A Companion to Data Organization

Aleksandar Ignjatović

Figure 1: A medieval copy of arguably the most influential textbook ever written, Euclid's *Elements* (Στοιχεία), from about 300 BC. It contains the oldest known algorithm recorded, Euclid's algorithm for computing the Greatest Common Divisor (GCD) of two natural numbers.

Figure 2: First data structures: a Mesopotamian cuneiform tablet from about 3000 BC with an inventory list.

Preface

This booklet is not meant to be a textbook, but only a companion to the textbook that we use in this course, Cormen, Leiserson, Rivest and Stein's *Introduction to Algorithms*, the second edition. Its purpose is to help you read the appropriate sections of the textbook, and integrate it better with the rest of the course. It also adds a number of exercises not present in the textbook. **Please read the textbook and do not rely on this supplement only.**

Contents

Chapter 1

Introduction

1.1 What is an Algorithm?

An algorithm is a collection of precisely defined steps that are executable using certain specified mechanical methods. By "mechanical" we mean the methods that do not involve any creativity, intuition or even intelligence. Thus, algorithms are specified by detailed, easily repeatable "recipes". The word "algorithm" comes by corruption of the name of *Mukhammad ibn Musa Al-Khorezmi* (780-850) who wrote an important book on algebra, *"Kitab al muhtasar fi hisab al gabr w'al muqubalah"* ("A compact introduction to calculation using rules of completion and reduction"). It is also believed that the title of this book is the source of the word "algebra".

In this course we will deal only with algorithms that are given as sequences of steps, thus assuming that only one step can be executed at a time. Such algorithms are called *sequential*. If the action of each step gives the same result whenever this step is executed for the same input, we call such algorithms *deterministic*. If the execution involves some random process, say throwing some (possibly software simulated) coin

or dice, such algorithms are called *randomized algorithms*. Randomized algorithms are important tools for tasks that have to be repeated many times.

Example 1.1 A *sorting algorithm* for an input sequence of numbers $\langle a_1, a_2, , a_n \rangle$ produces as output a permutation $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ of the input sequence such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$.

There are many algorithms that accomplish the task of sorting that have different efficiency for different kinds of inputs. Efficiency of an algorithm is measured by its *time complexity*, and to *analyze* an algorithm means to determine its time complexity for the given type of inputs. Usually we do the *worse case analysis*, i.e., we determine the largest number of basic steps needed to execute the algorithm for any input of size n. Also important is the *average case* analysis. It provides the expected number of basic steps needed to execute the algorithm for inputs of length at most n, assuming certain probability distribution of these inputs. Thus, if we assume that all inputs of size at most n are equally likely, the average case analysis gives the average number of steps needed to execute the algorithm for all inputs of size at most n. Clearly, a time complexity function $T(n)$ is non decreasing, i.e., $m \leq n \Rightarrow T(m) \leq T(n)$. An algorithm is *optimal* if it provides the necessary functionality with the smallest possible numbers of steps for each input.

Another important feature of an algorithm is its *correctness* i.e., that it provides correct functionality for all possible inputs. Generally, the only way to ensure the correctness of an algorithm is by means of a mathematical proof. This is due to the fact that it is impossible to test an algorithm for all possible inputs. Sometimes such correctness proofs are very simple, because the correctness follows immediately from the

definition of the algorithm. However, sometimes the correctness of an algorithm is far from obvious and requires a meticulous proof.

The activity of designing an algorithm and an appropriate data structure for the problem at hand is called *problem solving*. Clearly, problem solving precedes writing a code, and in fact it is largely independent of the programming language or even of the programming paradigm used (e.g., object oriented, imperative, etc). This is true for as long as the algorithm is to be implemented on a standard, sequential processor. After problem solving is completed and the correctness of designed algorithms is verified, there is still much to do, namely one has to *implement* the algorithms and the associated data structures in a specific programming language. Such implementation has to meet numerous important criteria:

- ***reliability*** - the implementation is correct and it will not introduce vulnerabilities to the system that runs it.

- ***maintainability*** - one can easily replace a part of the code in order to provide somewhat different functionality;

- ***extensibility*** - to obtain more features, additional piece of code can be integrated with the existing code with little or no modifications of the already existing code needed;

- ***reusability*** - possibility of using parts of the code in another application. This is accomplished by *modular design*.

- ***robustness*** - the implementation does not crush if the user inputs an unexpected or illegitimate input;

- ***user friendliness*** - users can interact with ease with the program during its execution;

- ***implementation time*** - implementation takes a reasonable amount of programming time and other resources.

Note that these features are not features of algorithms and data structures, but rather, of their implementations. Clearly, it is equally important that both the algorithms and the data structures on one hand, as well as their implementation on the other hand, be correct. Separation of design and implementation simplifies the process of software development and helps ensure reliability of the final product, because it allows that the correctness of algorithms be verified prior to their their implementations.

1.2 Abstraction Levels in Data Organization

While solving a problem that involves a collection of objects of any kind, only a limited number of properties of these objects and a limited number of their mutual relations are relevant for the problem considered. Thus, objects can be "abstracted away", i.e., replaced with suitable "names" and with an abstract representation of their corresponding properties and mutual relations. These properties of objects and their relations we call *data*.

Data is organized in a well-defined and structured abstract entity, called a *data structure*. Note that this makes sense regardless of whether data is to be manipulated by a human or by a machine. The data corresponding to each object usually consists of a *key* that identifies the object that data is abstracted from, and the remaining data that is called the *satellite data*. Together they form a *record*, which is an element of the data

structure. Data structures can change their content as elements are added or deleted, and for that reason a data structure is a *dynamic set*.

Example 1.2 A group of students as a "set of objects", with data for each student consisting of their names, used as the corresponding key, and their date of birth, address and student number, organized in a data structure in the form of an alphabetical list.

1.2.1 Abstract Data Structures

We can further ignore what particular data is involved, and consider only the abstract structure into which data is organized for the purpose of efficient storage and retrieval, e.g., a linked list, a binary search tree, etc. In this way, we obtain an *abstract data structure (ADS)*. An ADS is defined by the details of its internal organization and by internal operations that are used to maintain its properties as elements are added or removed. Information about what kind of data will be stored in such structures is not a part of abstract data structure specification. This has an important practical consequence that the same abstract data structure can be used to store different type of data, say integers or floating point numbers, by simply instantiating the same data structure with a different particular *type* of data.

Examples of Abstract Data Structures

1.1 A Singly Linked List is an Abstract Data Structure organized in a list of elements that has a special beginning element (the *head*) and an end element (the *tail*). Further, each element contains information what the next element in the list is.

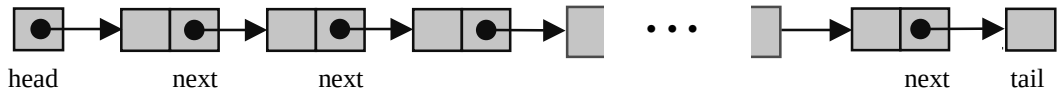


Figure 1.1: A singly linked list

1.2 A Doubly Linked List satisfies all properties of a singly linked list, but also contains with each element information what are its predecessor and its successor.

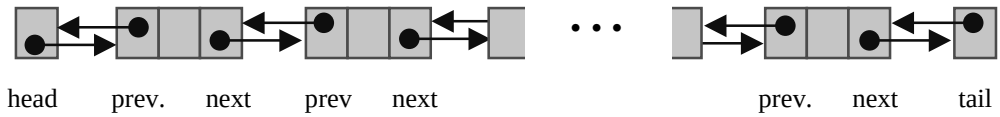


Figure 1.2: A doubly linked list

1.3 An array allows a direct and thus fast (constant time) access to each of its elements, but to insert a new element into the array while preserving the order among the existing elements, we may have to move a lot of elements to free the appropriate space, as shown on Figure 1.3.

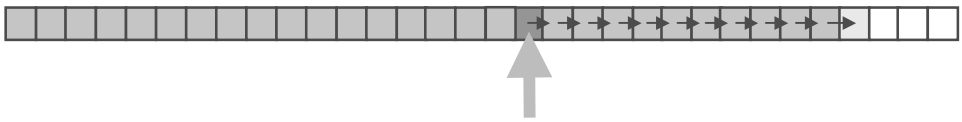


Figure 1.3: Insertion in an array

In a linked list access to a particular element may require traversing many elements in the list, but once the appropriate insertion point has been reached, insertion and deletion involve changing only a few references (pointers), as shown on Figure 1.4.

This difference should be kept in mind when deciding if a list of elements should be represented using an array or a linked list.

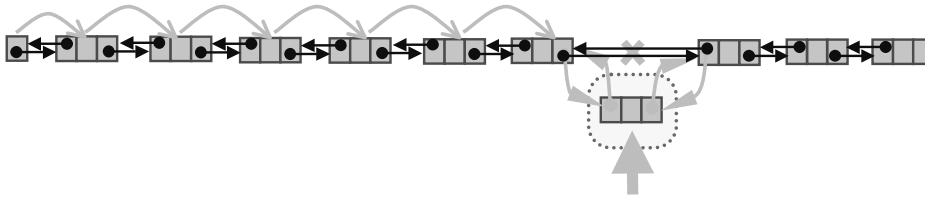


Figure 1.4: Insertion in a doubly linked list

Exercise 1.1 We say that a linked list contains a loop if one of its elements points back to one of the previous elements in the list; see Figure 1.5.

1. Write a code fragment that, given any two integers m,n such that $0 \leq n < m$ it produces a singly linked list of total length m and with a loop of length n. (Note that n can be zero.) The content of each link is 0.

2. Write a code fragment that, given a singly linked list, say produced by the code obtained in the first part of this exercise, it determines whether the list contains a loop or not. The program is not allowed to make any modifications to the list, and it can use only constant amount of additional storage space, i.e., an amount of additional space that does not depend on the size of the list. ***Hint:*** *Try advancing two pointers, one twice as fast as the other.*

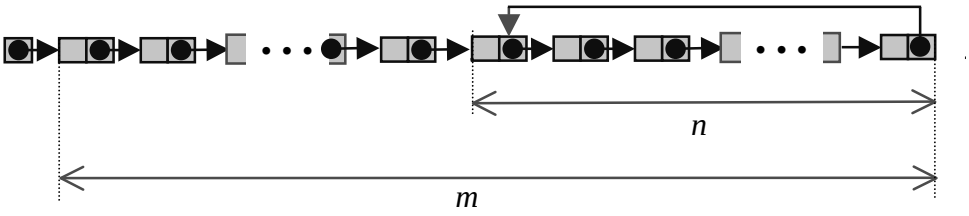


Figure 1.5: A singly linked list with a loop

3. Let A be an automaton that has a memory of 4 bits only, 3 bit wide input and output ports on two of its sides, an input for the synchronization clock pulses, and a single bit special output, as shown on Figure 1.6.

