řazení* as equivalent for English term *sort* appeared as first translating issue but Tomáš Valla (Valla 2007), a researcher at the Computer Science Institute of Charles University, provided a reasonable explanation regarding the choice of correct term and concluded that term *seřazení* is applied to sort for example numbers or data, on the other hand *třídění* is applied to classify numbers or data. Another example, *statement cost*, corresponds to *hodnota výroku* in logic, however it appeared more suitable for purpose of translation to adapt the meaning of original term to *cena výrazu* describing better activity of algorithm[3].

4.4.7 Sentence structure

As mentioned before, text presents information, descriptions and findings thanks to choice of proper style and features and there is no surprise that sentence structure holds the same level. It contains complex sentences with embedded information and includes some syntactic features, namely aposiopesis, perhaps to keep the reader´s attention and being not monotonous.

4.4.8 Suprasegmental features

The last part of intratextual factors deals with suprasegmental features which indicated application of special formatting like bold, italics, using of quotation marks, dashes, various parentheses, affirmative words, enumerations and working with theme and rheme. In case of text it is possible to find all the named features because of various reasons, for instance all elements or words in italics or bold represents emphasis, denoting of structure, procedure or terms (e.g. *A.length*, ***linear search***, ***short circuiting***, etc.). In addition, the intonation does not play a role in both texts because the main goal is to give unambiguously and objectively a guidance in learning of algorithms. Stress might be applied in form italics, bold or combination of both to point out the importance of highlighted element or term.

[3] Executing of the step in the algorithm takes a time which costs time. Therefore *cena* expresses better a quantity measured in performance the algorithm.

5 TRANSLATION OF AN INTRODUCTION TO ALGORITHMS

2 Na úvod

Tato kapitola vás seznámí se strukturou, která bude použita v celé učebnici s cílem zamyslet se nad vytvářením a analyzováním algoritmů. Kapitola je nezávislá, ale zahrnuje několik odkazů na látku probíranou v kapitole 3 a 4 (obsahuje též několik shrnutí, jejichž řešení je zobrazeno v příloze A).

Začneme s analýzou *insertion sort* (seřazení vkládáním), abychom vyřešili problém řazení uvedený v kapitole 1. Zároveň definujeme pojem pseudokód, který by vám měl být znám, pokud již máte počítačové programování za sebou a použijeme ho, abychom vám ukázali, jak budeme konkretizovat naše algoritmy. Po upřesnění algoritmu *insertion sort* se budeme věnovat dokazování správnosti řazení a analyzujeme jeho dobu běhu. Tato analýza představuje notace zaměřující se na to, jak se doba běhu zvyšuje s počtem objektů určených k seřazení. Po kapitole o *insertion sort* představíme přístup *divide-and-conquer* (rozděl a panuj) pro tvorbu algoritmů a použijeme ho na rozvoj algoritmu zvaného *merge sort* (seřazení sléváním). V poslední části se budeme zabývat analýzou doby běhu algoritmu *merge sort*.

2.1 Insertion sort

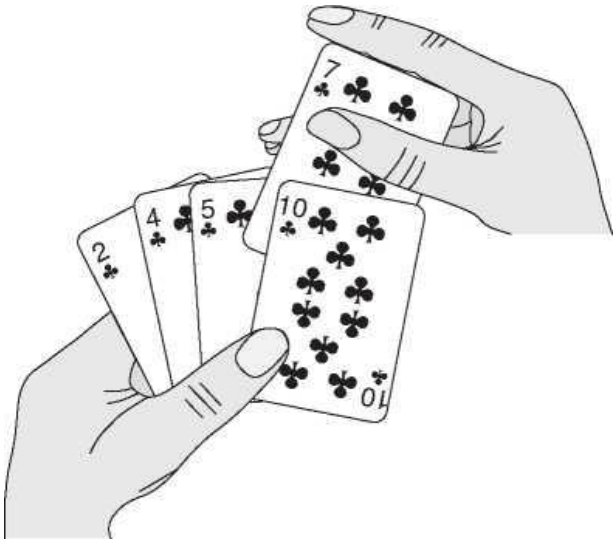
Náš první algoritmus řeší problém řazení představený v kapitole 1:

Vstup: posloupnost n čísel $\langle a_1, a_2, \ldots, a_n \rangle$.

Výstup: permutace (přeuspořádání) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ ze vstupní posloupnosti takové, že $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Čísla, která si přejeme seřadit, se nazývají *keys* (klíče). Ačkoliv z hlediska pojmů řadíme posloupnosti, tak vstup nám představuje pole *n* prvků.

V této publikaci popíšeme algoritmy jako programy psané v ***pseudokódu*** podobném v mnoha ohledech jazykům C, C++, Java, Python, nebo Pascal. Pokud jste byli seznámeni s některým z těchto jazyků, nemělo by vám činit čtení algoritmů velké problémy.



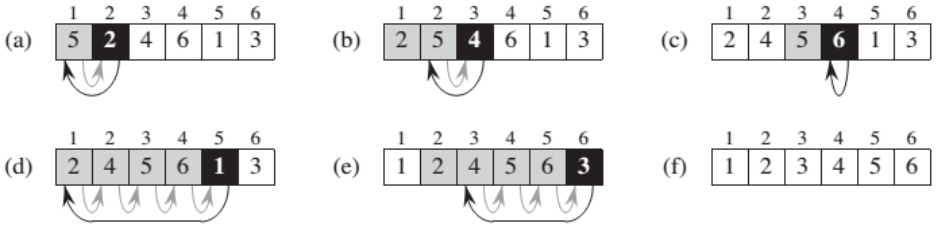
Ilustrace 2.1 Zařazení karet v ruce pomocí *insertion sort*.

To, co odděluje pseudokód od „reálného“ kódu, je fakt, že v pseudokódu uplatňujeme jakoukoliv výstižnou metodu, která je většinou pochopitelná a upřesňuje daný algoritmus. Někdy se jeví angličtina jako nejpochopitelnější forma, a proto nebuďte překvapeni, pokud narazíte na slovní popis začleněný do části „reálného“ kódu. Dalším rozdílem mezi pseudokódem a reálným kódem je, že pseudokód se nezabývá otázkami softwarového inženýrství. Záležitosti jako abstrakce dat, modularita a řešení chyb jsou často ignorovány, aby byla stručněji vyjádřena podstata algoritmu.

Začneme s ***insertion sort***, který je účinným algoritmem pro řazení malého množství prvků. *Insertion sort* funguje stejně, jako když si lidé seřazují karty v ruce. Začínáme s prázdnou levou rukou a karty jsou lícem dolů na stole. Poté sejmeme jednu kartu ze stolu a vložíme ji do

levé ruky na správnou pozici. Abychom našli pozici pro kartu, musíme kartu porovnat s každou kartou v levé ruce, a to zprava doleva, viz ilustrace 2.1. Karty v levé ruce, což jsou karty, které ležely původně na stole, jsou neustále seřazeny.

Ukážeme si náš pseudokód jako proceduru zvanou INSERTION-SORT, která bere jako parametr pole $A[1 .. n]$ obsahující posloupnost o délce n, jež má být seřazena (v kódu je počet prvků n v A označeno jako *A.length*). Tento algoritmus řadí vstupní čísla v jejich původním umístění. To znamená, že přeuspořádává čísla uvnitř pole A s tím, že jich je neustále nejvýše konstantní počet uložen mimo pole A. Když je procedura INSERTION-SORT hotova, vstupní pole A obsahuje seřazenou posloupnost.



Ilustrace 2.2 Aplikace procedury INSERTION-SORT na pole $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Indexy pole se objevují nad obdélníky a hodnoty jsou uloženy v těchto obdélnících. **(a)-(e)** představují iterace cyklů **for** na řádcích 1-8. V každé iteraci uchovává černý obdélník *key* převzatý z $A[j]$, který je porovnán s hodnotami v šedých obdélnících nalevo od něj (řádek 5). Šedé šipky ukazují pohyb hodnot pole o jednu pozici doprava (řádek 6) a černé šipky představují, kam se přesouvá *key* (řádek 8). **(f)** je pak seřazeným polem.

```
INSERTION-SORT (A)
1   for j = 2 to A.length
2       key = A[j]
3       // Vlož A[j] do seřazené posloupnosti A[1 .. j – 1].
4       i = j – 1
5       while i > 0 and A[i] > key
6           A[i + 1] = A[i]
7           i = i – 1
8       A[i + 1] = key
```

Invarianty cyklu a správnost insertion sort

Ilustrace 2.2 ukazuje, jak funguje algoritmus pro $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Index j ukazuje vybranou kartu, kterou vložíme do ruky. Na začátku každé iterace cyklu **for**, která je zaindexována pomocí j, se subpole $A[1 .. j - 1]$ skládá z aktuálně seřazených karet v ruce a zbývající subpole $A[j + 1 .. n]$ odpovídá balíčku karet, který je stále na stole. Prvky $[1 .. j - 1]$ jsou prvky, které byly původně na pozici 1 až $j - 1$, ale nyní jsou v seřazeném pořadí. Uvádíme tedy vlastnosti $A[1 .. j - 1]$ formálně jako ***invariant cyklu***:

> Na začátku každé iterace cyklu **for**, na řádcích 1-8, se subpole $A[1 .. j - 1]$ skládá z prvků nacházejících se původně v $A[1 .. j - 1]$, ale v seřazeném pořadí.

