Definition, Structure and Properties of Algorithms 1.2

Definition An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

Structure and properties

An algorithm has the following structure:

(i) Input step
(ii) Assignment step
(iii) Decision step
(iv) Repetitive step
(v) Output step

Example 1.2 Consider the demonstration of Al Khwarizmi's algorithm shown on the addition of the numbers 987 and 76 in Example 1.1. In this, the input step considers the two operands 987 and 76 for addition. The assignment step sets the pair of digits from the two numbers and the previous carry digit if it exists, for addition. The decision step decides at each step whether the added digits yield a value that is greater than 10 and if so, to generate the appropriate carry digit. The repetitive step repeats the process for every pair of digits beginning from the least significant digit onwards. The output step releases the output which is 1063.

An algorithm is endowed with the following properties:

Finiteness	an algorithm must terminate after a finite number of steps.
Definiteness	the steps of the algorithm must be precisely defined or unambiguously specified.
Generality	an algorithm must be generic enough to solve all problems of a particular class.
Effectiveness	the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation!
Input-Output	the algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties. Thus one could even write an algorithm in one's own expressive way to make a cup of hot coffee! However, there is this observation that a cooking recipe that calls for

instructions such as "add a pinch of salt and pepper', 'fry until it turns golden brown' are "non-algorithmic" for the reason that terms such as 'a pinch', 'golden brown' are subject to ambiguity and hence violate the property of definiteness!

An algorithm may be represented using pictorial representations such as flow charts. An algorithm encoded in a programming language for implementation on a computer is called a *program*. However, there exists a school of thought which distinguishes between a program and an algorithm. The claim put forward by them is that programs need not exhibit the property of finiteness which algorithms insist upon and quote an operating systems program as a counter example. An operating system is supposed to be an 'infinite' program which terminates only when the system crashes! At all other times other than its execution, it is said to be in the 'wait' mode!

1.3 Development of an Algorithm

The steps involved in the development of an algorithm are as follows:

(i) Problem statement
(ii) Model formulation
(iii) Algorithm design
(iv) Algorithm correctness
(v) Implementation
(vi) Algorithm analysis
(vii) Program testing
(viii) Documentation

Once a clear statement of the problem is done, the model for the solution of the problem is to be formulated. The next step is to design the algorithm based on the solution model that is formulated. It is here that one sees the role of data structures. The right choice of the data structure needs to be made at the design stage itself since data structures influence the efficiency of the algorithm. Once the correctness of the algorithm is checked and the algorithm implemented, the most important step of measuring the performance of the algorithm is done. This is what is termed as ***algorithm analysis***. It can be seen how the use of appropriate data structures results in a better performance of the algorithm. Finally the program is tested and the development ends with proper documentation.

1.4 Data Structures and Algorithms

As was detailed in the previous section, the design of an *efficient* algorithm for the solution of the problem calls for the *inclusion of appropriate data structures*. A clear, unambiguous set of instructions following the properties of the algorithm alone does not contribute to the efficiency of the solution. It is essential that the data on which the problems need to work on are appropriately *structured* to suit the needs of the problem, thereby contributing to the efficiency of the solution.

For example, let us consider the problem of searching for a telephone number of a person, in the telephone directory. It is well known that searching for the telephone number in the directory is an easy task since the data is sorted according to the alphabetical order of the subscribers' names. All that the search calls for, is to turn over the pages until one reaches the page that is approximately closest to the subscriber's name and undertake a sequential search in the relevant page. Now, what if the telephone directory were to have its data arranged according to the order in which the subscriptions for telephones were received. What a mess would it be! One may need

to go through the entire directory–name after name, page after page in a sequential fashion until the name and the corresponding telephone number are retrieved!

This is a classic example to illustrate the significant role played by data structures in the efficiency of algorithms. The problem was retrieval of a telephone number. The algorithm was a simple search for the name in the directory and thereby retrieve the corresponding telephone number. In the first case since the data was appropriately structured (sorted according to alphabetical order), the search algorithm undertaken turned out to be efficient. On the other hand, in the second case, when the data was unstructured, the search algorithm turned out to be crude and hence inefficient.

For the design of efficient programs and for the solution of problems, it is essential that algorithm design goes hand in hand with appropriate data structures. (Refer Fig. 1.3.)