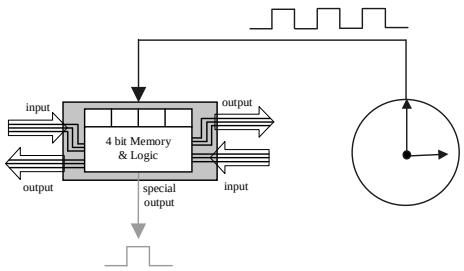


Figure 1.6: A simple finite automaton

Consider now a very long chain consisting of 2^k such automata (k is very large), shown on Figure 1.7, such that output ports of each automaton are directly connected to the input ports of the next and the previous automaton. At every clock cycle, each automaton can read the input port values, perform a computation and set the values of the output port that become available to the previous and the next automata on the next clock cycle. Thus, it takes at least n clock cycles to send a message from the first to the last automaton. Design an algorithm that

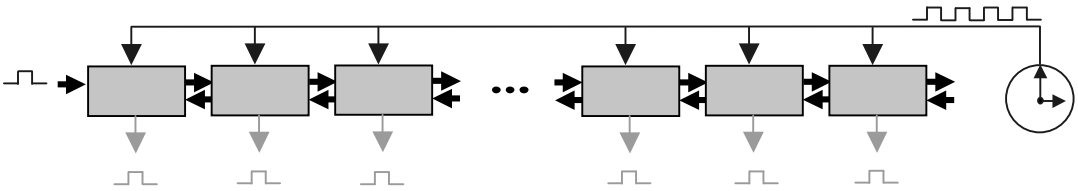


Figure 1.7: A chain of finite automata

should be implemented by such automata to allow them to function in the following way. After a triggering pulse arrives at the input port of the first automaton, all automata should output 1 at their special outputs *simultaneously* (after the necessary number of clock cycles), and then reset and wait for the next triggering pulse. Notice that, given extreme memory limitations of these finite automata, they cannot determine their position in the chain. ***Hint:*** *You might find the hint for 2. above inspiring!*

1.4 In a **Binary Search Tree (BST)** the key at every node is equal or larger than all keys in the left subtree at this node, and equal or smaller than all keys at the right subtree at that node; see Figure 1.8. Such tree itself is **not** data and caries no information, but it is a construct used to organize data efficiently. The time complexity of a search for a particular key in a binary search tree is bounded by the height of such a tree, rather than by its size. This is important, because, as we will see, the size of a well balanced tree is exponential in its hight, and there are methods of ensuring a good balance as elements are added or deleted. Note that a tree is *well balanced* if at each of its nodes the height of the left subtree is about the same as the height of the right subtree; we will explain the details later.

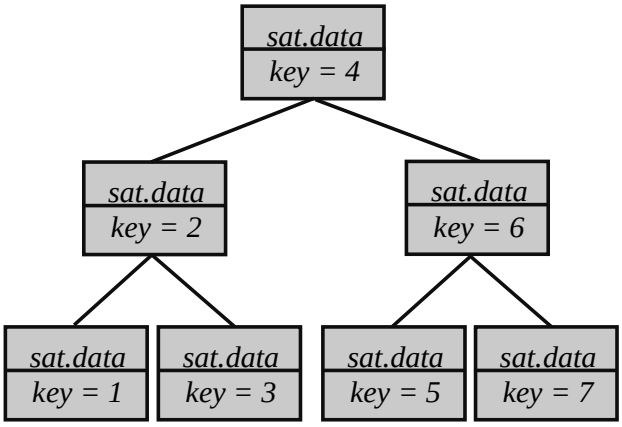
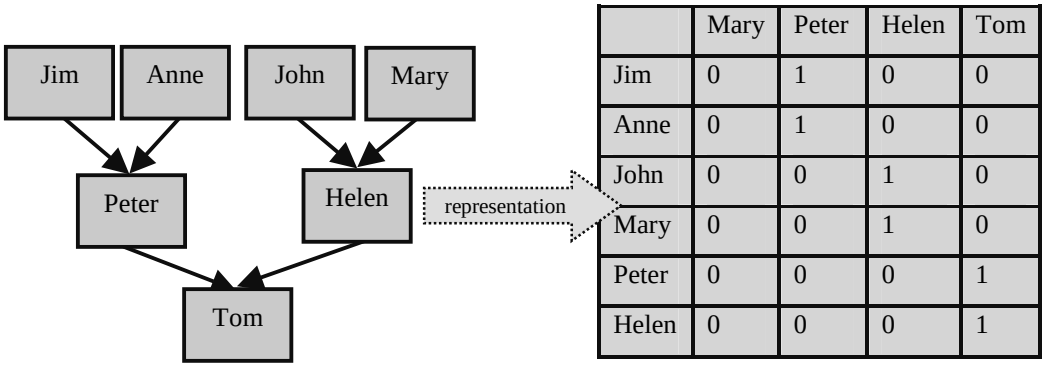


Figure 1.8: A binary search tree

If we have a collection of elements with a mutual relationship that naturally forms a tree, then we must treat such a tree structure as a part of data. For example, members of an extended family with "*x is a parent of y*" relationship form a genealogical tree. However, an abstract data structure that can be used to represent such a genealogical tree way might not itself be a tree but, for example, a table instead, as shown on the Figure 1.9.



	Mary	Peter	Helen	Tom
Jim	0	1	0	0
Anne	0	1	0	0
John	0	0	1	0
Mary	0	0	1	0
Peter	0	0	0	1
Helen	0	0	0	1

Figure 1.9: Representation of a genealogical tree

Consequently, we must be careful to distinguish between structures that are inherent in the collection of objects we consider and are thus part of data, from the auxiliary structures introduced in the process of data organization whose purpose is only to allow efficient storage and retrieval of data.

1.2.2 Abstract Data Types

If we further ignore the particular internal organization of a data structure, and consider only its *functionality*, then we obtain an *abstract data type (ADT)*. Functionality of an ADT is provided by a set of operations forming the *interface* of the ADT. This is the only aspect of an ADT visible and accessible from the outside of an ADT. We say that these operations are *exported* by the ADT. The program that utilizes an ADT via its interface operations is usually called the *client*.

An ADT is *realized* by an appropriate abstract data structure that is in turn *implemented* as a construct in a particular programming language. For example, the *Stack* ADT can be realized by a Linked List ADS, which can be implemented as a pointer based construct in, say, C. This is illustrated on the Figure 1.10.

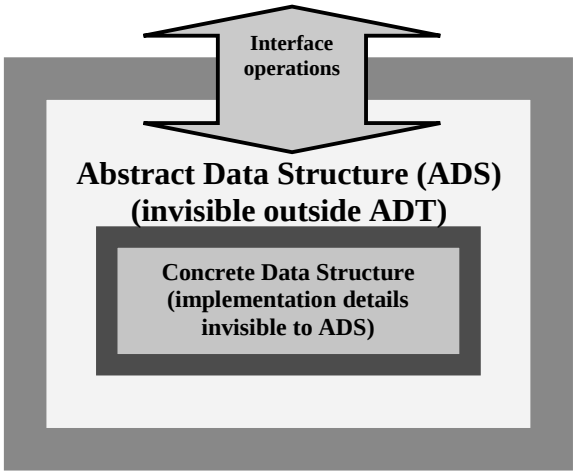


Figure 1.10: Abstract Data Type

Examples of Abstract Data Types

1.5 *Stack ADT* has interface consisting of operations `push(x), pop(), size(), top(), makeEmpty()` and `isEmpty()`. Notice that the stack itself is a "hidden variable" of these operations; only push has an explicit input. Insertions and deletions from the stack follow the *last-in-first-out scheme (LIFO)*. Its functionality is described by the following proerties of its operations:

- `push(x)` takes an element x as input and inserts it at the top of the stack; since the stack itself is hidden, this operation produces no output;

- `pop()` removes and returns the top element of the stack as the output; if the stack is empty, it returns an error message;

- `top(x)` returns the top element of the stack as the output without removing it; if the stack is empty, it returns an error message;

- `size()` returns the number of elements in the stack;

- `makeEmpty()` makes the stack empty;
- `isEmpty()` returns `True` just in case the stack is empty.

Note that in the above "programming language type" of notation, the stack itself is hidden, because the interface functions hide "the inside" of the abstract data structure. To formulate the above properties in a rigorous way, we make the hidden "stack" variable explicit and use for it a variable of separate kind, denoted by a Greek letter. Accordingly, the output of operations also have two components: the "visible", exported value given by the first element of the ordered pair, and the "invisible" new stack content given by the second element of the ordered pair.

1. `pop(push(x,a))=(x,a)`. Thus, `pop` returns the top element of the stack `a`, but it also modifies the stack by removing the top element; consequently, the output is the pair consisting of the top element `x` and the stack `a`.
2. `top(push(x,a))=(x, push(x,a))`. Thus, `top` returns the top element of the stack `a`, without removing it; the output is the top element `x` and the stack `push(x,a)`.
3. `isEmpty(a)= True` $\Rightarrow$ `size(a)= 0; size(push(x,a))=size(a)+1`
4. `isEmpty(a)= True` $\Rightarrow$ `pop(a)= top(a)= error"EmptyStack"`
5. `isEmpty(makeEmpty(a))= True`

The functionality of the queue is thus defined purely algebraically. This highlights the fact that an ADT is defined only by its functionality, completely hiding how such functionality is achieved. There are important practical benefits of such an approach:

1. It eliminates the need to maintain uniform data representation throughout the system: for as long as we know how interface operations input and output data, inside an ADT data can be represented in an arbitrary way, different from the way how the same data is represented in the client program. Thus, changes in the design of the client or the ADT raise no question regarding their subsequent compatibility.

2. Design becomes modular; different modules can be designed by different people, who need to know nothing about how other modules are designed, but only what the interface operations are and what the functionality provided by the ADT is.

3. The structure of the system can be described on a higher level, without going into the details of implementation. This greatly simplifies design and understanding of complex systems. It also simplifies testing and verification of systems, because each module can be tested separately prior to its incorporation into the system.

4. Modules can be easily reused in other systems without change for as long as the functionality needed is the same.

Examples of stack applications include: history of pages visited in a Web browser; undo sequence in a text editor; CPU stack that handles recursion; etc.

Exercise 1.2

1. Consider algebraic expressions built using the following rules:

   (a) a variable denoted by a small Roman letter is an expression. For example, x and y are both expressions;

   (b) if t_1 and t_2 are expressions, then so is $(t_1 t_2)$;

   (c) expressions can be built only using the above two rules.

Thus, for example $((xy)z)w)$ is a correct expression (we do not remove the outermost parentheses), but, for example, $(xy)(z$ is not a correct expression. Similarly, $((xy))$ is not a correct expression, because it cannot be obtained using the above rules. Design an algorithm that checks if a sequence of variables and brackets is a correct algebraic expression or not, i.e., if it has correctly matched parentheses. Implement your procedure and test it. ***Hint:*** *Set a counter and increment it whenever you open a bracket and decrement it when you close it. Ignore the variables.*

The other most commonly used ADT is the ***queue***. Unlike the stack that follows the LIFO scheme, insertions and deletions from the queue follow the *first-in-first-out scheme (LIFO)* as shown on the Figure below.