Invarianty cyklu používáme proto, aby nám pomohly pochopit, proč je algoritmus správně. Musíme si tedy ukázat tři věci týkající se invariantu cyklu:

Inicializace: Je pravdivá před první iterací cyklu.

Údržba: Pokud je uchování pravdivé před libovolnou iterací cyklu, pak zůstává pravdivé i před další iterací cyklu.

Ukončení: Když cyklus končí, invariant předá užitečnou vlastnost, která nám pomůže pochopit správnost algoritmu.

Když jsou první dvě vlastnosti dodrženy, invariant cyklu je pravdivý pro každou iteraci cyklu (samozřejmě můžeme uvažovat i jiná prokázaná fakta než samotný invariant cyklu, abychom dokázali, že invariant cyklu zůstává před každou iterací pravdivý). Poukažme na podobnost s matematickou indukcí, kde k tomu, abychom dokázali, že vlastnost platí, dokážeme platnost prvního kroku a indukčního kroku. Zde invariant platí před první iterací, což odpovídá prvnímu kroku a dále invariant platí od iterace k iteraci, což odpovídá indukčnímu kroku.

Třetí vlastnost je snad tou nejdůležitější, neboť pomocí invariantu cyklu ukazuje správnost algoritmu. Za běžných okolností používáme invariant cyklu s podmínkou, která způsobuje ukončení cyklu. Vlastnost ukončení odlišuje od toho, jak používáme obvykle matematickou

indukci, ve které aplikujeme indukční krok nekonečněkrát; tady ale zastavíme „indukci" tehdy, když cyklus skončí.

Podívejme se na to, jak jsou tyto vlastnosti dodrženy u *insertion sort*.

Inicializace: Poukázáním na to, že invariant cyklu, platí před první iterací právě tehdy, když $j = 2$.[4] Subpole $A[1 .. j - 1]$ se tedy skládá právě z jediného prvku $A[1]$, který je defacto původním prvkem v $A[1]$. Kromě toho je toto subpole seřazeno (triviálně) tak, což ukazuje, že invariant cyklu platí před první iterací cyklu.

Údržba: V dalším kroku se pouštíme do kontroly druhé vlastnosti a ukazujeme, že každá iterace uchovává invariant cyklu. Neformálně řečeno, tělo cyklu **for** funguje na základě přesouvání $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, atd. o jednu pozici vpravo, dokud nenajde vhodnou pozici pro $A[j]$ (řádky 4-7), na kterou se vloží hodnota $A[j]$ (řádek 8). Subpole $A[1 .. j]$ se poté skládá z prvků původně se nacházejících v $A[1 .. j]$, ale v seřazeném pořadí. Inkrementací j pro další iteraci cyklu for dochází k zachování invariantu cyklu.

Formálnější zacházení s druhou vlastností by od nás vyžadovalo stanovení a určení invariantu pro cyklus **while** na řádcích 5-7. Avšak v tuto chvíli dáváme přednost nezastavovat se nad tímto formalismem, a tak se spoléháme na naši neformální analýzu ukazující dodržení druhé vlastnosti pro vnější cyklus.

Ukončení: Tímto se dostáváme k prozkoumání toho, co se děje s cyklem, když končí. Podmínka, která způsobuje ukončení cyklu **for**, je $j > A.length = n$. Protože každá iterace cyklu zvyšuje hodnotu čítače j o 1, musíme mít zároveň $j = n + 1$. Nahrazením $n + 1$ za čítač j ve formulaci invariantu získáme subpole $A[1 .. n]$ skládající se původně z prvků v $A[1 .. n]$, ale v seřazeném pořadí. Pozorováním faktu, že

[4] Pokud se jedná o cyklus **for**, pak moment, kdy před první iterací cyklu kontrolujeme invariant cyklu, nastává okamžitě po počátečním přiřazení do čítače a právě před prvním testem v hlavičce cyklu. V případě INSERTION-SORT k tomu dochází po přiřazení hodnoty 2 do proměnné j, ale ještě před prvním testem, zda je $j \leq A.length$.

subpole $A[1 .. n]$ je celé pole, můžeme vyvodit závěr o správnosti algoritmu, neboť je celé pole seřazeno.

Tuto metodu invariantu cyklu použijeme, abychom znázornili správnost i dalších algoritmů v následujících kapitolách.

Konvence pseudokódu

V našem pseudokódu používáme následující pravidla.

- Odsazení znázorňuje blokovou strukturu, například tělo cyklu **for** začíná na řádku 1 a skládá se z řádků 2-8 a tělo cyklu **while** začíná na řádku 5 a obsahuje řádky 6-7 kromě řádku 8. Náš styl odsazení se používá také na **if-else** větve[5]. Pomocí odsazení namísto standardních ukazatelů blokové struktury, jako například **begin** a **end**, dochází k značnému zmenšení nepořádku, zatímco je zachována či dokonce zlepšena srozumitelnost.[6]
- Cykly **while**, **for**, **repeat-until** a **if-else** podmínka mají interpretaci podobnou svým protějškům v C, C++, Java, Python a Pascal.[7] V této publikaci si čítač ponechává svou hodnotu po ukončení cyklu na rozdíl od některých situací, které nastávají v C++, Java a Pascal. Proto je okamžitě po cyklu for hodnota čítače tou první překročenou mezí. Použili jsme tuto vlastnost v ověření správnosti zkoumání *insertion sort*. Hlavička cyklu **for** na řádku 1 je **for** $j = 2$ **to** $A.length$, a tak, když cyklus končí, $j = A.length + 1$ (nebo ekvivalentně $j = n + 1$, kde $n = A.length$). Pro inkrementaci čítače cyklu **for** v každé iteraci používáme klíčové slovo **to**, pro jeho dekrementaci používáme klíčové slovo **downto**. Když se mění hodnota čítače cyklu o více než 1, užívá se klíčové slovo **by**.

---

[5]V **if-else větvi** odsazujeme **else** na stejnou úroveň jako podmínku **if**. Ačkoliv vynecháváme klíčové slovo **then**, občas odkazujeme k provedenému úseku za předpokladu splnění testu v podmínce **if**, tzn. k ***then***. Větev **elseif** používáme pro vícecestné testy po prvním otestování podmínky **if**.

[6]Každá procedura pseudokódu v této publikaci se objevuje vždy na jedné straně, takže nebudete muset rozlišovat odsazení v kódu, který je rozdělen na další stranu.

[7]Většina blokově strukturovaných jazyků má ekvivalentní koncepty, ačkoliv se přesná syntaxe může lišit. Python například postrádá cyklus **repeat-until** a jeho cykly **for** fungují trochu odlišně v porovnání s cykly **for** v této publikaci.

- Symbol „//" označuje zbytek řádku jako komentář.
- Mnohonásobné přiřazení ve tvaru $i = j = e$ přiřazuje oběma proměnným i a j hodnotu výrazu e; tento výraz by měl být brán v potaz jako ekvivalent přiřazení $j = e$ následovaný přiřazením $i = j$.
- Lokálními proměnnými dané procedury jsou i, j a *key*. Nebudeme tedy používat globální proměnné bez explicitního označení.
- K prvkům pole přistupujeme pomocí vymezení jména pole následovaného indexem v hranatých závorkách, například $A[i]$ představuje i-tý prvek pole A. Notace „.." se používá pro znázornění rozmezí hodnot uvnitř pole. Proto $A[1 .. j]$ představuje subpole A skládající se z j prvků $A[1], A[2], \ldots, A[j]$.
- Za běžných podmínek se složená data uspořádávají do ***objektů*** skládajících se z ***atributů.*** Ke konkrétním atributům přistupujeme pomocí syntaxe v mnoha objektově orientovaných jazycích, tj. jméno objektu následované tečkou a poté následované jménem atributu. S polem zacházíme například jako s objektem majícím atribut *length*, který ukazuje kolik má pole prvků. Píšeme tedy *A.length*, abychom upřesnili počet prvků v poli *A*.

  S proměnnou reprezentující pole či objekt zacházíme jako s ukazatelem na data ve formě pole nebo objektu. Pro všechny atributy f objektu x nastavením $y = x$ způsobíme, že $y.f$ je rovno $x.f$. Kromě toho, pokud nastavíme $x.f = 3$, nejenže se $x.f$ bude rovnat 3, ale i $y.f$ se bude rovnat 3. Jinými slovy, po přiřazení $y = x$ ukazují x a y na stejný objekt.