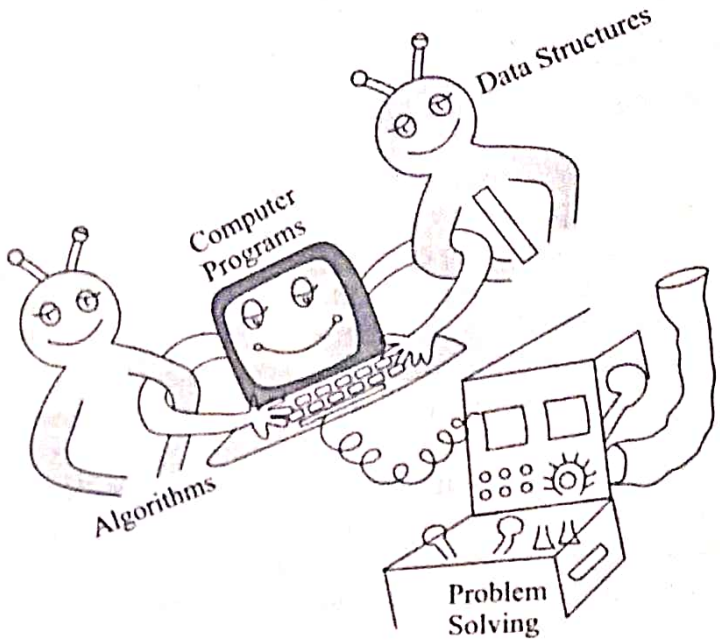


Fig. 1.3 *Algorithms and Data structures for efficient problem solving using computers*

Data Structure—Definition and Classification 1.5

Abstract data types

A *data type* refers to the type of values that variables in a programming language hold. Thus the data types of integer, real, character, Boolean which are inherently provided in programming languages are referred to as *primitive data types*.

A list of elements is called as a *data object*. For example, we could have a list of integers or list of alphabetical strings as data objects.

The data objects which comprise the data structure, and their fundamental operations are known as *Abstract Data Type (ADT)*. In other words, an ADT is defined as a set of data objects D defined over a domain L and supporting a list of operations O.

Example 1.3 Consider an ADT for the data structure of positive integers called POSITIVE_INTEGER defined over a domain of integers Z^+, supporting the operations of addition (ADD), subtraction(MINUS) and check if positive (CHECK_POSITIVE). The ADT is defined as follows:

$$L = Z^+,\ D = \{x \mid x \in L\},\ Q = \{ADD,\ MINUS,\ CHECK\_POSITIVE\}$$

A descriptive and clear presentation of the ADT is as follows:

```
Data objects
Set of all positive integers D
                D = {x|x ∈ L}, L = Z+
```

Operations

- Addition of positive integers INT1 and INT2 into RESULT
  ADD (INT1, INT2, RESULT)
- Subtraction of positive integers INT1 and INT2 into RESULT
  SUBTRACT (INT1, INT2, RESULT)
- Check if a number INT1 is a positive integer
  CHECK_POSITIVE(INT1) (Boolean function)

An ADT promotes *data abstraction* and focuses on *what* a data structure does rather than *how* it does. It is easier to comprehend a data structure by means of its ADT since it helps a designer to plan on the implementation of the data objects and its supportive operations in any programming language belonging to any paradigm such as procedural or object oriented or functional etc. Quite often it may be essential that one data structure calls for other data structures for its implementation. For example, the implementation of stack and queue data structures calls for their implementation using either arrays or lists.

While deciding on the ADT of a data structure, a designer may decide on the set of operations *O* that are to be provided, based on the application and accessibility options provided to various users making use of the ADT implementation.

The ADTs for various data structures discussed in the book are presented as box items in the respective chapters.

Classification

Figure 1.4 illustrates the classification of data structures. The data structures are broadly classified as ***linear data structures*** and ***non-linear data structures***. Linear data structures are uni-dimensional in structure and represent linear lists. These are further classified as ***sequential*** and ***linked representations***. On the other hand, non-linear data structures are two-dimensional representations of data lists. The individual data structures listed under each class have been shown in Fig. 1.4.