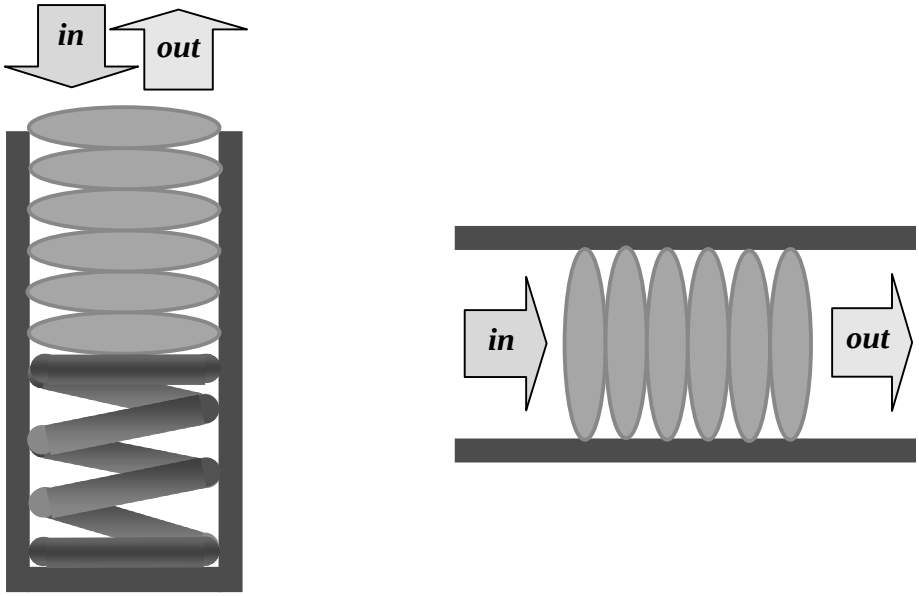


Figure 1.11: A Stack and a Queue

1.6 *Queue ADT* has interface consisting of operations `enqueue(x)`, `dequeue()`, `front()`, `size()`, `makeEmpty()` and `isEmpty()`. Again, the stack itself is

a "hidden variable" of these operations; only `push` has an explicit input. Since insertions and deletions from the queue follow the *first-in-first-out scheme (FIFO)*, a queue encodes temporal order how elements have been added. Its functionality is specified by the following properties of its operations:

- `enqueue(x)` takes an element `x` as input and inserts it at the end of the queue; since the queue itself is hidden, this operation produces no output;
- `dequeue()` removes and returns the first element in the queue as the output; if the queue is empty, it returns an error message;
- `front()` returns the first element in the queue as the output; if the queue is empty, it returns an error message;
- `size()` returns the number of elements in the queue;
- `makeEmpty()` makes the queue empty;
- `isEmpty()` returns `True` if the queue is empty.

An example of the Queue ADT is the printing queue: jobs are printed in the order they arrive at the printer. As with the stack, if we make the queue itself visible by using a variable of a separate kind, we can define the functionality of a queue purely algebraically:

1. `isEmpty(a)=True` $\Rightarrow$ `decueue(a)=front(a)=error"EmptyQueue"`
2. `isEmpty(a)=True` $\Rightarrow$ `decueue(enqueue(x,a))=(x,a)`
3. `isEmpty(a)=True` $\Rightarrow$ `front((enqueue(x,a)))=(x,enqueue(x,a))`

4. `isEmpty(a)=False` $\Rightarrow$ `decueue(enqueue(x,a))=decueue(a)`

5. `isEmpty(a)=False` $\Rightarrow$ `front(enqueue(x,a))=front(a)`

6. `isEmpty(a)=True` $\Rightarrow$ `size(a)=0`

7. `size(enqueue(x,a))=size(a)+1`

8. `isEmpty(makeEmpty(a))=True`

Exercise 1.3 A simple implementation of a queue would consist of an array together with two pointers that keep track of where the front and the rear end of the queue are. However, with such a design, as elements are added and removed, the content of the queue would move through the array and would eventually "run away" (overflow the array) even if there are fewer elements in the queue than the size of the array. Solve this problem by designing a circular structure. Implement it and test it.

1.7 *Binary Search Tree (BST)* is an ADT for efficient storage, removal and retrieval of records `(k,a)` with keys that are linearly ordered. There are several versions of BST; we chose one with the following interface operations:

- `insert((k,x))` inserts element `(k,x)` into the BST; if there is already an element `(k,y)` with the same key `k`, it is replaced with the new element;
- `delete(k)` deletes the element with the key `k` from the BST; if there is no such element, it leaves the tree unchanged.
- `find(k)` finds and outputs the element with key `k` if there is such element in the BST and it returns an error message if there is no such element;
- `in(k)` returns True just in case the BST contains an element with the key `k`.

- `size()` returns the number of elements in the BST;

- `isEmpty()` returns True if the BST is empty.

- `makeEmpty()` makes the BST empty.

- We require that all of the above operations run in time bounded by $K \lg(n+2)$ where K is a fixed constant, n is the number of elements in the tree, and $\lg n = log_2 n$.

Thus, a purely algebraic definition of a Binary Search Tree ADT is expected to be more complicated than the definitions of a stack or a queue, because it involves not only the interface operations but also their complexity constraints. Informally speaking, such complexity constraints are met by requiring that a search in a BST that is obtained by joining two trees of equal size takes only a constant number of steps more than the search in one of them. We postpone the details until a later, systematic treatment of BST.

Note that in the definition of BST there is no reference to any kind of tree structure, but only to the properties of the interface operations. Their name comes from the fact that they are in fact realized using abstract data structures that do have a tree structure. Examples include AVL trees and Red-Black trees. Clearly, unlike stacks and queues, a BST does not encode the order in which elements are added to the structure, but instead, it allows retrieval of a record by the value of its key.

1.3 Recursion

1.3.1 What is Recursion?

Example 1.3 (Towers of Hanoi) Referring to the Figure 1.12, the task is move discs from A to C, one at a time, never placing a larger disc on top of a smaller disc.

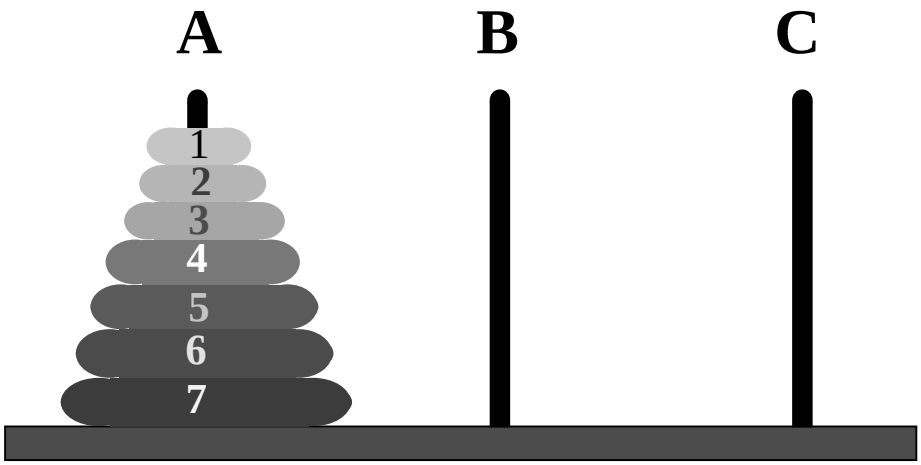


Figure 1.12: Towers of Hanoi

Algorithm: To move n discs from A to C:

Move $n-1$ discs from A to B;

Move the largest disc from A to C;

Move $n-1$ discs from B to C.

Thus, to obtain the procedure for n discs, we call the same procedure for $n-1$ discs twice, with a simple modification (changing the roles of poles). We say that such recursion is linear or of type $n \to n-1$, because the computation for an input with n elements (discs) is reduced to several computations with $n-1$ elements.

Let XY stands for "move the top-most disc of pole X to pole Y". For example, AB means "move the top-most disc of pole A to pole B". If we denote by $^\wedge$ concatenation

of moves, then the above algorithm can be written as:

$$f(n,A,B,C) = f(n-1,A,C,B)^{\wedge}AC^{\wedge}f(n-1,B,A,C)$$

For $n = 4$ we can "unwind" the recursion as in the Figure 1.13 below. The boxes correspond to recursive calls. Note that computations in each box can be completed only after the computations in the smaller boxes contained within are completed.

Exercise 1.4 Tom and his wife Mary went to a party where nine more couples were present. Not every one knew every everyone else, so people who did not know each other introduced themselves and shook hands. People that knew each other from before did not shake hands. Later that evening Tom got bored, so he walked around and asked all other guests (including his wife) how many hands they had shaken that evening, and got 19 different answers. How many hands did Mary shake?

Hint: *Try recursion on the number of couples, by finding what couple you should eliminate first. If there are 19 different answers what could they be? Which ones are the most interesting numbers among those 19 numbers and how are these people related?*

1.3.2 Types of Recursion

Consider the following two recipes to how to cook potatoes:

"Take a kilo of potatoes and boil them for sixty minutes."

"Take a kilo of potatoes and boil them until soft."

These two recipes exemplify the first major distinction among recursion types. Type A is an example of a recursive procedure that involves a number of steps that can be determined before the procedure has started ("keep boiling potatoes for 60 one minute

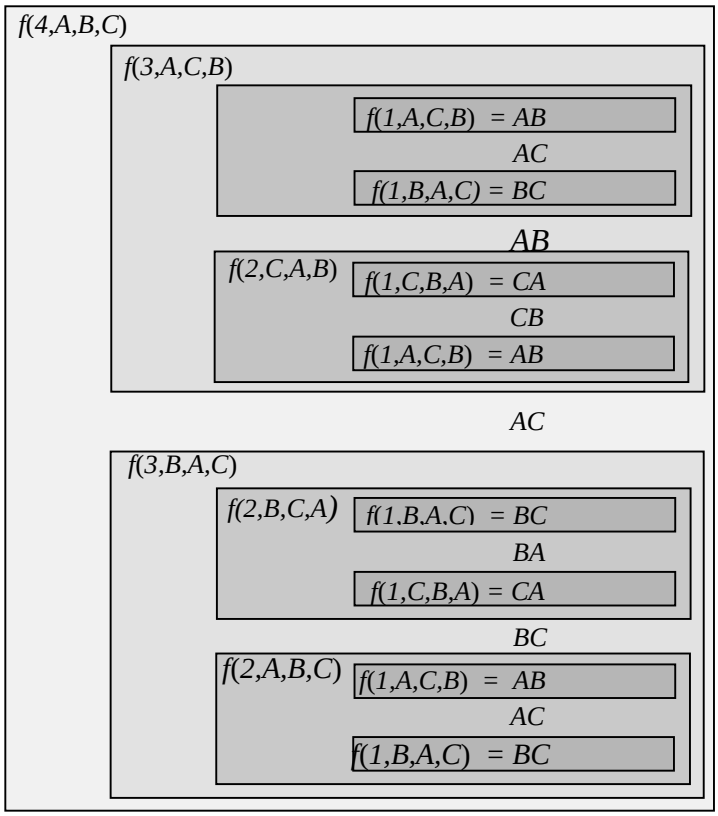


Figure 1.13: The solution of the Towers of Hanoi puzzle for $n = 4$, producing $f(4,A,B,C) = \langle AB,AC,BC,AB,CA,CB,AB,AC,BC,BA,CA,BC,AB,AC,BC\rangle$

intervals"), i.e., of the form: **for** `i=0` **to** `60` **do** $\{\ldots\}$. On the other hand, the number of "one minute boiling intervals" in the procedure B cannot be determined in advance; it is an example of a recursion of the form **do**$\{\ldots\}$ **until** $\{\ldots\}$ Functions computable using only "**for** `i=0` **to** ... **do** $\{\ldots\}$" loops are called primitive recursive functions; functions computable using "**do**$\{\ldots\}$ **until**$\{\ldots\}$ " loops are called general recursive functions.

Example 1.4

1. ***The factorial function*** is defined by primitive recursion: $1! = 1$; $n! = n \cdot (n-1)!$ Note that such function is easily definable by initializing `k = 1` and then

executing the loop **for** i = 1 **to** n **do** k ← k · i.

2. f(n) = n^{th} number p such that both p and p+2 are prime. To get such a number we initialize k=0, p=2 and then execute: **do** {{**if** p **and** (p+2) are prime **then** k←k+1}; p←p+1} **until** {k==n} . In fact, we do not even know if for every n our algorithm will terminate. The hypothesis that there infinitely many primes p such that $p+2$ is also a prime is known the *twin primes hypothesis*.

There are functions that are not primitive recursive, yet they are still simpler than most general recursive functions. The best known example is the Ackerman function defined by double recursion:

$$\begin{aligned} F(0,n) &= n+1 \\ F(m,0) &= F(m-1,1) \\ F(m,n) &= F(m-1,F(m,n-1)) \quad \text{for } m,\ n>0 \end{aligned}$$

One can show that such function cannot be reduced to the usual primitive recursion; in fact, Ackerman function grows faster than any function defined by primitive recursion.