  Naše notace atributu se může stupňovat. Předpokládejme například, že atribut f je sám ukazatelem na nějaký druh objektu mající atribut g. Poté je notace $x.f.g$ implicitně uzávorkována stejně jako $(x.f).g$. Jinými slovy, jestliže jsme provedli přiřazení $y = x.f$, poté $x.f.g$ je to samé jako $y.g$.

  Někdy ukazatel neodkazuje na žádný objekt a v tomto případě přiřazujeme speciální hodnotu NIL.
- Procedurou předáváme parametry ***by value***. Volaná procedura obdrží

svou vlastní kopii parametrů, a pokud přiřadí hodnotu do některého z parametrů, tato změna není volající procedurou viditelná. Když jsou předány objekty, dojde ke zkopírování ukazatele na data zastupující objekt, ale bez atributů objektu. Například, pokud je *x* parametrem volané procedury, není přiřazení *x* = *y* uvnitř volané procedury viditelné pro volající proceduru, avšak přiřazení *x.f* = 3 již viditelné je. Podobně jsou pole předávána pomocí ukazatele tak, že je předán ukazatel na pole spíše než celé pole a změny provedené na jednotlivých prvcích jsou viditelné pro volající proceduru.

- Příkaz **return** okamžitě předává řízení zpět do bodu volání ve volající proceduře. Většina příkazů **return** bere hodnotu a předá ji zpět dobodu volání. Náš pseudokód se liší od mnohých programovacích jazyků tím, že dovolujeme jedním příkazem **return** vrátit několik hodnot.
- Booleovské operátory „and" a „or" se vyhodnocují pomocí ***líného vyhodnocování***. To znamená, že když vyhodnotíme výraz „*x* and *y*", vyhodnocujeme nejprve *x*. Jestliže je *x* vyhodnoceno jako NEPRAVDA, nemůže být poté celý výraz vyhodnocen jako PRAVDA, tím pádem nedochází k vyhodnocení *y*. Na druhou stranu, jestliže je *x* vyhodnoceno jako PRAVDA, pak musíme vyhodnotit *y*, abychom posoudili pravdivostní hodnotu výrazu. Podobně je tomu ve výrazu„*x* or *y*" – vyhodnocujeme výraz y pouze tehdy, když je *x* vyhodnoceno jako NEPRAVDA. Líné vyhodnocování nám umožňuje psát booleovské výrazy jako „x ≠ NIL and *x.f* = *y*" bez obav o to, co se stane, pokud se pokusíme vyhodnotit hodnotu *x.f*, když je x NIL.
- Klíčové slovo **error** značí chybu ve volané proceduře, pokud se vyskytla chyba v podmínkách. Volající procedura je zodpovědná za řešení chyby, a proto nespecifikujeme, co má být provedeno.

Cvičení

2.1-1

Použijte ilustraci 2.2 jako vzor a ilustrujte průběh výpočtu INSERTION-SORT na poli $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Přepište proceduru INSERTION-SORT na nerostoucí řazení místo neklesajícího řazení.

2.1-3

Zvažte tento ***problém hledání****:*

Vstup*:* řada n čísel pole $A = \langle a_1, a_2, \ldots, a_n \rangle$ a hodnota v.

Výstup*:* index i takový, že $v = A[i]$ nebo speciální hodnotě NIL, jestliže se v neobjevuje v poli A.

Napište pseudokód pro ***lineární hledání****,* který projde řadu a bude hledat hodnotu v. Pomocí invariantu cyklu dokažte, že je váš algoritmus správný. Ujistěte se, že invariant cyklu splňuje tři potřebné vlastnosti.

2.1-4

Zvažte problém přidávání dvou n-bitových binárních čísel uložených v n-prvkových polích A a B. Součet těchto dvou čísel by měl být uložen v binární formě $(n + 1)$-prvkovém poli C. Uveďte problém formálně a napište pseudokód na přidání dvou čísel.

2.2 Analyzování algoritmů

Naše ***analýza*** algoritmů pokročila až ke způsobu předpovídání množství zdrojů, které algoritmy potřebují. Zdroje, jako například paměť, šířka komunikačního pásma nebo hardware, jsou příležitostně primárním zájmem, ale nejčastěji je to výpočetní čas, který chceme změřit. Obecně vzato, analyzováním několika algoritmů řešících daný problém můžeme identifikovat ten nejúčinnější. Tato analýza může ukázat více než

jednoho kandidáta, ale může také často vyřadit několik horších algoritmů.

Předtím, než můžeme analyzovat algoritmus, musíme mít model implementační technologie, který použijeme, a to včetně modelu zdrojů této technologie a její ceny. Pro většinu úloh v této publikaci uvažujme model jednoprocesorového stroje s pamětí s přímým přístupem (***RAM*** - ***random-access machine***) na výpočty, jehož technologii implementujeme pro naše analýzy, a pochopme, že náš algoritmus bude implementován jako počítačové programy. V RAM modelu jsou prováděny instrukce jedna po druhé, a to bez souběžných operací.

V přesném slova smyslu bychom měli důkladně definovat instrukce RAM modelu a jejich náročnost na zdroje. Učinit tak by bylo únavné a přineslo by to jen malý náhled do návrhu algoritmu a jeho analýzy. Je potřeba se mít na pozoru před nesprávným použitím RAM modelu, například co kdyby měl RAM model instrukci, která má seřadit? Poté můžeme řadit pouze v jedné instrukci. Takový RAM model by byl nerealistický, neboť skutečné počítače nemají takovéto instrukce. Náš návod se proto zabývá tím, jak jsou skutečné počítače navrženy. Tento model obsahuje instrukce, jež se dají obvykle nalézt ve skutečných počítačích: aritmetické instrukce (např. součet, odečítání, násobení, dělení, zbytek, zaokrouhlování na celá čísla nahoru a dolů), pohyb dat (načítání, ukládání, kopírování) a řízení (podmíněné a nepodmíněné větvení, volání podprogramu a navrácení hodnot). Každá tato instrukce trvá konstantní čas.

Celá čísla a čísla s pohyblivou řádovou čárka (pro ukládání reálných čísel) jsou datovými typy v RAM modelu. I když se většinou přesností v této publikaci nezabýváme, je v některých aplikacích stěžejní. Také předpokládáme limit velikosti dat pro každé slovo. Například když se pracuje se vstupy o velikosti n, normálně uvažujeme, že celá čísla zastoupená $c \lg n$ bitů pro libovolné konstantu $c \geq 1$. Požadujeme, aby $c \geq 1$, díky němuž může každé slovo obsahovat hodnotu n a délka slova neroste libovolně (pokud by délka slova mohla růst libovolně, mohli bychom ukládat obrovské množství dat v jednom slově a pracovat s nimi v konstantním čase – čistě nereálný scénář).

Skutečné počítače obsahují instrukce, které zde nejsou uvedeny. Takové instrukce představují šedou oblast v RAM modelu. Je například umocnění časově konstantní instrukcí? V obecném případě ne. Výpočet x^y , kdy x a y jsou reálná čísla, představuje několik instrukcí. Avšak ve vyhrazených situacích je umocnění časově konstantní operací. Mnoho počítačů má „shift left" instrukci, která v konstatním čase posouvá bity celého čísla o k pozic doleva. U většiny počítačů je takové posouvání bitů o k pozic doleva ekvivalentní násobení 2, čili tento posun bitů o k pozic doleva je ekvivalentní násobení 2^k. Proto tedy tyto počítače mohou spočítat 2^k v jedné časově konstantní instrukci pomocí posouvání celého čísla o k pozic doleva, pokud není k větší než počet bitů ve slově. Budeme usilovat o to, abychom se vyhnuli těmto šedým oblastem v RAM modelu, ale budeme zacházet s výpočtem 2^k jako s konstantně časovou operací, kdy k je dostatečně malé kladné číslo.

V RAM modelu se nepokoušíme vytvářet strukturu paměti, jež je běžná pro současné počítače – to proto, že nevytváříme mezipamět (cache) nebo virtuální pamět. Několik výpočetních modelů se pokouší brát v potaz důsledky strukturování paměti, které jsou občas významné pro reálný program na skutečných strojích. Hrstkou problémů se tato publikace zabývá, ale ve většině částí této publikace je analýza nebere v potaz. Modely zahrnující strukturu paměti jsou trochu více komplexní než RAM model, a proto se s nimi pracuje obtížně. Kromě toho jsou RAM model analýzy obvykle skvělými ukazateli výkonu současných strojů.

Analyzování i jednoduchého algoritmu v RAM modelu může být výzvou. Matematické nástroje potřebné k analýze zahrnují kombinatoriku, teorii pravděpodobnosti, algebraické znalosti a schopnost identifikovat nejdůležitější termy ve formuli. Protože se chování algoritmu může měnit pro každý možný vstup, potřebujeme prostředky pro shrnutí takového chování v jednoduchých a lehce pochopitelných formulích.