Exercise 1.5 Write a program that computes $F(m,n)$ and try to evaluate $F(3,3)$ and $F(4,4)$.

In this booklet we call a harder exercise a *problem*; nevertheless, you should not be discouraged; try to solve it!

Problem 1.1 Two thieves have robbed a warehouse and have to split a pile of items without price tags on them. How do they do this in away that ensures that each thief

believes that he has got at least one half of the value of the whole pile? *(You might want to try to solve this problem before reading any further.)* The solution is that one of the two thieves splits the pile in two parts, so that he believes that both parts are of equal value. The other thief then chooses the part that he believes is at least one half. Assume now that ten thieves have robbed a warehouse. How do they split the pile of items so that each thief believes that he got at least one tenth of the total value of the pile?

Linear Recursion considered above (except the double recursion used to define the Ackerman function) reduces computation of $F(n,x)$ to one or several computations of $F(n-1,\ldots)$. For example, identity $n! = n(n-1)!$ reduces computation of $n!$ to a computation of $(n-1)!$ and a multiplication of the result by n. Much more efficient is *divide-and-conquer recursion*, that reduces a computation of $F(n,x)$ to computations of $F(\lceil n/k \rceil,\ldots)$, where $k > 1$. Often $k = 2$, and so $F(n,\ldots)$ is reduced to computations of the form $F(\lceil n/2 \rceil,\ldots)$. We sometimes do not write integer part function and simply write $F(n/2,\ldots)$ for $F(\lceil n/2 \rceil,\ldots)$, if no confusion can arise.

Example 1.5 $y = x^n$ can be computed using linear recursion as follows: $x^0 = 1$; $x^{n+1} = x \cdot x^n$. However, one can also use recursion as follows: $x^0 = 1$; $x^{2n} = (x^n)^2$; $x^{2n+1} = x \cdot (x^n)^2$. Clearly, this way the number of multiplications needed to evaluate x^n is at most $2 \lg n$.

Example 1.6 We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.

Solution: Split $27 = 3^3$ coins into three equal groups with 9 coins each. Put two groups on the pan balance. If equal, the false coin must be in the third group; otherwise

take the lighter group. Split these nine coins into three groups with three coins each and compare two of these groups. As in the previous step, you can find a group containing the false coin. Finally, compare two of the three coins left; if equal, the false coin is the remaining coin, otherwise take the coin on the lighter side. □

Notice that if we had $3n$ coins and can weigh coins n times, then splitting them into three groups with $3(n-1)$ coins each and comparing two of the three groups we end up with $3(n-1)$ coins where the fake coin might be and can weigh $n-1$ times. Thus, this type of recursion reduces the size of the problem into a sub-problem of size equal to a fraction of the original problem; this is why such recursion is called divide-and-conquer recursion.

Exercise 1.6 We are given twelve coins and one of them is a counterfeit but we do not know if it is heavier or lighter. Determine which one is a counterfeit and if it is lighter or heavier by weighing coins on a pan balance three times only.

Hint: *divide and conquer; note that you can reuse coins that are established not to be counterfeit!*

Example 1.7 We have nine coins and three of them are heavier than the remaining six. Can you find the heavier coins by weighing coins on a pan balance only four times?

Solution: If we weigh the coins four times, since every weighing has three possible outcomes, we have altogether $3^4 = 81$ outcomes altogether. However, there are $\binom{9}{3} = \frac{9\cdot 8\cdot 7}{3!} = 84$ possible combinations what three coins out of nine might be heavier. Thus, the number of possibilities we have do distinguish between is larger than the number of possible outcomes of our weighing! Consequently, it is not possible to find

the heavier coins by weighing coins on a pan balance only four times. □

This is an example of the lower bound estimation for the complexity of algorithms, i.e., estimation of the minimal number of steps needed to solve a problem for an input of given size. Exactly the same method as above can be used to determine the minimal number of steps needed to sort a sequence of numbers by comparing pairs of numbers.

1.3.3 Linear versus Divide-and-Conquer Recursion in Sorting

1.8 Algorithm 1.1 is called *Insertion Sort* and is based on the procedure used to sort playing cards at hand. Note that the procedure does not call itself; thus, it is an *iterative* procedure, rather than a recursive procedure. On the other hand, algorithm 1.2 is a "genuine recursion" version of insertion sort.

Algorithm 1.1: INSERTION-SORT(A)

```
1: for j ← 2 to length[A] do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     A[i + 1] ← key
```

Note that insertion sort uses only the space taken by the elements of the array plus only one extra storage space for the key. Thus, it is very memory efficient. We say that it *sorts in place*.

We seldom sort "unattached" numbers; most often we sort records according to their keys. When sorting by keys, an important feature that a sorting algorithm can have is *stability*. A sorting algorithm is *stable* if it does not change the ordering among

Algorithm 1.2: INSERTION-SORT-REC($A[1 \ldots n]$)

1: INSERTION-SORT-REC($A[1 \ldots n-1]$)
2: $key \leftarrow A[n]$
3: $i \leftarrow n-1$
4: **while** $i > 0$ **and** $A[i] > key$ **do**
5: $\quad A[i+1] \leftarrow A[i]$
6: $\quad i \leftarrow i-1$
7: $A[i+1] \leftarrow key$

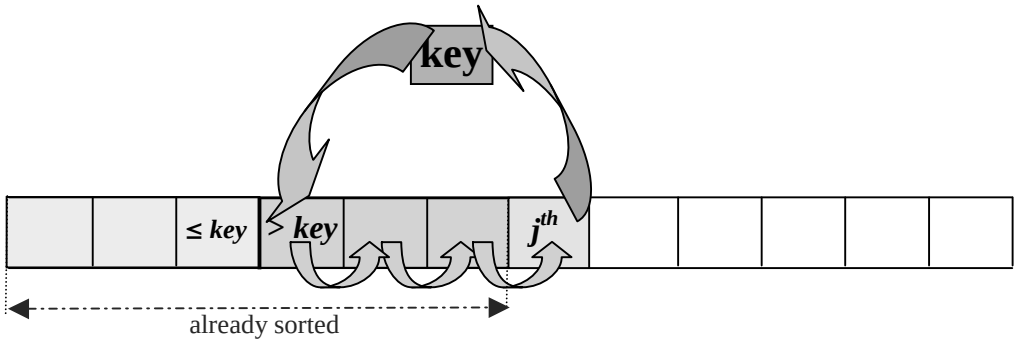


Figure 1.14: Insertion Sort

equal keys. For example, medical records of patients arriving to the emergency ward of a hospital are kept sorted according to their keys, that are assigned in a way that reflects the severity of the patient's injury. Thus, the algorithm that sorts medical records according to such a key must be stable, to keep the patients with the same severity of the injury in the same order as they have arrived. This ensures that the waiting time for each patient is kept as short as possible.

We now analyze INSERTION-SORT i.e., we estimate its run-time.

Worst case: The array is in reverse order (from the largest to the smallest number). In this case each element of the array (except the first) is put in `key` and then compared to all of the previous elements. All of the previous elements have to be shifted to the right and the element stored in `key` is then put into the first cell of the array. For the i^{th} element this takes ci many steps. The constant c depends on what exactly elementary

instructions that can be executed in a single step are. Thus the total number of steps is a quadratic polynomial in n:

$$\sum_{i=2}^{n} ci = c\left(\frac{n(n-1)}{2} - 1\right) = c\,\frac{n^2+n-2}{2} \tag{1.1}$$

We did not count overhead of invoking subroutines etc. However, this overhead is linear in n (i.e., a constant multiple of n) and thus it does not change the nature of the estimate of the time complexity of the entire algorithm, which remains quadratic in n. We will develop later a calculus (the "O-notation") that will allow us to estimate the growth rate of functions without having to worry about such "details".

Average case: If the numbers in the array are randomly chosen, any member of the array is smaller than about one half of the previous elements of the array. Thus, for the i^{th} element in the array there will be about $i/2$ comparisons and shifts. Consequently, the total number of operations will be about

$$\sum_{i=2}^{n} c\,\frac{i}{2} = \frac{c}{2}\,\frac{n^2+n-2}{2} \tag{1.2}$$

i.e., it is again a quadratic polynomial in n. Thus, the average case is about as bad as the worst case. If you have to sort many arrays with randomly distributed content, the algorithm will on average run in quadratic time in the length of the arrays. This, as we will see, is very slow.

Best case: The array is already sorted. In this case each element of the array (except the first) is put in key and then, after a comparison with the previous element, it is returned to its old position. Thus, for each element of the array the number of atomic operations (moving numbers, comparing numbers) is constant. Consequently the total

number of operations is of the form cn, i.e. it is linear in n.

What is wrong with the above "analysis"? There is **no** such thing as "the best case analysis". The "best case" provides no useful information. However, knowing a significant class of inputs for which an algorithm is particularly fast can be useful. For example, if you expect few inversions in your input sequences, then INSERTION SORT algorithm will run in linear time, which is very fast. This happens if you sort checks in a bank: most of the checks arrive to the bank in the order you write them, with few inversions.

Exercise 1.7 Let k be a natural number. Consider the family $\mathscr{A}_k$ of all arrays $A[1 \ldots n]$ such that for every $i \leq n$ there are at most k elements among $A[1 \ldots i-1]$ larger than $A[i]$. Show that there exists a constant c such that every array $A[1 \ldots n]$ from $\mathscr{A}_k$ can be sorted in time $c \cdot n$.

Exercise 1.8 Insertion sort consists of two main operations: searching for the right place in the list where the key should be inserted, and inserting the key element into the list. If the list of elements is realized as an array, what operation is the bottle-neck that accounts for the quadratic run time of the Insertion Sort? What if the list is implemented as a linked list? Refer to the remark 1.3 on page 12.

Divide-and-Conquer Recursion consists of three steps at each level of recursion:

- ***Divide*** the problem into a number of sub-problems of smaller size.
- ***Conquer*** the sub-problems using recursive calls. If the problem size is very small, i.e., at the base of recursion, do it directly without using a recursion call.
- **Combine** the solutions of the sub-problems into a solution of the starting problem.

MERGE-SORT(A, p, r) sorts the elements of the sequence contained in the sub-array $A[p \ldots r]$ of an array A in the following way. It:

- divides the n-element sequence into two subsequences of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$;
- conquers by sorting the two subsequences using recursive calls of Merge-Sort.
- merges the two sorted subsequences to get the initial sequence sorted.

When the *divide* procedure produces a sequence of length one, such sequence is already sorted. To merge two sorted sequences we compare the first members of each subsequence and pick the smaller one as the next element of the joint sequence. We repeat this procedure until we exhaust one of the sequences, at which stage we just concatenate the remaining part of the other sequence to the end of the joint sequence. We now give the pseudo-codes of the `Merge`$[A, p, q, r]$ algorithm which merges sequences $A[p \ldots q]$ and $A[q+1 \ldots r]$, and the pseudo-code for the `Merge-Sort` algorithm:

26	18	26	18	26	18	26	18	26	18	26	18	26	26
19	17	19	17	19	17	19	17	19	17	19	↓	19	↓
13	16	13	16	13	16		16		↓			↓	
8	7	8		↓			↓						
	↓	↓											
	7	8		13			16		17		18	19	26

Example of merging the sequences $\langle 8, 13, 19, 26 \rangle$ and $\langle 7, 16, 17, 18 \rangle$ into a single increasing sequence $\langle 7, 8, 13, 16, 17, 18, 19, 26 \rangle$ is represented in the above table; a run of `Merge-Sort` on input sequence $\langle 7, 4, 2, 5, 3, 2, 3, 1 \rangle$ is shown on Figure1.15.

Algorithm 1.3: `Merge`$[A,p,q,r]$

```
1:  i ← p
2:  j ← q + 1
3:  k ← p
4:  while i < q + 1 and j < r + 1 do
5:      if A[i] ≤ A[j] then
6:          B[k] ← A[i]
7:          i ← i + 1
8:          k ← k + 1
9:          if i = q + 1 then
10:             for m ← j to r do
11:                 B[k − j + m] ← A[m]
12:     else
13:         B[k] ← A[j]
14:         j ← j + 1
15:         k ← k + 1
16:         if j = r + 1 then
17:             for m ← i to q do
18:                 B[k − i + m] ← A[m]
19:     for n ← p to r do
20:         A[n] ← B[n]
```

Analysis of Merge-Sort. How efficient is the Merge-Sort? Assume that the input array has n elements, and let us denote by $T(n)$ the running time for arrays of length n.

Divide This part just calculates the middle point of the array, which takes constant amount of time.

Conquer We apply the same algorithm on arrays of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Thus, the running times are $T(\lceil n/2 \rceil)$ and $T(\lfloor n/2 \rfloor)$, which is smaller or equal than $2T(\lceil n/2 \rceil)$.

Algorithm 1.4: MERGE-SORT(A,p,r)

```
1: if p < r then
2:     q ← ⌊(p+q)/2⌋
3:     MERGE-SORT(A, p, q)
4:     MERGE-SORT(A, q + 1, r)
5:     MERGE(A, p, q, r)
```

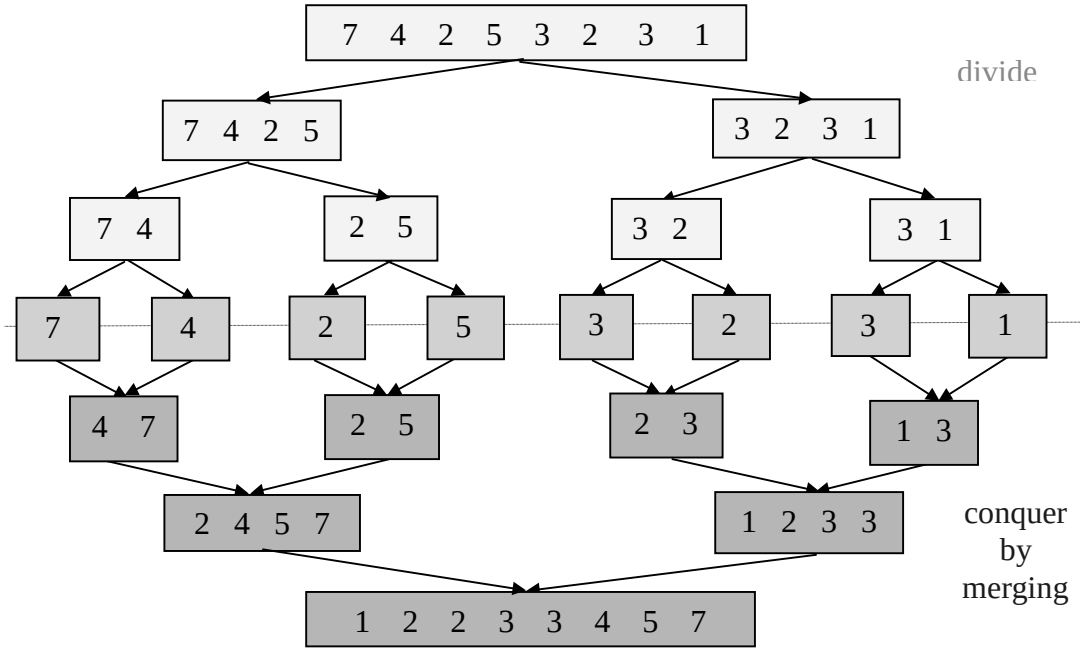


Figure 1.15: Merge Sort example

Combine Uses an application of the MERGE procedure applied on two sequences of total length n. For each element of the merged sequence we did at most one comparison and a few moving operations. Thus, the total number of operations used in the procedure MERGE is linear in n.

Summing up all three components, we get the inequality: $T(n) \leq 2T(\lceil n/2 \rceil) + cn$. Thus, sice a time complexity function is non decreasing, to find an *upper bound* for $T(n)$ it is enough to solve the recurrence $T(n) = 2T(\lceil n/2 \rceil) + cn$.

We "unwind" the recursion as shown on Figure 1.16 and estimate the height of the tree and the work we have to do at each level of the tree.

Clearly, since at each step of *divide* the size of sub-arrays is halved, for $n > 1$ the height of the tree is equal to $\lceil \lg\ n \rceil$. Since the total number of elements in all sub-arrays is always n, and the MERGE procedure works in time linear in n, the total work at each level is cn for some constant c. Thus, the total number of steps of MERGE SORT is a

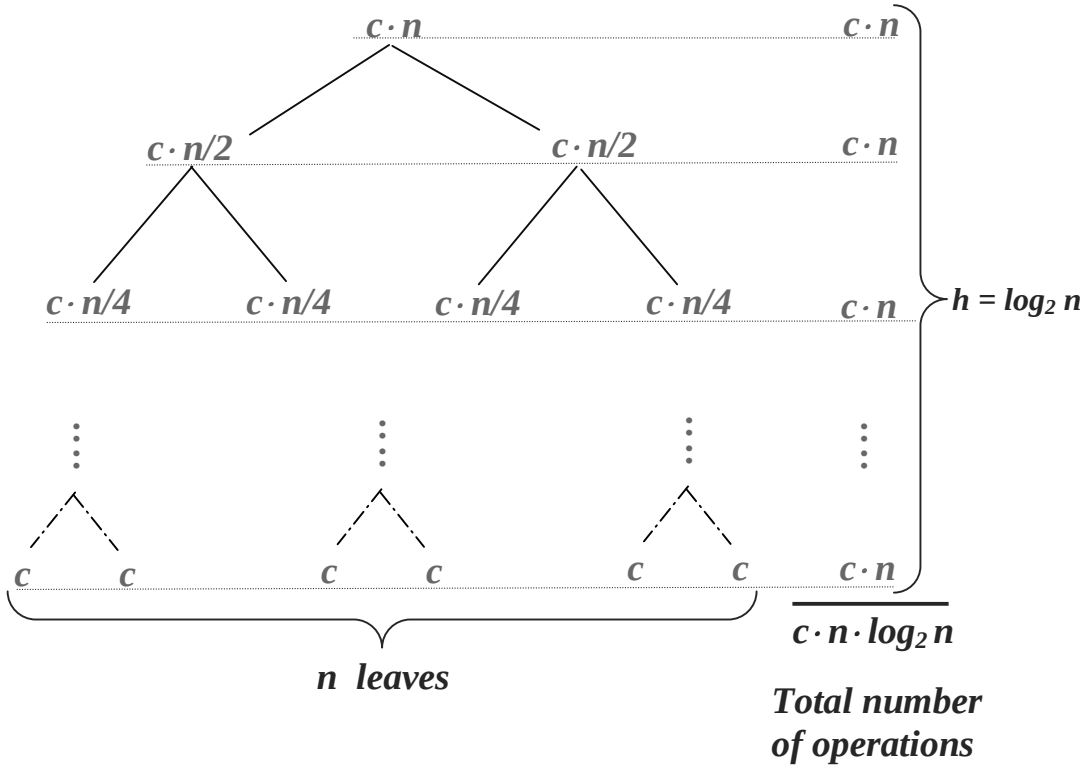


Figure 1.16: Analysis of Merge Sort

constant multiple of $n \lg n$.

Since $\lg n$ is much smaller than n and, as we have seen, insertion sort runs in time that is a multiple of n^2, MERGE SORT is in general much more efficient than the INSERTION SORT. In practice, MERGE SORT beats INSERTION SORT for inputs of size about 30. For much larger n the difference in performance is huge; the growth rates of n^2 and $n \lg n$ are compared on Figure1.18.

Exercise 1.9 1. Show that the MERGE SORT is a stable sorting algorithm.

2. Show that if we replace line 5: $A[i] \leq A[j]$ in Algorithm 1.3 (MERGE) with: $A[i] < A[j]$ the algorithm remains correct, but is no longer stable.

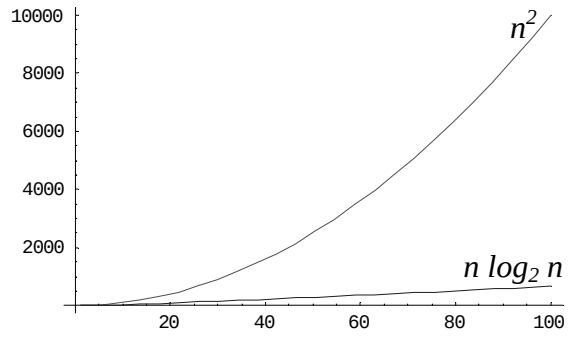


Figure 1.17: Growth rate comparison for n^2 (dark) and $n \lg\ n$ (light)

1.4 Asymptotic Time Complexity of Algorithms

1.4.1 Basics

Present day microprocessors are all capable of implementing the same kind of computations, and differ essentially only in terms of speed of execution and their memory capacity. This feature of processors is called *Turing completeness*. Turing completeness explains why we can buy software for new tasks without having to buy a new, different computer for each new task. Our computers are "universal machines" in the sense that they can execute different tasks by changing only their software.

Nearly all processors of the type that is commonly used in general purpose computers are built according to the same architecture, usually called the *Von Neumann Architecture*. They all consist of a CPU (*Central Processing Unit*) and a (potentially infinite) collection of memory cells that can each be directly accessed, called *Random Access Memory* (RAM). All such machines are equivalent to a theoretical model of computation, called the Universal Turing Machine and thus are equally powerful computing machines (save memory and speed constraints).

Two different (sequential) processors A, B may have different "hard-wired" basic

operations, i.e., different sets of instructions implemented in the hardware by special circuits. Since they can perform the same tasks, every instruction executable as a basic instruction of A, if not hard-wired in B, can be implemented as a sequence of steps of instructions of B. Since there are only finitely many basic instructions implemented in the hardware of A, there exists a natural number M such that every basic instruction of A can be implemented as a sequence of at most M basic instructions of B. Thus, a program p containing only basic instructions available on A can replaced by a program containing only instructions available on B, such that p' is at most as long as M times the length of p. Consequently, execution time of the same algorithm on A and B for the same input can differ by at most a multiplicative constant M.

Since algorithms to be run on a processor of Von Neumann Architecture machines are abstract entities independent of individual platforms on which they can be executed, we need a method for measuring the execution time of an algorithm that is independent for inessential features of the algorithm, such as which particular basic instructions are used in specifying the algorithm. In light of the above discussion, we want to consider two algorithms with execution times that, for sufficiently large inputs, differ by at most a multiplicative constant as equally efficient. This leads us to the notion of asymptotic growth rate of functions.

1.4.2 Asymptotic growth rate of functions

"Big Oh notation": $f(n) = O(g(n))$ is abbreviation for the statement *"There exists positive constants c and* n_0 *such that* $0 \leq f(n) \leq cg(n) for all n \geq n_0$*"*. In this case we say that $g(n)$ *is an asymptotic upper bound for* $f(n)$.

Thus, $f(n) = O(g(n))$ means that $f(n)$ does not grow substantially faster than $g(n)$ because a multiple of $g(n)$ eventually dominates $f(n)$. Clearly, multiplying constants of interest will be larger than 1, thus "enlarging" $g(n)$.