I přesto, že volíme pouze jeden model stroje k analýze algoritmu, čelíme stále mnoha volbám při rozhodnutí, jak vyjádřit naši analýzu. Chtěli bychom jednoduchý způsob na zapsání a manipulaci, který

ukazuje důležité charakteristiky požadavků algoritmu na zdroje a omezit nudné detaily.

Analýza insertion sort

Čas potřebný na provedení INSERTION-SORT procedury zavisí na vstupu - řazení tisíce čísel trvá déle než seřazení tří čísel. Kromě toho, vykonání INSERTION-SORT může trvat rozdílnou dobu pro dvě vstupní posloupnosti stejné velikosti v závislosti na tom, jak moc jsou již seřazeny. Obecně čas roste s velikostí vstupu a je na místě popsat dobu běhu programu jako funkci velikosti jeho vstupu. Aby se tak stalo, je zapotřebí definovat důkladněji pojmy *doba běhu* a *velikost vstupu* s opatrností.

Pohled na ***velikost vstupu*** zavisí na problému, kterým se zabýváme. Pro mnoho problémů, jako například řazení nebo počítání diskrétních Fourierových transformací, je nejpřirozenější mírou počet položek na vstupu – například pole velikosti n na seřazení. Pro mnoho dalších problémů, jako například pro násobení dvou celých čísel, je nejlepší mírou velikosti vstupu celkový počet bitů potřebných pro vyjádření vstupu v běžné binární notaci. Někdy je vhodnější popsat velikost vstupu dvěma čísly než pouze jedním. Například, jestliže je vstup pro algoritmus graf, může být velikost vstupu popsána počtem vrcholů a hran. Budeme tedy vyjadřovat, jaké měření velikosti vstupu je užito u každého zkoumaného problému.

Doba běhu algoritmu na konkrétním vstupu je počet elementárních operací nebo provedených „kroků“. Je vhodné definovat pojem krok, aby byl nezávislý na stroji tak, jak jen to bude možné. Pro tento okamžik se přizpůsobme následujícímu pohledu: konstantní množství času je vyžadováno pro vykonání každého řádku našeho pseudokódu. Vykonání jednoho řádku může trvat rozdílnou dobu než u jiného řádku, ale v našem případě budeme uvažovat provedení i-tého řádku trvajícího c_i času, kde

c_i je konstantní. Tento pohled je v souladu s RAM modelem a odráží, jak by byl pseudokód implementován na většině současných počítačů.[8]

V následujícím pojednání se naše vyjádření pro dobu běhu INSERTION-SORT vyvine z neuspořádaného vzorce, který používá všechny ceny výrazu c_i, do mnohem jednodušší, shrnující a snadněji uchopitelné notace. Tato jednodušší notace také usnadní rozhodnutí, zda je jeden algoritmus účinnější než druhý. Začneme s ukázkou procedury INSERTION-SORT s „cenou" v jednotkách času každého výrazu a počtem dob vykonání každého výrazu. Pro každé $j = 2, 3, \ldots, n$, kde $n = A.length$, nechť j označuje kolikrát se cyklus **while** na řádku 5 provede pro dané j. Když cyklus **for** nebo **while** skončí běžným způsobem (tj. kvůli testu v hlavičce cyklu), je test proveden o jedenkrát více než tělo cyklu. Předpokládáme, že komentáře nejsou vykonatelné výrazy, a tudíž netrvají žádný čas.

INSERTION-SORT (A)	*cena*	*počty*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Vložte $A[j]$ do seřazené posloupnosti $A[1 .. j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

Doba běhu algoritmu je součtem časů běhu pro každou provedený výraz; ta potřebuje k provedení c_i kroků, provádí se n-krát a přispěje

[8]Zde se nachází jisté nuance.Výpočetní kroky, které specifikujeme v angličtině, jsou často variantami procedury vyžadující více než konstantní množství času. Později můžeme například říct: „seřaďte body podle souřadnice x", což, jak uvidíme, potrvá déle než konstantní délku času. Poznamenejme též, že příkaz, který volá podprogram, trvá konstantní čas, ačkoliv jakmile je tento podprogram spuštěn, může trvat déle. Tedy oddělujeme proces volání podprogramu – předání parametrů podprogramu, atd.- od procesu provedení podprogramu.

$c_i n$ k celkové době běhu.[9] Abychom vypočítali $T(n)$, čili dobu běhu INSERTION-SORT pro n vstupních hodnot, musíme sečíst výsledky součinů hodnot ze sloupců *cena* a *počty*, čímž získáme dobu běhu pro algoritmus

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Dokonce i pro vstupy dané velikosti může doba běhu algoritmu záviset na tom, který vstup dané velikosti je předán. Například u INSERTION-SORT nastává nejlepší případ tehdy, když je pole již seřazeno. Pro každé $j = 2, 3, \ldots, n$, zjistíme, že je $A[i] \leq key$ na řádku 5, pokud i má počáteční hodnotu $j - 1$. Tedy $t_j = 1$ je pro $j = 2, 3, \ldots, n$ a doba běhu nejlepšího případu je

$$\begin{aligned} T(n) &= C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1) \\ &= (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8). \end{aligned}$$

Tuto dobu běhu můžeme vyjádřit jako $an + b$ pro konstanty a a b zavisející na ceně výrazu c_i; je to ***lineární funkce*** na n.

Pokud je pole v seřazeném obráceném pořadí, tj. sestupně seřazeném, představuje pak nejhorší případ. Musíme porovnat každý prvek $A[j]$ s každým prvkem celého seřazeného subpole $A[1..j-1]$, a tak platí $t_j = j$ pro $j = 2, 3, \ldots, n$. Poznamenáváme, že

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(viz příloha A pro kontrolu, jak vyřešit tyto sumy) v nejhorším případě je doba běhu INSERTION-SORT

[9]Tato charakteristika není nezbytně relevantní pro zdroje jako například paměť. Výraz, který adresuje m slov paměti a je prováděn n-krát, neadresuje nezbytně na mn různých slov paměti.

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}
$$

Nejhorší případ doby běhu můžeme vyjádřit jako $an^2 + bn + c$ pro konstanty a, b a c, které opět závisí na ceně výrazu c_i; a proto se jedná o **kvadratickou funkci** na n.

Za normálních okolností, stejně jako u *insertion sort*, je doba běhu algoritmu pro daný vstup pevně stanovena, ačkoliv v dalších kapitolách se setkáme s některými zajímavými randomizovanými algoritmy, jejichž chování se může lišit i pro pevně daný vstup.

Analýza nejhorší a průměrné doby běhu

V naší analýze *insertion sort* jsme se podívali jak na nejlepší případ, kde bylo vstupní pole již seřazeno, tak na nejhorší případ, kde bylo vstupní pole seřazeno obráceně. Ve zbylé části publikace se zaměřujeme pouze na hledání ***nejhorších případů doby běhu***, tj. nejdelší doby běhu pro jakýkoliv vstup velikosti n. Pro toto zaměření máme tři důvody.

- Nejhorší případ doby běhu algoritmu nám stanovuje horní mez doby běhu pro jakýkoliv vstup. Tím dostáváme záruku, že provedení algoritmu nebude trvat déle. Nemusíme tedy vytvářet nějaké odborné dohady o době běhu a doufat, že se situace nikdy nezhorší.
- Nejhorší případ nastává pro některé algoritmy celkem často. Například když se prohledává databáze kvůli konkrétní informaci, nastane často nejhorší případ hledacího algoritmu, pokud se informace v databázi nenachází. Hledání chybějících informací může být pro některé aplikace časté.
- Průměrná doba běhu algoritmu je často zhruba stejně tak špatná jako nejhorší případ. Předpokládejme, že náhodně vybereme n čísel a použijeme *insertion sort*. Jak dlouho bude trvat určení, kam do subpole $A[1 .. j-1]$ vložit prvek $A[j]$? Průměrně je polovina prvků v $A[1 .. j-1]$ menší než hodnota $A[j]$ a hodnoty zbylé

poloviny prvků jsou větší. A proto kontrolujeme průměrně polovinu subpole $A[1 .. j - 1]$, tedy t_j je zhruba j/2. Výsledný případ průměrné doby běhu se ukazuje jako kvadratická funkce na velikost vstupních hodnot, stejně jako doba běhu algoritmu nejhoršího případu.

V některých konkrétních případech se budeme zajímat o průměrnou dobu běhu algoritmu a seznámíme se s metodou pravděpodobnostní analýzy pro různé algoritmy v této publikaci. Rámec analýzy je omezen, neboť by nemohlo být zřejmé, co představuje „průměrný" vstup pro konkrétní problém. Často budeme předpokládat, že všechny vstupy dané velikosti jsou stejně pravděpodobné. V praxi může být tento předpoklad porušen, ale někdy můžeme použít randomizovaný algoritmus, který dělá náhodná rozhodnutí, aby umožnil provedení analýzy prvaděpodobnosti a vykazuje **očekávané** doby běhu. Randomizované algoritmy budeme více prozkoumávat v kapitole 5 a několika dalších následujících kapitolách.

Velikost růstu

Pro ulehčení analýzy procedury INSERTION-SORT jsme použili některé zjednodušující abstrakce. Nejprve jsme ignorovali skutečnou cenu každého výrazu, a to stanovením konstant c_i představujících tuto hodnotu. Poté jsme pozorovali, že i tyto konstanty nám dávají více informací, než jsme skutečně potřebovali. Vyjádřili jsme nejhorší případ běhu času jako $an^2 + bn + c$ pro libovolné konstanty a, b a c závislé na cenách výrazu c_i. Proto jsme neignorovali pouze skutečné ceny výrazů, ale také i abstraktní hodnoty c_i.

Provedeme ještě jednu zjednodušující abstrakci: jde o ***míru růstu*** nebo ***velikost růstu*** doby běhu, která nás opravdu zajímá. Budeme tedy uvažovat pouze hlavní člen rovnice (např. an^2) vzhledem k tomu, že hodnoty termů nižšího řádu jsou pro vysoké hodnoty relativně bezvýznamné. Ignorujeme také konstantní koeficient hlavního termu, který je méně významný při podmiňování výpočetní účinnosti pro velké

vstupy než míra růstu. Pro *insertion sort*, kdy ignorujeme členy nižšího řádu a konstantní koeficient hlavního termu, nám zbývá prvek n^2 z hlavního členu. Zapisujeme, že *insertion sort* má v nejhorším případě dobu běhu $\Theta(n^2)$ (vyslovováno „théta *n* na druhou"). V této kapitole použijeme Θ-notaci neformálně a budeme ji přesně definovat v kapitole 3.

Obvykle považujeme jeden algoritmus za účinnější než ten druhý, jestliže nejhorší doba běhu má nižší rychlost růstu. Kvůli konstantám a členům nižšího řádu, může provedení algoritmu, jehož doba běhu má vyšší rychlost růstu, trvat kratší dobu pro menší vstupy než algoritmus, jehož doba běhu má nižší rychlost růstu. Ale s dostatečně velkými vstupy, například $\Theta(n^2)$ algoritmus poběží stále rychleji než $\Theta(n^3)$ algoritmus v tom nejhorším případě.

Cvičení

2.2-1

Vyjádřete funkci $n^3/1000 - 100n^2 - 100n + 3$ v pojmech Θ-notace.

2.2-2

Zvažte seřazení *n* čísel uložených v poli *A* nalezením prvního nejmenšího prvku *A* a jeho výměnu za prvek v *A*[1]. Poté najděte druhý nejmenší prvek *A* a vyměňte ho s *A*[2]. Pokračujte tímto způsobem pro prvních $n - 1$ prvků *A*. Napište pseudokód pro tento algoritmus, který je znám jako ***selection sort*** (seřazení výběrem). Který invariant cyklu uchovává tento algoritmus? Proč je potřeba spustit algoritmus pouze pro prvních $n - 1$ prvků spíše než pro všechny *n* prvky? Uveďte nejlepší a nejhorší případ doby běhu *selection sort* v Θ-notaci.

2.2-3

Zvažte znova lineární prohledávání (viz cvičení 2.1-3). Kolik prvků vstupní posloupnosti bude zapotřebí průměrně projít za předpokladu, že hledaný prvek je se stejnou pravděpodobností jako jakýkoliv prvek v poli? Jak to bude s nejhorším případem? Jaký je průměrný a nejhorší

případ doby běhu lineárního prohledávání v Θ-notaci? Zdůvodněte vaši odpověď.

2.2-4

Jak můžete změnit většinu jakýchkoliv algoritmů, abyste dosáhli v nejlepším případě dobré doby běhu?

2.3 Navrhování algoritmů

Máme na výběr z širokého rozmezí technik návrhů algoritmů. Pro *insertion sort* jsme použili **inkrementační** přístup, tj. do seřazeného subpole $A[1 .. j - 1]$ jsme na vhodné místo vložením jednoho prvku $A[j]$ na správné místo získali seřazené subpole $A[1 .. j]$.

V této části se budeme zabývat alternativním přístupem k návrhu známým jako „rozděl a panuj" (*divide-and-conquer*), který prozkoumáme detailněji v kapitole 4. Tento přístup použijeme k návrhu řadícího algoritmu, jehož doba běhu v nejhorším případě je mnohem menší než u *insertion sort*. Jednou výhodou těchto algoritmů je, že jejich doby běhu jsou často lehce určitelné pomocí technik, se kterými se seznámíme v kapitole 4.

2.3.1 Přístup rozděl a panuj

Mnoho užitečných algoritmů je ve své struktuře ***rekurzivní***: k vyřešení problému volají rekurzivně samy sebe jednou či víckrát s cílem vyřešit úzce spjaté podproblémy. Tyto algoritmy běžně následují přístup ***rozděl a panuj***: rozdělí problém do několika podproblémů, které jsou podobné původnímu problému, ale mají menší velikost. Tyto podproblémy se řeší rekurzivně, a poté se jejich řešení kombinují za účelem vytvoření řešení původního problému.

Toto paradigma zahrnuje na každé úrovni rekurze tři kroky:

Rozděl problém na počet podproblémů, které jsou menšími

instancemi stejného problému.

Panuj nad podproblémy tím, že je vyřešíš rekurzivně. Pokud je ale podproblém dostatečně malý, tak řeš podproblémy přímočaře.

Zkombinuj řešení podproblémů a vytvoř řešení pro původní problém.

Algoritmus ***merge sort*** (slévání) úzce následuje paradigma rozděl a panuj. Funguje intuitivně podle následujících kroků:

Rozděl: Rozdělte *n*-prvkovou posloupnost určenou k seřazení na dvě posloupnosti, každou o *n*/2 prvcích.

Panuj: Setřiďte tyto dvě posloupnosti rekurzivně pomocí *merge sort*.

Zkombinuj: Spojte tyto dvě posloupnosti, abyste vytvořili seřazené řešení.

Rekurze dosáhne limitní podmínky, je-li posloupnost určená k seřazení délky 1. V tomto případě rekurze neprobíhá, neboť je každá posloupnost o délky 1 již seřazena.

Klíčovou operací algoritmu *merge sort* je spojování dvou řad v kroku kombinace. Sléváme je voláním pomocné procedury MERGE (A, p, q, r), kde A představuje pole a p, q a r jsou indexy v poli takové, že $p \leq q < r$. Procedura předpokládá, že subpole $A[p \,..\, q]$ a $A[q + 1..r]$ jsou v seřazeném pořadí. To je slévá do formy jednoho seřazeného pole nahrazujícího současné subpole $A[p \,..\, r]$.

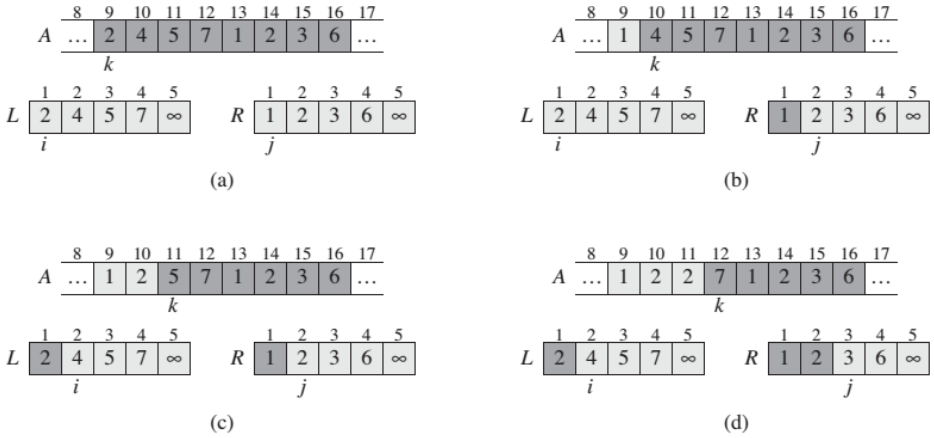
Naše procedura MERGE trvá $\Theta(n)$ času, kde je $n = r – p + 1$ součtem spojovaných prvků, a funguje podle následujícího popisu. Vrátíme se k případu s kartami a předpokládejme, že máme na stole dva balíčky karet lícem nahoru. Každý balíček je seřazen tak, že nejmenší karta balíčku je na vrchu balíčku. Přejeme si slít tyto dva balíčky do jednoho seřazeného tak, že bude na stole lícem dolů. Náš základní krok se skládá z výběru menší ze dvou karet otočených lícem nahoru z vrchu balíčku, jejího odstranění z balíčku, což odkryje novou vrchní kartu, a umístění této karty lícem dolů na výsledný balíček. Tento krok opakujeme do doby, než je jeden ze vstupních balíčků prázdný. V tuto chvíli už jen vezmeme zbývající vstupní balíček a umístíme ho lícem dolů na

výsledný balíček. Z výpočetního hlediska trvá každý krok konstantní čas, protože porovnáváme právě dvě svrchní karty. Vzhledem k tomu, že provádíme nejvýše n základních kroků, trvá slévání $\Theta(n)$ času.