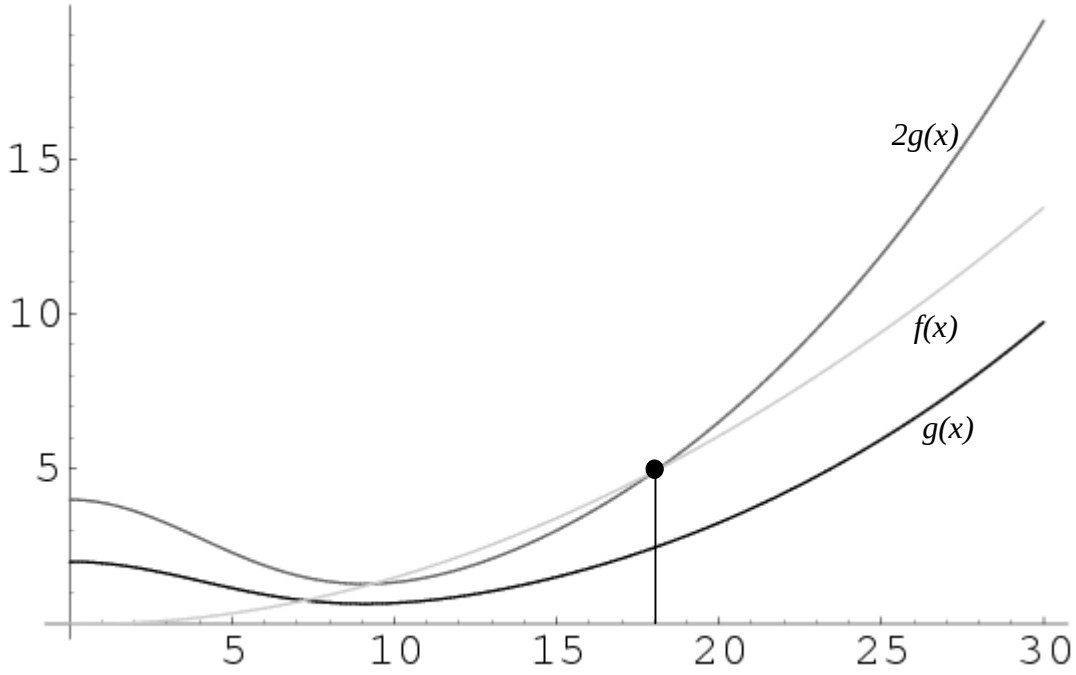


Figure 1.18: $f(x) = O(g(x))$; $c = 2$ and $n_0 = 18$. Thus, $f(x) \leq 2g(x)$ for $n \geq 18$

Example 1.8

1. $f(n) = n^2 + 10n$ does not grow substantially faster than $g(n) = n^2$ in the sense of the above definition because $f(n) = n^2 + 10n \leq 2n^2 = 2g(n)$ for all $n \geq 10$. Thus, here we can take $c = 2$, $n_0 = 10$, and so $n^2 + 10n = O(n^2)$.

2. $f(n) = 1000n^2$ does not grow substantially faster than $g(n) = n^2$ because $f(n) = 1000n^2 = 1000g(n)$ for all $n \geq 0$. Thus, for $c = 1000$ and $n_0 = 0$ we get $1000n^2 = O(n^2)$.

3. $f(n) = n^2$ does grow substantially faster than $g(n) = 1000n$ because there are no constants c, n_0 such that $f(n) = n^2 \leq c1000n$ for all $n \geq n_0$. This is because $n \geq 1000c$ implies $n^2 > 1000cn$. Consequently, $1000n = O(n^2)$.

Thus, a linear function can never be an asymptotic upper bound for a quadratic function.

"Omega notation": $f(n) = \Omega(g(n))$ is an abbreviation for the statement *"There exists positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$."* In this case we say that $g(n)$ *is an asymptotic lower bound for* $f(n)$.

Thus, $f(n) = \Omega(g(n))$ essentially says that $f(n)$ grows at least as fast as $g(n)$, because $f(n)$ eventually dominates a multiple of g(n). Clearly, multiplying constants of interest will be smaller than 1, thus "reducing" g(n) by a constant factor.

Example 1.9

1. $g(n) = 10n$ is an asymptotic lower bound for $f(n) = n^2$ because when n is sufficiently large ($n > 10$) we have $g(n) = 10n < n^2 = f(n)$. Thus, here we can take $c = 1$ and $n_0 = 10$.

2. $g(n) = n^2$ is *not* an asymptotic lower bound for $f(n) = 10n$ because there is no constant c such that $10n > cn^2$ for sufficiently large n. No matter how small $c > 0$ is chosen, cn^2 will eventually be larger than $10n$.

"Theta notation": $f(n) = \Theta(g(n))$ stands for $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. In this case we say that f and g have the same asymptotic growth rate, or that are *asymptotically equivalent*.

Thus, $f(n) = \Theta(g(n))$ means that $g(n)$ is both an asymptotical upper bound and an asymptotical lower bound for $f(n)$. It is easy to see that in this case there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1\ g(n) \leq f(n) \leq c_2\ g(n)$ for all $n \geq n_0$.

Example 1.10

1. $n2+5n+1 = \Theta(n^2)$ because $0 \leq n^2 \leq n^2+5n+1 \leq 2n^2$ for all $n \geq 6$. In general, for any polynomial $p(x)$ of order k we have $p(x) = \Theta(x^k)$.

2. $2^{n+1} = \Theta(2^n)$ because $0 \geq 2^n \geq 2^{n+1} = 2 \cdot 2^n$ for all $n \geq 0$.

1.4.3 Basic property of logarithms

1.9 The Logarithm function is very important for estimating the run time of algorithms. For example, we have seen that the MERGE SORT algorithm runs in time $O(n \lg\ (n))$. We now review briefly the main properties of the logarithm function.

$$\log_c(ab) = \log_c a + \log_c b \tag{1.3}$$

$$\log_c(a/b) = \log_c a - \log_c b \tag{1.4}$$

$$\log(a^b) = b\ \log a \tag{1.5}$$

$$a^{log_a b} = b \tag{1.6}$$

$$\log_a x = log_b x\ \log_a b\ =\ \log_b x / \log_b a \tag{1.7}$$

Proof: Properties 1.3 - 1.5 follow directly from the definition of the log function, i.e., from the fact that $\log_c x = y$ just in case $c^y = x$, and the corresponding basic properties of exponentiation: $c^{x+y} = c^x c^y$; $c^{x-y} = c^x / c^y$; $(c^x)^y = c^{xy}$. Property 1.7 is a bit trickier: let $\log_b x = w$; thus, $b^w = x$. Using 1.6 and substituting in $b^w = x$ we get

$(a^{\log_a b})^w = x$, i.e., $a^{w \log_a b} = x$. Take now $\log_a$ of both sides to get $w \log_a b = \log_a x$, i.e., $\log_b x \log_a b = \log_a x$. □

Note that 1.7 implies that for every two numbers $a, b > 1$ we have $\log_a n = \Theta(\log_b n)$.

1.10 ***The growth rate of the factorial function.*** We now show that $\lg\ n! = \Theta(n\ \lg\ n)$. Using Stirling's formula $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)(1+\Theta(1/n))$ and taking lg of both sides we get:

$$\lg\ (n!) \quad = \quad \lg\ \sqrt{2\pi n} + n\ \lg\ n - n\ \lg\ e + \lg\ (1+\Theta(1/n))$$

However, it is easy to see that the expression on the right is $\Theta(n\ \lg\ n)$.

Exercise 1.10 Prove $\lg\ (n!) = \Theta(n\ \lg\ n)$ using basic algebra only.

Hint: *Note that for $n > 0$, $n! \leq n^n$ and take logarithm of both sides; for the other direction show that $n^n < (n!)^2$ by pairing suitably factors of $(n!)^2$ and then using the fact that $x(n+1-x) < n$ for all $1 \leq x \leq n$. This inequality can be proved using basic algebra of quadratic polynomials.*

Exercise 1.11 Determine if $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ or $f(n) = \Theta(g(n))$ for the following pairs:

$f(n)$	$g(n)$
n	$(n - 2\ \lg\ n)(n + cos\ n)$
$(\lg\ n)^2$	$\lg\ (n^{\lg\ n}) + 2\ \lg\ n$
$n^{1+sin(\pi\ n/2))/2}$	$\sqrt{n}$ (a plot might help)

How important is the run time of an algorithm? Is it just the matter of “elegance”? The graph on the Figure 1.19 might help explain the significance of algorithm efficiency. To comprehend the growth rate of these functions even better, assume we have algorithms with time complexities shown in the table, that are executed on a 1 GHz machine executing one instruction each clock cycle. In the table one can find how long it would take to complete these programs; time is in seconds unless stated otherwise.

For many real-time applications, for example algorithms used in digital telecommunications, algorithms that run in quadratic time are unacceptably slow. For standard applications cubic run time is unacceptably slow, unless inputs are of relatively small size. ”Brute force” algorithms often tend to run in exponential time, and as you can see, for inputs of size only 100 it would take the entire age of the universe! Thus, poor efficiency of an algorithm can often render it entirely useless.

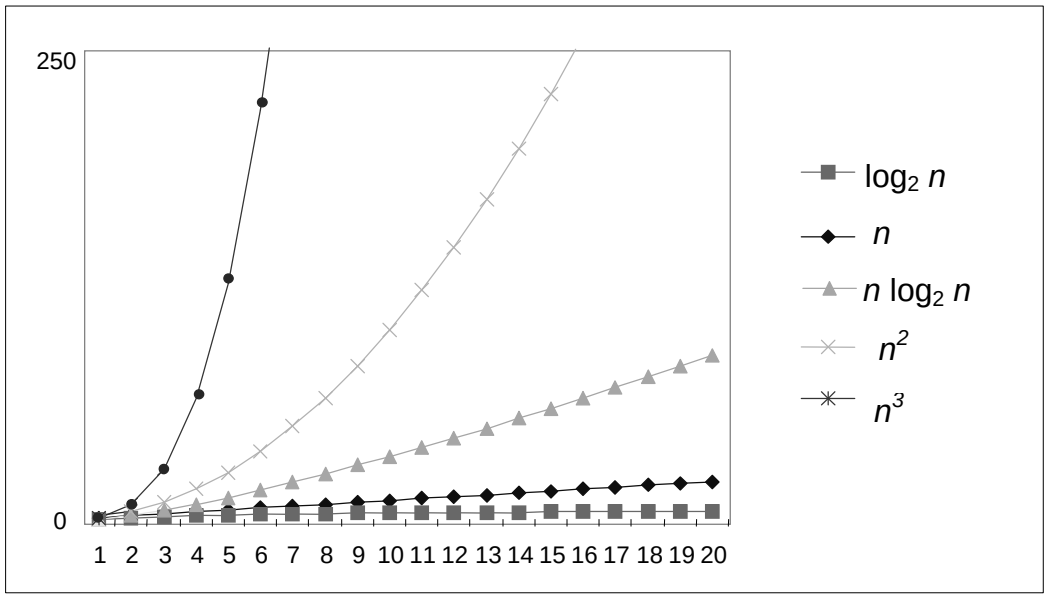


Figure 1.19: Growth rate comparison for basic functions

$n =$	10	100	1,000	10,000	100,000	1,000,000
$\lg n$	$3 \cdot 10^{-9}$	$6 \cdot 10^{-9}$	$9 \cdot 10^{-9}$	$1.3 \cdot 10^{-8}$	$1.6 \cdot 10^{-8}$	1
n	10^{-8}	10^{-7}	10^{-6}	10^{-5}	1	10^{-3}
$n \lg n$	$3 \cdot 10^{-8}$	$7 \cdot 10^{-7}$	10^{-5}	10^{-4}	1	10^{-2}
n^2	10^{-7}	10^{-5}	10^{-3}	0.1	10	17 min
n^3	10^{-6}	10^{-3}	1	17 min	11.5 days	37 years
2^n	10^{-7}	age of the universe	10^{292} ridiculous	10^{3001} non-sensical	10^{30093} pure math	$10^{3010191}$ pure theology

We now turn to a systematic study of sorting algorithms.

Chapter 2

Sorting Algorithms

Sorting is one of the most frequently encountered data processing tasks. There are many sorting algorithms, and one could wonder why there are so many algorithms for the same task. We will see that for different probability distributions of inputs, different algorithms for the same task can have vastly different performance. For example, we have seen that INSERTION SORT is extremely fast if inputs are nearly sorted; on such inputs it runs in linear time. However, if all permutations of inputs are equally likely, then INSERTION SORT is extremely slow; for such inputs it runs in quadratic time. `Merge Sort`, on the other hand, runs in time $\Theta(n\ lgn)$ for all inputs, but the constants involved in this asymptotic bound are not very small. For that reason, the MERGE SORT algorithm is rather slow for smaller size inputs (less than about 30-40 elements). We will study another sorting algorithm, the QUICK SORT, that has very low average sorting time for random inputs, but if inputs are frequently nearly sorted, then QUICK SORT is very slow: it runs in quadratic time. *Thus, the choice of a sorting algorithms depends on the expected input.*

Besides being important on their own right as massively used tools, sorting algo-

rithms will be also used to illustrate:

1. how to chose the right algorithm for the task at hand;
2. general algorithm design techniques, like divide-and-conquer;
3. how to estimate time complexity of algorithms in general.

2.1 The Quick Sort

The Quick Sort is one of the most frequently used sorting algorithms. Assume that we want to sort the part of an array A that is between indices p and r, i.e., to sort $A[p \dots r]$. We employ "divide-and-conquer" strategy, as follows:

Divide: The array $A[p \dots r]$ is partitioned and rearranged into two nonempty sub-arrays $A[p \dots q]$ and $A[q+1 \dots r]$ such that every element of $A[p \dots q]$ is smaller or equal than $A[p]$, chosen to be *the pivoting element*, and every element of $A[q+1 \dots r]$ is larger or equal than $A[p]$. Thus, each element of $A[p \dots q]$ is smaller or equal than every element of $A[q+1 \dots r]$.

Conquer: The two sub-arrays $A[p \dots q]$ and $A[q+1 \dots r]$ are sorted by recursive calls to QUICK SORT.

Combine: Since both sub-arrays are now sorted in place, no work is needed to combine them; the entire array $A[p \dots r]$ is already sorted.

We first describe the PARTITION procedure which rearranges and splits the array.

Algorithm 2.1: PARTITION(A, p, r)

```
1:  pivot ← A[p]
2:  i ← p − 1
3:  j ← r + 1
4:  repeat
5:      j ← j − 1
6:  until A[j] ≤ pivot
7:  repeat
8:      i ← i + 1
9:  until A[i] ≥ pivot
10: if i < j then
11:     exchange A[i] ↔ A[j]
12: else
13:     return j
```

Here is an example run for the PARTITION algorithm.

	p	$p+1$						r	
i	4	10	9	3	5	7	2	6	j
	i 4	10	9	3	5	7	j 2	**6**	
	i 2	10	9	3	5	7	j **4**	**6**	
	2	i 10	9	j 3	**5**	**7**	**4**	**6**	
	2	i 3	9	j **10**	**5**	**7**	**4**	**6**	
	2	j 3	i **9**	**10**	**5**	**7**	**4**	**6**	

We can now define the QUICK SORT algorithm.

Algorithm 2.2: QUICK SORT(A, p, r)

```
1: if p < r then
2:     q ← PARTITION1(A, p, r)
3: QUICK SORT(A, p, q)
4: QUICK SORT(A, q + 1, r)
```

How fast is the QUICK SORT? This is not a simple question.

The Worst Case Behavior. When the array is already sorted or in the reverse order (from the largest to the smallest element), then the partition procedure produces one region with a single element, and one region with $n-1$ elements. Clearly, this is a maximally unbalanced partition, and it is easy to see that in fact, in this case the number of steps needed to execute the QUICK SORT algorithm is maximal for all inputs of this length. The running time is then given by the recurrence relation $T(n) = T(n-1) + \Theta(n)$, because in the next stage of recursion the algorithm is applied to a single subarray with $n-1$ elements, and the PARTITION algorithm clearly runs in linear time. Thus, for some c_1, c_2 and all sufficiently large n,

$$T(n-1) + c_1 n < T(n) < T(n-1) + c_2 n$$

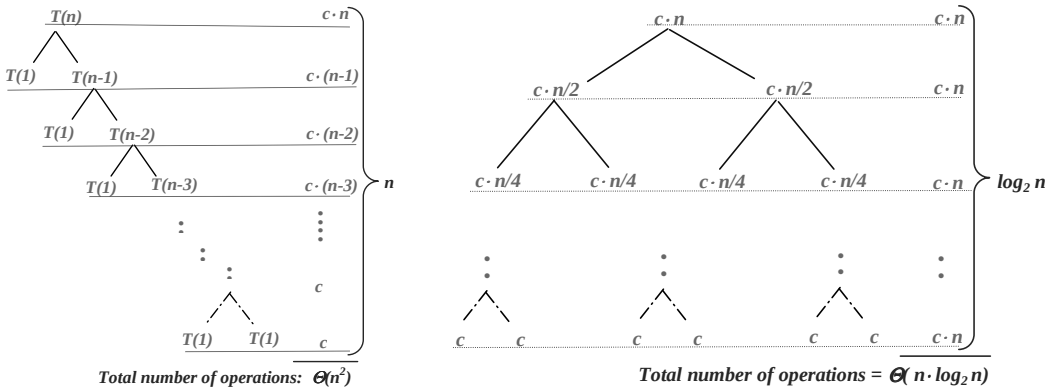
Thus, by consecutive substitutions and by summing the resulting arithmetic progression, we get that for all sufficiently large n

$$c_1 + 2c_1 + 3c_1 + \ldots nc_1 \ < T(n) < \ c_2 + 2c_2 + 3c_2 + \ldots nc_2$$

i.e.,

$$\frac{c_1\, n(n+1)}{2} \ < T(n) < \ \frac{c_2\, n(n+1)}{2}$$

Thus, in the worst case $T(n) = \Theta(n^2)$. The best case happens when the PARTITION procedure always produces two regions of equal size. In this case we have the recurrence $T(n) = 2T(n/2) + \Theta(n)$. Just as for the MERGE SORT algorithm, we have that in this case $T(n) = \Theta(n\ lg\ n)$. Let us examine the "intermediate" cases. Assume, for example, that the PARTITION algorithm always produces a split that is at most 9:1



(a) Worst case with maximally unbalanced partitions

(b) Best case with perfectly balanced partitions

Figure 2.1: Extreme cases of QUICK SORT performance

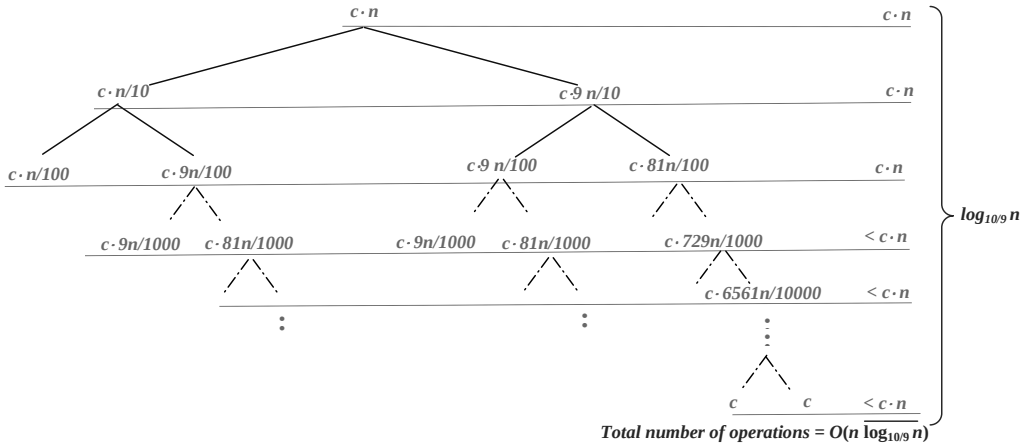


Figure 2.2: A case of QUICK SORT run with partitions 1:9

unbalanced. *Prima facie,* this might appear as a quite unbalanced partitioning; see Figure 2.2. The height of the resulting tree is equal to the smallest integer h such that $(9/10)^h n \leq 1$. This implies $n \leq (10/9)^h$ and so $h = \lceil \log_{\frac{10}{9}} n \rceil$. Thus, $h = O(\log_{\frac{10}{9}} n)$. Since $\log_{\frac{10}{9}} n = \log_{\frac{10}{9}} 2 \ \log_2 n$ we get that also $h = O(\lg n)$. On each level the cost is $O(n)$, because it is the cost of splitting and rearranging the arrays whose total length is at most n; after some of the branches of the tree terminate, the total length of arrays on which the recursion is called again is strictly smaller than n. Thus, the total cost is

again $O(n\ lgn)$. Consequently, the only case when QUICK SORT runs slow is if most of the splitting results in extremely unbalanced partitions. This looks unlikely to happen for randomly produced inputs, because only very particular arrays cause extremely unbalanced partitions. To precisely estimate the average run time of QUICK SORT, let us assume that the chosen pivot is the i^{th} element in size among the elements of the input array. Then the *average* sorting time *under such assumption*, denoted $T_j(n)$, is given by

$$T_j(n) = T(j) + T(n-j) + \Theta(n) \tag{2.1}$$

where $T(j)$ and $T(n-j)$ are average times to sort an array of length j and $n-j$ respectively. This is because the cost of the partitioning procedure is linear in n, i.e., $\Theta(n)$, and $T(j)$ and $T(n-j)$ are costs of sorting procedures for the two sub arrays of length j and $n-j$ respectively. Since for all $j \leq n$ the probability that the pivoting element $A[p]$ is j^{th} in size among all elements of the array, is equal, we get that the average sorting time for an input of length n, denoted by $T(n)$, is given by:

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

Since the term $T(i)$ is equal to the term $T(n-j)$ for $j = n-i$, all the terms repeat twice, and we have:

$$T(n) = \frac{2}{n}\sum_{j=1}^{n-1} T(j) + c\ n \tag{2.2}$$

For $n-1$ this becomes

$$T(n-1) = \frac{2}{n-1}\sum_{j=1}^{n-2} T(j) + c(n-1) \tag{2.3}$$

Multiplying the first equation by n and the second by $n-1$ and by subtracting, we get

$$nT(n)-(n-1)T(n-1)=2T(n-1)+c(n^2-(n-1)^2) \tag{2.4}$$

because all terms of the second sum cancel the corresponding term of the first sum, leaving only $2T(n-1)$, that is not present in the second sum. Thus,

$$nT(n)-(n+1)T(n-1)=c(2n+1) \tag{2.5}$$

which, by dividing both sides by $n(n+1)$, becomes

$$\frac{T(n)}{n+1}-\frac{T(n-1)}{n}=\frac{c(2n+1)}{n(n+1)}<\frac{2c}{n} \tag{2.6}$$

i.e.,

$$\frac{T(n)}{n+1}<\frac{T(n-1)}{n}+\frac{2c}{n} \tag{2.7}$$

By applying the same fact on $T(n-1)$ we get

$$\frac{T(n-1)}{n}<\frac{T(n-2)}{n-1}+\frac{2c}{n-1} \tag{2.8}$$

by combining 2.7 and 2.8 we get

$$\frac{T(n)}{n}<\frac{T(n-2)}{n-1}+\frac{2c}{n}+\frac{2c}{n-1} \tag{2.9}$$

Applying 2.7 to $T(n-2)$ and continuing in the above manner, we get

$$\frac{T(n)}{n}<2c\sum_{i=1}^{n}\frac{1}{i} \tag{2.10}$$

The above sum is a partial sum of the harmonic series $\sum_{j=1}^{n} \frac{1}{j}$, and can be easily bounded by a definite integral, as shown on Figure 2.3. Since the surface area under the graph of