Následující pseudokód implementuje výše zmíněnou myšlenku, ale s přidaným vylepšením odstraňujícím potřebu kontrolovat v každém základním kroku, zda je balíček prázdný. Na dno balíčku umisťujeme *sentinel card* (mezní kartu), obsahující zvláštní hodnotu, kterou zjednodušujeme náš kód. Použijeme ∞ jako *sentinel value* (mezní hodnotu), takže kdykoliv se odhalí karta s hodnotou ∞ v obou balíčcích, nemůže se objevit karta menší hodnoty. Jakmile k tomu dojde, jsou všechny ostatní karty vyskládány do výsledného balíčku. Protože dopředu víme, že bude $r - p + 1$ karet umístěno na vrch výsledného balíčku, můžeme snímání zastavit tehdy, kdy jsme už provedli takové množství základních kroků.

```
MERGE (A, p, q, r)
1     n1 = q – p + 1
2     n2 = r – q
3     nechť L[1..n1 + 1] a R[1 ..n2 + 1] jsou novými poli
4     for i = 1 to n1
5         L[i] = A[p + i – 1]
6     for j = 1 to n2
7     R[j] = A[q + j]
8     L[n1 + 1] = ∞
9     R[n2 + 1] = ∞
10    i = 1
11    j = 1
12    for k = p to r
13        if L[i] ≤ R[j]
14        A[k] = L[i]
15        i = i + 1
16    else A[k] = R[j]
17        j = j + 1
```

Procedura MERGE funguje konkrétně podle tohoto pseudokódu. Na řádku 1 se vypočítá délka n_1 subpole $A[p..q]$, na řádku 2 se vypočítá délka n_2 subpole $A[q + 1..r]$. Na řádku 3 vytvoříme pole L a R („levé" a „pravé") o délkách $n_1 + 1$ a $n_2 + 1$; pozice navíc v každém poli bude mít uloženu *sentinel value* v každém poli. Cyklus for na řádcích 4-5 kopíruje subpole $A[p .. q]$ do pole $L[1..n_1]$ a cyklus **for** na řádcích 6-7 kopíruje subpole $A[q + 1 .. r]$ do pole $R[1.. n_2]$. Řádky 8-9 vkládají *sentinel value*s na konec polí L a R. Na řádcích 10-17, znázorněných v ilustraci 2.3,



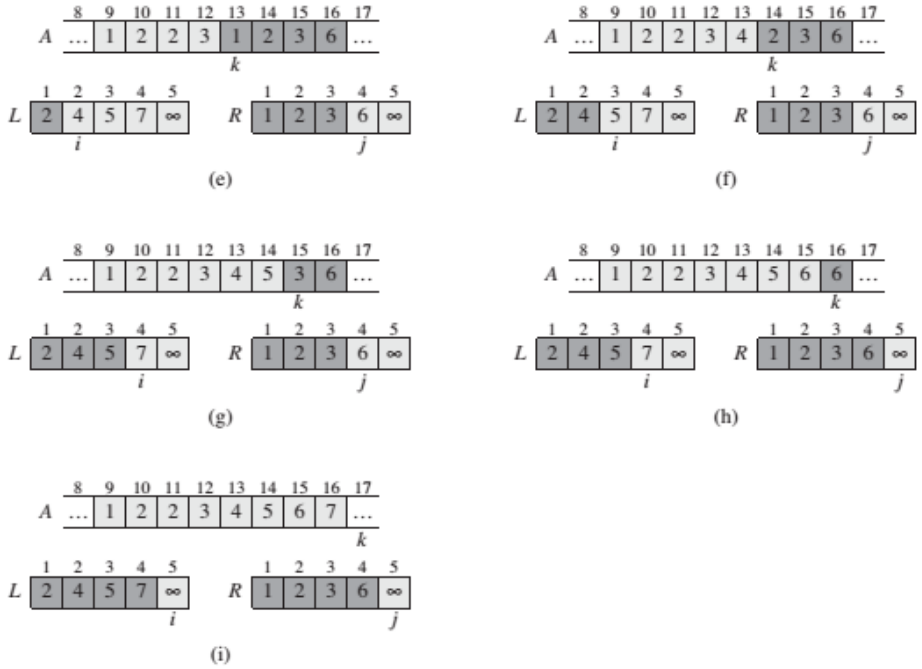
Ilustrace 2.3 Operace na řádcích 10-17 představuje volání MERGE (A, 9, 12, 16), kdy subpole [9.. 16] obsahuje posloupnost ⟨2, 4, 5, 7, 1, 2, 3, 6⟩. Po zkopírování a vložení *sentinel values* obsahuje pole L hodnoty ⟨2, 4, 5, 7, 1⟩ a pole R hodnoty ⟨1, 2, 3, 6, 1⟩. Světle šedé pozice v poli A obsahují konečné hodnoty a světle šedé pozice v polích L a R obsahují hodnoty, které nebyly ještě zkopírovány zpět do pole A. Celkově vzato, světle šedé pozice se vždy skládají z hodnot objevujících se původně v A[9.. 16] společně s dvěma *sentinel values*. Tmavě šedé pozice v poli A obsahují hodnoty, které budou přepsány, a tmavě šedé pozice polí L a R obsahují hodnoty, které byly již zkopírovány zpět do pole A. **(a)-(h)** znázorňují pole A, L a R a příslušné indexy k, i a j, které předcházejí každé interaci cyklu na řádcích 12-17.

se provádí r – p + 1 základních kroků udržováním následujícího invariantu cyklu:

Na začátku každé iterace cyklu **for** z řádků 12-17 obsahuje subpole $A[p.. k – 1]$ $k – p$ nejmenších prvků pole $L[1..n_1 + 1]$ a $R[1..n_2 + 1]$, a to v seřazeném pořadí. Kromě toho, $L[i]$ a$R[j]$ jsou nejmenšími prvky příslušných polí, která nebyla dosud zkopírována zpět do pole A.

Musíme ještě znázornit, že tento invariant cyklu se uchová ještě před první iterací cyklu **for** zapsaného na řádcích 12-17. Každá iterace cyklu udržuje invariant, který poskytuje užitečnou schopnost zobrazit správnost tehdy, když dojde k ukončení cyklu.

Inicializace: Před první iterací cyklu máme $k = p$, tzn. subpole $A[p \,..\, k-1]$ je prázdné. Toto prázdné subpole obsahuje $k - p = 0$ nejmenších prvků polí L a R a odtud $i = j = 1$, čili obě $L[i]$ a $R[j]$ jsou nejmenšími prvky polí, která nebyla dosud zkopírována zpět do pole A.



Ilustrace 2.3, pokračování (i) Pole a indexy při ukončení. V této chvíli je subpole v $A[9 \,..\, 16]$ seřazeno a dvě sentinel hodnoty v poli L a R jsou pouze dvěma elementy, které nebyly dosud zkopírovány do pole A.

Údržba: Abychom mohli vidět, že každá iterace udržuje invariant cyklu, předpokládejme, že platí $L[i] \leq R[j]$. Jakmile je tedy $L[i]$ nejmenším prvkem, který nebyl dosud překopírován do pole A. Protože $A[p \,..\, k-1]$ obsahuje $k - p$ nejmenších prvků, po řádku 14 se kopíruje $L[i]$ do $A[k]$ a subpole $A[p \,..\, k]$ bude obsahovat $k - p + 1$ nejmenších prvků. Inkrementací k (v aktualizaci cyklu **for**) a i (na řádku 15) se znovu vytváří invariant cyklu pro další iteraci.

Jestliže místo toho platí $L[i] > R[j]$, je pak provedení řádků 16 – 17 vhodnou činností pro údržbu invariantu cyklu.

Ukončení: Při ukončení platí, že $k = r + 1$. Díky invariantu cyklu subpole $A[p .. k - 1]$, které je $A[p .. r]$, obsahuje k – p = $r - p + 1$ nejmenších prvků pole $L[1 .. n_1 + 1]$ a pole $R[1 .. n_2 + 1]$, a to v seřazeném pořadí. Pole L a R budou společně obsahovat $n_1 + n_2 + 2 = r - p + 3$ prvků. Ne všechny, ale jen tyto dvě největší pole byla zkopírována zpět do A a tyto dva největší pvrky jsou *sentinel values*.

CONCLUSION

The theoretical part of the bachelor thesis provided a distinction between the aforementioned types of translation and equivalence. The scientific style of writing has helped to understand the features that should be considered during a translating process. Afterwards, the analysis of the text emphasized the key attributes of the text and contributed to better understanding of settings used thereof. The translated text followed the analysis with a focus on creating a true and readable copy.

The translation of the text revealed several findings examined in the analysis. Firstly, the translating process showed that the translator should be well educated in computer science and mathematics in order to preserve the precise meaning. Secondly, this kind of text confirmed that a text can be understood more easily with some corresponding illustrations, however, it seemed difficult to omit some summarizing or repeating parts in order to sustain a coherent text. The translated text also presented several options for translating certain parts, though a discussion on the choice from these options was necessary.

To sum up, the translated text also posed a question whether a translation should contain some explanative footnotes because it transpired that expectations regarding the education of the source and target audiences may differ.

BIBLIOGRAPHY

Baker, Mona. 2011. *In Other Words: A Coursebook on Translation*. Oxon: Routledge.

Byrne, Jody. 2010. *Technical Translation: Usability Strategies for Translating Technical Documentation*. Softcover ed. publication place: Springer.

Cormen, Thomas H. 2009. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press.

Fawcett, Peter D. 1997. *Translation Theories Explained*. Vol. 3, *Translation and Language: Linguistic Theories Explained*. Manchester, U.K.: St. Jerome.

Hatim, B, and Jeremy Munday. 2004. *Translation: An Advanced Resource Book*. Routledge Applied Linguistics. London: Routledge.

House, Juliane. 2009. *Translation*. Oxford: Oxford University Press.

Jakobson, Roman. 1959. *On linguistic aspects of translation*. In Brower, R. A. *On Translation*. Cambridge: Harvard University Press.

Knittlová, Dagmar, Bronislava Grygová, and Jitka Zehnalová. 2010. *Překlad a překládání*. Olomouc: Univerzita Palackého v Olomouci, Filozofická fakulta.

Knittlová, Dagmar. 2000. *K Teorii i Praxi Překladu*. 2. vyd. ed. Olomouc: Univerzita Palackého.

Munday, J. 2001. *Introducing translation studies: theories and applications*. 1st pub. London: Routledge.

Neubert, Albrecht, and Gregory M. Shreve. 1992. *Translation Studies*. Vol. 1, *Translation as Text*. Kent, Ohio: Kent State University Press.

Nord, Christiane. 2005. *Text Analysis in Translation: Theory, Methodology and Didactic Application of a Model for Translation-Oriented Text Analysis*. 2nd ed. Amsterdam: Rodopi.

Newmark, Peter. 1988. *A Textbook of Translation*. New York: Prentice-Hall International.

Rubens, Philip. 2001. *Science and Technical Writing: A Manual of Style*. 2nd ed. New York: Routledge.

Valla, Tomáš. 2007. Recepty z Programátorské Kuchařky Třídění Korespondenční seminář z programování July 18. Accessed May 3, 2015. http://ksp.mff.cuni.cz/tasks/16/cook2.html.

Wright, Sue Ellen, and Leland D. Wright, eds. 1993. *American Translators Association Scholarly Monograph Series,*. Vol. 6, *Scientific and Technical Translation*. Amsterdam: John Benjamins Pub. Co.

APPENDICES

APPENDIX P I: TERMINOLOGICAL GLOSSARY

Insertion sort	[in'sə:šn so:t]	Řazení vkládáním
Running time	[raniŋ taim]	Doba běhu
Merge sort	[mə:dž so:t]	Seřazení sléváním
Sequence	['si:kwəns]	Posloupnost (čísel / prvků)
Pseudocode	[ˈs(j)uːdəʊˌkəʊd]	Pseudokód
Element	[eliment]	Prvek
Array	[ə'rei]	Pole (obsahující prvky)
Subarray	[sʌbə'rei]	Subpole
Loop	[lu:p]	Cyklus
Iteration	[ˌitə'reišn]	Iterace, opakování
Inductive step	[in'daktiv step]	Indukční krok
Loop invariant	[lu:p ɪnˈvɛːrɪənt]	Invariant cyklu
Increment	[inkrəmənt]	Zvýšit
Decrement	[ˈdɛkrɪm(ə)nt]	Snížit
Assignment	[ə'sainmənt]	Přiřazení (hodnoty, parametru)
Loop counter	[lu:p kauntə]	Čítač
Loop header	[lu:p hedə]	Hlavička cyklu
Parenthesize	[pəˈrɛnθɪsʌɪz]	Uzávorkovat
Caller	[ko:lə]	Volající funkce
Calling procedure	[ko:liŋ prə'si:džə]	Volající procedura
Nil	[nil]	Nula
Integer	[intidžə]	Celé číslo
Vertex	[və:teks]	Vrchol (grafu)
Edge	[edž]	Hrana (grafu)
Cost statement	[kost steitmənt]	Cena výroku
Order of growth	[o:də ov grəuθ]	Míra růstu
Term	[tə:m]	Term / člen
Formula	[fo:mjulə]	Formule, vzorec, rovnice
Short circuiting	[šo:t ˈsə:kɪtiŋ]	Líné vyhodnocování
Linear search	[liniə sə:č]	Linearní prohledávání
Floor	[flo:]	Zaokrouhlení dolů
Ceiling	[si:liŋ]	Zaokrouhlení nahoru

Floating point	[fləutiŋ]	Pohyblivá řádová čárka
Subroutine call	[ˈsʌbru:ti:n ko:l]	Volání podprogramu
Unconditional branch	[ˌankən'dišənl bra:nč]	Nepodmíněný skok / nepodmíněné větvení
Computational time	[ˌkompju'teišənl taim]	Výpočetní čas
Exponentiation	[ˌɛkspənɛnʃɪˈeɪʃ(ə)n]	Umocnění
Probabilistic analysis	[ˌprɒbəbɪˈlɪstɪk ə'næləsis]	Analýza pravděpodobnosti
Sentinel value	[sentinl vælju:]	Limitní hodnota

APPENDIX P II: ORIGINAL OF TRANSLATED TEXT

2 Getting Started

This chapter will familiarize you with the framework we shall use throughout the book to think about the design and analysis of algorithms. It is self-contained, but it does include several references to material that we introduce in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We define a "pseudocode" that should be familiar to you if you have done computer programming, and we use it to show how we shall specify our algorithms. Having specified the insertion sort algorithm, we then argue that it correctly sorts, and we analyze its running time. The analysis introduces a notation that focuses on how that time increases with the number of items to be sorted. Following our discussion of insertion sort, we introduce the divide-and-conquer approach to the design of algorithms and use it to develop an algorithm called merge sort. We end with an analysis of merge sort's running time.

2.1 Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers that we wish to sort are also known as the ***keys***. Although conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.

In this book, we shall typically describe algorithms as programs written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python, or Pascal. If you have been introduced to any of these languages, you should have little trouble

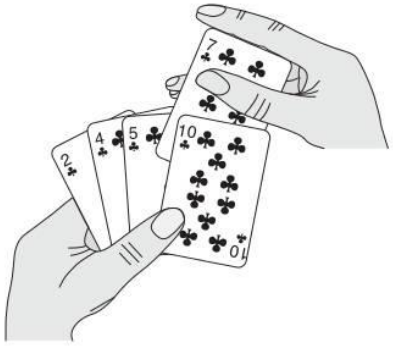


Figure 2.1 Sorting a hand of cards using insertion sort.

reading our algorithms. What separates pseudocode from "real" code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of "real" code. Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

We present our pseudocode for insertion sort as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1..n]$ containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by $A.length$.) The algorithm sorts the input numbers ***in place***: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.

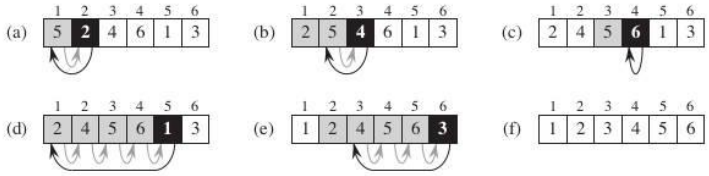


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by j, the subarray consisting of elements $A[1 \,..\, j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \,..\, n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \,..\, j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \,..\, j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \,..\, j-1]$ consists of the elements originally in $A[1 \,..\, j-1]$, but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. (Of course, we are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property is perhaps the most important one, since we are using the loop invariant to show correctness. Typically, we use the loop invariant along with the condition that caused the loop to terminate. The termination property differs from how we usually use mathematical induction, in which we apply the inductive step infinitely; here, we stop the "induction" when the loop terminates.