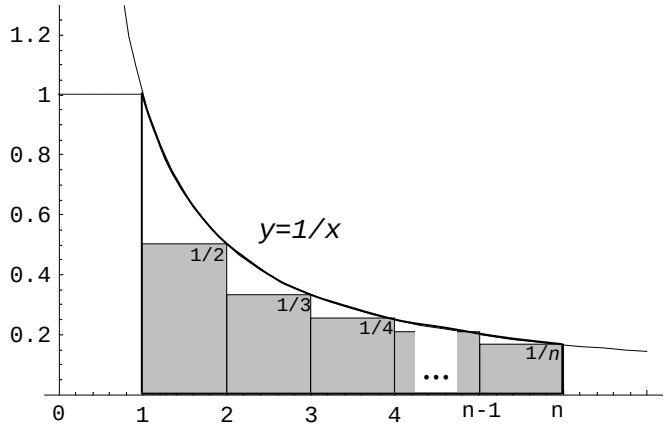


Figure 2.3: Estimating the partial sum of the harmonic series

$y = 1/x$ and between $x = 1$ and $x = n$ is larger than the sum of the areas of the rectangles shown, we get $\sum_{j=2}^{n} \frac{1}{j} < \int_{1}^{n} \frac{1}{x} dx$. Thus, $\sum_{j=1}^{n} \frac{1}{j} < 1 + (\ln n - \ln 1) = 1 + \ln n$ and so $\frac{T(n)}{n} < 2c(1 + \ln n)$. This implies the average (expected) run time of the is $T(n) = O(n \lg n)$.

Exercise 2.1 What is the performance of the QUICK SORT algorithm if all elements of the input array are equal?

Exercise 2.2 It appears that lines 7 , and 10 in 2.1 involve unnecessary work; *prima facie*, if elements are equal to the pivot, there is no need to swap them. How would the performance of the PARTITION procedure change if these lines are replaced by $A[j] < pivot$ and $A[j] > pivot$ respectively? ***Hint:*** *Show that the resulting partition becomes very unbalanced if there are many elements equal to the* pivot. *Thus, non strict inequalities actually make the algorithm behave better on arrays that contain lots of repetitions, by keeping the partition balanced.*

Exercise 2.3 A different version of the PARTITION procedure is given in the second edition of our textbook (Cormen, Leiserson, Rivest and Stein, "*Introduction to algorithms*"). The procedure first picks $A[r]$ as a "pivoting" element. Then it "grows" two regions, one consisting of elements smaller or equal than the *pivot*, placed to occupy positions from p to i, and the other consisting of elements larger than $A[r]$, placed to occupy positions $i+1$ to $j-1$. This is accomplished by advancing pointer j, and whenever the j^{th} element is smaller than the pivot, it is swapped with the first element in $A[p \ldots j]$ that is larger than the pivot, namely with $A[i+1]$, and pointer i is advanced to the next position in the array, thus again delimiting the region consisting of elements smaller or equal to the pivot.

Algorithm 2.3: PARTITION2(A, p, r)

```
1: pivot ← A[r]
2: i ← p − 1
3: for j ← p to r − 1 do
4:     if A[j] ≤ pivot then
5:         i ← i + 1
6:         exchange A[i] ↔ A[j]
7: exchange A[i + 1] ↔ A[r]
8: return i + 1
```

	p				p+4			r
i	j 4	10	9	3	5	7	2	6
	i 4	j 10	9	3	5	7	2	6
	i 4	**10**	j 9	3	5	7	2	6
	i 4	**10**	**9**	j 3	5	7	2	6
	4	i 3	**9**	**10**	j 5	7	2	6
	4	3	i 5	**10**	**9**	j 7	2	6
	4	3	i 5	**10**	**9**	**7**	j 2	6
	4	3	5	i 2	**9**	**7**	**10**	6
	4	3	5	i 2	6	**7**	**10**	**9**

An example of execution of the PARTITION2 algorithm is shown in the table, for input consisting of the array of integers 4, 10, 9, 3, 5, 7, 2, 6. Each row represents the content of the array after a single step of execution of the PARTITION2 algorithm, and also shows the positions of the pointers i, j at each step. Numbers in the region for numbers smaller than the pivot are shown in small italics, and numbers larger than the pivot in large bold print. The value returned is one plus the last value of i; in this case it is $p+4$. The corresponding version of QUICK SORT2(p,r) algorithm, operating on the part of an array A between positions p and r is shown below.

Algorithm 2.4: QUICK SORT2(A, p, r)

1: **if** $p < r$ **then**
2: $q \leftarrow$ PARTITION2(A, p, r)
3: QUICK SORT2$(A, p, q-1)$
4: QUICK SORT2$(A, q+1, r)$

1. What is the performance of the QUICK SORT2 algorithm if the input array has

lots of repetitions, say, if all input elements are equal?

2. One might want to improve the performance of the above version QUICK SORT2 algorithm by building *three*, rather than two regions; one for all elements smaller than the pivot, one for elements equal to the pivot and one for elements larger than the pivot. The corresponding algorithms for such approach are PARTITION3 and QUICK SORT3. Determine which "fix" is better overall , QUICK SORT3 or QUICK SORT4, in terms of performance for random inputs and for inputs that contain repetitions.

Algorithm 2.5: PARTITION3(A, p, r)

```
1:  pivot ← A[r]
2:  i ← p − 1
3:  k ← p − 1
4:  for j ← p to r − 1 do
5:      if A[j] < pivot then
6:          i ← i + 1
7:          exchange A[i] ↔ A[j]
8:          k ← k + 1
9:          if k > i then
10:             exchange A[k] ↔ A[j]
11:     else if A[j] = pivot then
12:         k ← k + 1
13:         exchange A[k] ↔ A[j]
14: exchange A[k + 1] ↔ A[r]
15: return (i, k + 1)
```

Algorithm 2.6: QUICK SORT3(A, p, r)

```
1: if p < r then
2:     (q, s) ← PARTITION3(A, p, r)
3: QUICK SORT3(A, p, q)
4: QUICK SORT3(A, s + 1, r)
```

How does such QUICK SORT3 perform if there are a lot of repetitions in the

array? Is the performance of this algorithm with such a "fix" adversely affected if there are only a few repetitions in the array?

3. Finally, lets consider yet another "fix" for PARTITION, algorithm PARTITION4, and the corresponding QUICK SORT4.

Algorithm 2.7: PARTITION4(A, p, r)

```
1:  pivot ← A[r]
2:  i ← p − 1
3:  k ← 1
4:  for j ← p to r − 1 do
5:      if A[j] < pivot then
6:          i ← i + 1
7:          exchange A[i] ↔ A[j]
8:      else if A[j] = pivot then
9:          k ← −k
10:         if k < 0 then
11:             i ← i + 1
12:             exchange A[i] ↔ A[j]
13: exchange A[i + 1] ↔ A[r]
14: return i + 1
```

Algorithm 2.8: QUICK SORT4(A, p, r)

```
1: if p < r then
2:     q ← PARTITION4(A, p, r)
3: QUICK SORT3(A, p, q − 1)
4: QUICK SORT3(A, q + 1, r)
```

Is this "fix" better than the previous one, and under what circumstances? How much unnecessary overhead is introduced by this fix if the array has very few or no repetitions?

2.2 Randomization

In practice, the Quick Sort is on average about twice as fast as the Merge Sort. However, despite its low average run time, on particular inputs the Quick Sort can be extremely slow: it might run in quadratic time. For example, if you use the Quick Sort to sort checks in a bank, the performance of Quick Sort will be extremely poor, because checks appear in nearly sorted order, and for such sequences the Quick Sort has quadratic run time. However, it is easy to fix this problem of the Quick Sort algorithm. We can insure that the performance of the Quick Sort algorithm on a large set of inputs is essentially independent on the probability distribution of different permutations of input strings by using *randomization*. Randomization **cannot** guarantee good performance on any particular input, but it brings the *average* run time for this set of inputs closer to the *expected value* of the run time of the Quick Sort algorithm for random inputs, i.e., for inputs in which all permutations are equally likely. Thus, we can expect efficient performance of the randomized Quick Sort algorithm on large sets of inputs.

Randomization is accomplished by picking the pivot from the input array randomly, by first swapping the pivoting element with a randomly chosen element. Thus, for example, Algorithm 2.1 is transformed in this way into Algorithm 2.9; RANDOM(p,r) is a random number generator producing integers between p and r with (approximately) equal probability.

Clearly, our analysis of the expected time of the QUICK SORT algorithm equally applies to the expected time of the RANDOMIZED QUICK SORT algorithm. Note that randomization does **not** reduce the expected run time, but only insures that no particular permutation of input elicits slow performance of the algorithm.

Note that calling a random number generator at each call of the PARTITION algo-

Algorithm 2.9: PARTITION(A, p, r)

```
1:  exchange A[p] ↔ A[RANDOM(p, r)]
2:  pivot ← A[p]
3:  i ← p − 1
4:  j ← r + 1
5:  repeat
6:      j ← j − 1
7:  until A[j] ≤ pivot
8:  repeat
9:      i ← i + 1
10: until A[i] ≥ pivot
11: if i < j then
12:     exchange A[i] ↔ A[j]
13: else
14:     return j
```

rithm is computationally expensive, so one has to be careful when deciding if expected inputs warrant using the RANDOMIZED QUICK SORT algorithm.

Exercise 2.4 (reviewing basic probability and expectation) Assume I took 1000 envelopes and he put in one of them 1000 dollars.

1. I now offer you to pick one of the envelopes for the price of 10 cents. Would you pay?

2. How about if I ask you for $10 dollars?

3. What is the highest price that you would pay? Would you pay 999 dollars for all of the envelopes? Would you pay 999 dollars for all envelopes but one?

4. In general, what is reasonable price to pay in such cases? What should be the relationship between the expected gain and the price?

5. Can you base your decision solely on the basis of expected gain and the price?

Other methods for improving the performance of the Quick Sort involve ensuring that we pick a good pivot (i.e., as close to the median of elements in the array, thus ensuring a balanced ensuing partition. They include taking a median of three elements, picked one from both ends of the array and one from the center. Also, when the size of the input array becomes small, recursive divide and conquer becomes inefficient: small size arrays are best sorted by the Insertion Sort. Thus, one improvement of the Quick Sort involves stopping the recursive partitioning when the size of the array drops below certain threshold, usually about 20 or so. Small size sub-arrays are left unsorted, until the Quick Sort terminates, and then the resulting "nearly sorted" array is sorted in linear time using Insertion Sort.

Since the efficiency of the Quick Sort comes from the simplicity of the inner loop, as with randomization, every "improvement" must be carefully weighted against added computational complexity. Lots of useful details on this topic can be found in Sedgewick's classical book *Algorithms* (editions are available with implementations for several languages, including *C*, *C++* and *Java*).