Let us see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \,..\, j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \,..\, j]$ then consists of the elements originally in $A[1 \,..\, j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. At this point, however,

[1] When the loop is a **for** loop, the moment at which we check the loop invariant just prior to the first iteration is immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable j but before the first test of whether $j \leq A.length$.

we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order. Observing that the subarray $A[1 \,..\, n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if-else** statements[2] as well. Using indentation instead of conventional indicators of block structure, such as **begin** and **end** statements, greatly reduces clutter while preserving, or even enhancing, clarity.[3]
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.[4] In this book, the loop counter retains its value after exiting the loop, unlike some situations that arise in C++, Java, and Pascal. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound. We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $j = 2$ **to** $A.length$, and so when this loop terminates, $j = A.length + 1$ (or, equivalently, $j = n + 1$, since $n = A.length$). We use the keyword **to** when a **for** loop increments its loop

---

[2] In an **if-else** statement, we indent **else** at the same level as its matching **if**. Although we omit the keyword **then**, we occasionally refer to the portion executed when the test following **if** is true as a ***then clause***. For multiway tests, we use **elseif** for tests after the first one.

[3] Each pseudocode procedure in this book appears on one page so that you will not have to discern levels of indentation in code that is split across pages.

[4] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat-until** loops, and its **for** loops operate a little differently from the **for** loops in this book.

counter in each iteration, and we use the keyword **downto** when a **for** loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.

- The symbol "//" indicates that the remainder of the line is a comment.
- A multiple assignment of the form $i = j = e$ assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment $j = e$ followed by the assignment $i = j$.
- Variables (such as i, j, and *key*) are local to the given procedure. We shall not use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the ith element of the array A. The notation ".." is used to indicate a range of values within an array. Thus, $A[1..j]$ indicates the subarray of A consisting of the j elements $A[1], A[2], \ldots, A[j]$.
- We typically organize compound data into ***objects***, which are composed of ***attributes***. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, we treat an array as an object with the attribute *length* indicating how many elements it contains. To specify the number of elements in an array A, we write $A.length$.

  We treat a variable representing an array or object as a pointer to the data representing the array or object. For all attributes f of an object x, setting $y = x$ causes $y.f$ to equal $x.f$. Moreover, if we now set $x.f = 3$, then afterward not only does $x.f$ equal 3, but $y.f$ equals 3 as well. In other words, x and y point to the same object after the assignment $y = x$.

  Our attribute notation can "cascade." For example, suppose that the attribute f is itself a pointer to some type of object that has an attribute g. Then the notation $x.f.g$ is implicitly parenthesized as $(x.f).g$. In other words, if we had assigned $y = x.f$, then $x.f.g$ is the same as $y.g$.

  Sometimes, a pointer will refer to no object at all. In this case, we give it the special value NIL.
- We pass parameters to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not. For example, if x is a parameter of a called procedure, the assignment $x = y$ within the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible. Similarly, arrays are passed by pointer, so that

a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure.

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement.
- The boolean operators "and" and "or" are ***short circuiting***. That is, when we evaluate the expression "x and y" we first evaluate x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and so we do not evaluate y. If, on the other hand, x evaluates to TRUE, we must evaluate y to determine the value of the entire expression. Similarly, in the expression "x or y" we evaluate the expression y only if x evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $x.f = y$" without worrying about what happens when we try to evaluate $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called. The calling procedure is responsible for handling the error, and so we do not specify what action to take.

Exercises

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

2.1-3

Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.

Output: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-4

Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in

an $(n + 1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine (RAM)*** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no; it takes several instructions to compute x^y when x and y are real numbers. In restricted situations, however, exponentiation is a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by k positions to the left. In most computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2, so that shifting the bits by k positions to the left is equivalent to multiplication by 2^k. Therefore, such computers can compute 2^k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word. We will endeavor to avoid such gray areas in the RAM model, but we will treat computation of 2^k as a constant-time operation when k is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory. Several computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. A handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book will not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Even though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way that is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time c_i, where c_i is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[5]

In the following discussion, our expression for the running time of INSERTION-SORT will evolve from a messy formula that uses all the statement costs c_i to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n = A.length$, we let t_j denote the number of times the **while** loop test in line 5 is executed for that value of j. When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

---

[5]There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, later in this book we might say "sort the points by x-coordinate," which, as we shall see, takes more than a constant amount of time. Also, note that a statement that calls a subroutine takes constant time, though the subroutine, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) \\ &\quad + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)\ . \end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \le key$ in line 5 when i has its initial value of $j-1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

[6] This characteristic does not necessarily hold for a resource such as memory. A statement that references m words of memory and is executed n times does not necessarily reference mn distinct words of memory.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(see Appendix A for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. We give three reasons for this orientation.

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.

- The "average case" is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray $A[1 \,..\, j-1]$ to insert element $A[j]$? On average, half the elements in $A[1 \,..\, j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1 \,..\, j-1]$, and so t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the ***average-case*** running time of an algorithm; we shall see the technique of ***probabilistic analysis*** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a ***randomized algorithm***, which makes random choices, to allow a probabilistic analysis and yield an ***expected*** running time. We explore randomized algorithms more in Chapter 5 and in several other subsequent chapters.

Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure. First, we ignored the actual cost of each statement, using the constants c_i to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

We shall now make one more simplifying abstraction: it is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared"). We shall use Θ-notation informally in this chapter, and we will define it precisely in Chapter 3.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower

order of growth. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.

Exercises

2.2-1
Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2
Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n - 1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

2.2-3
Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

2.2-4
How can we modify almost any algorithm to have a good best-case running time?

2.3 Designing algorithms

We can choose from a wide range of algorithm design techniques. For insertion sort, we used an ***incremental*** approach: having sorted the subarray $A[1 \,..\, j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1 \,..\, j]$.

In this section, we examine an alternative design approach, known as "divide-and-conquer," which we shall explore in more detail in Chapter 4. We'll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that we will see in Chapter 4.

2.3.1 The divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \le q < r$. The procedure assumes that the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p\,..\,r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto

the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \,..\, q]$, and line 2 computes the length n_2 of the subarray $A[q + 1 \,..\, r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray $A[p \,..\, q]$ into $L[1 \,..\, n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q + 1 \,..\, r]$ into $R[1 \,..\, n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illus-

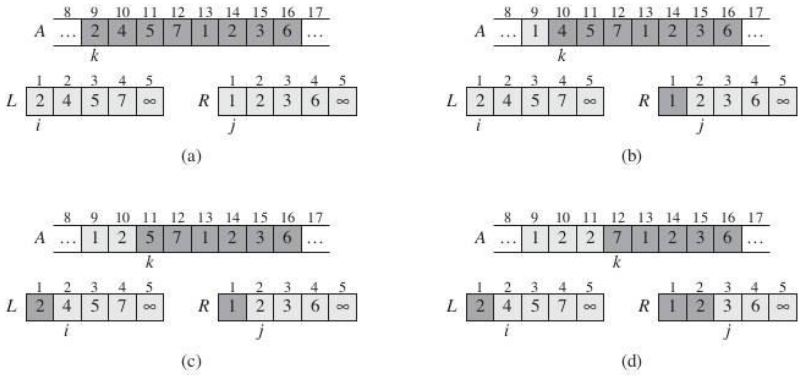


Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty\rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.

trated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

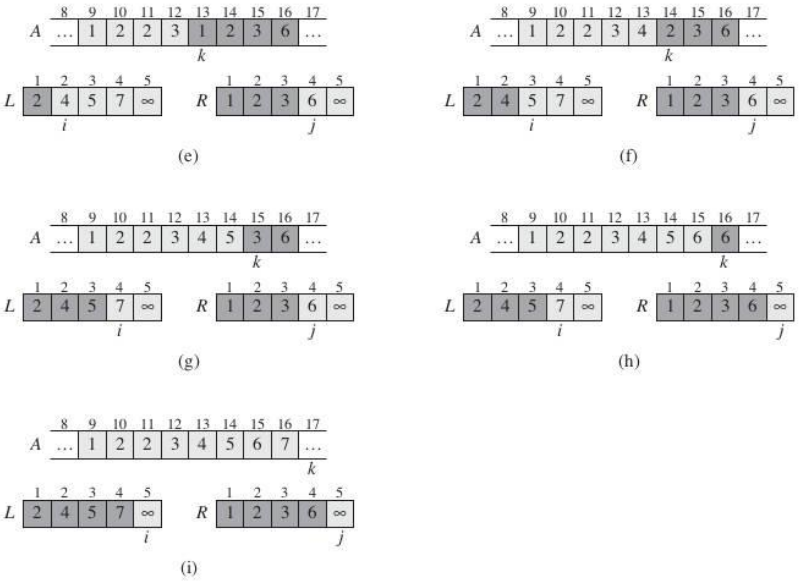


Figure 2.3, continued **(i)** The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r+1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k-p = r-p+1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.