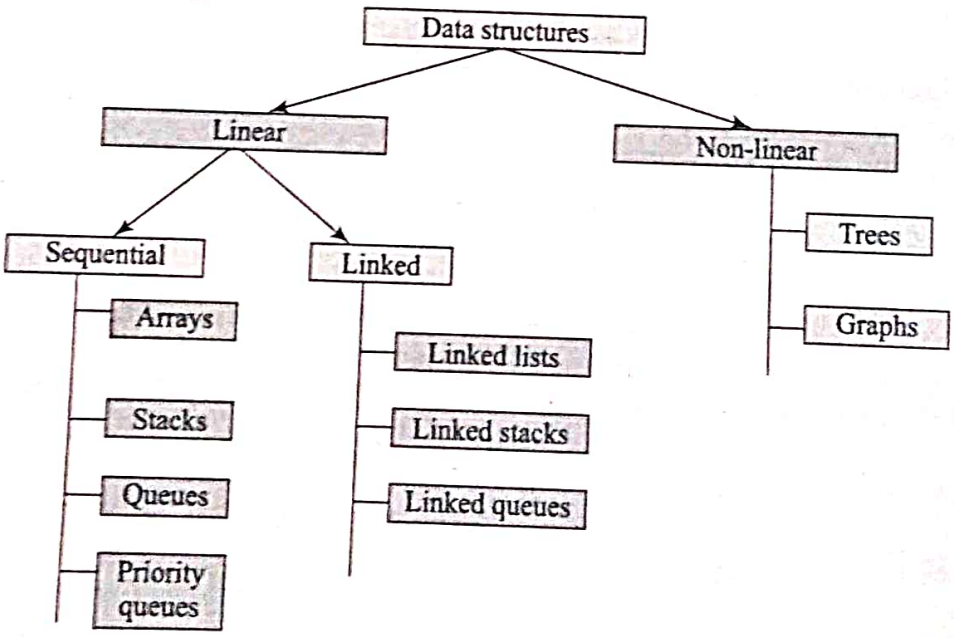


Fig. 1.4 *Classification of data structures*

CHAPTER 2

ANALYSIS OF ALGORITHMS

In the previous chapter we introduced the discipline of computer science from the perspective of problem solving. It was detailed how problem solving using computers calls not just for good algorithm design but also for the appropriate use of data structures to render them efficient. This chapter discusses methods and techniques to analyze the efficiency of algorithms.

Efficiency of Algorithms 2.1

When there is a problem to be solved it is probable that several algorithms crop up for its solution and therefore one is at a loss to know which one is the best. This raises the question of how one could decide on which among the algorithms is preferable and which among them is the best.

The performance of algorithms can be measured on the scales of ***time*** and ***space***. The former would mean looking for the fastest algorithm for the problem or that which performs its task in the minimum possible time. In this case the performance measure is termed ***time complexity***. The time complexity of an algorithm or a program is a function of the running time of the algorithm or program.

In the case of the latter, it would mean looking for an algorithm that consumes or needs limited memory space for its execution. The performance measure in such a case is termed *space* ***complexity***. The space complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run to completion. However, in this book our discussions would emphasize mostly on time complexities of the algorithms presented.

The time complexity of an algorithm can be computed either by an empirical or theoretical approach.

The ***empirical*** or ***posteriori testing*** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. That algorithm whose implementation yields the least time, is considered as the best among the candidate algorithmic solutions.

The *theoretical* or *apriori* approach calls for mathematically determining the resources such as time and space needed by the algorithm, as a function of a parameter related to the instances of the problem considered. A parameter that is often used is the size of the input instances. For example, for the problem of searching for a name in the telephone directory, an apriori approach could determine the efficiency of the algorithm used, in terms of the size of the telephone directory (i.e.) the number of subscribers listed in the directory. There exist algorithms for various classes of problems which make use of the number of basic operations such as additions or multiplications or element comparisons, as a parameter to determine their efficiency.

The disadvantage of posteriori testing is that it is dependent on various other factors such as the machine on which the program is executed, the programming language with which it is implemented and why, even on the skills of the programmer who writes the program code! On the other hand, the advantage of apriori analysis is that it is entirely machine, language and program independent.

The efficiency of a newly discovered algorithm over that of its predecessors can be better assessed only when they are tested over large input instance sizes. For smaller to moderate input instance sizes it is highly likely that their performances may break even. In the case of posteriori testing, practical considerations may permit testing the efficiency of the algorithm only on input instances of moderate sizes. On the other hand, apriori analysis permits study of the efficiency of algorithms on any input instance of any size.

2.2 Apriori Analysis

Let us consider a program statement, for example, $x = x + 2$ in a sequential programming environment. We do not consider any parallelism in the environment. Apriori estimation is interested in the following for the computation of efficiency:

(i) the number of times the statement is executed in the program, known as the ***frequency count*** of the statement, and

(ii) the time taken for a single execution of the statement.

To consider the second factor would render the estimation machine dependent since the time taken for the execution of the statement is determined by the machine instruction set, the machine configuration, and so on. Hence apriori analysis considers only the first factor and computes the efficiency of the program as a function of the ***total frequency count*** of the statements comprising the program. The estimation of efficiency is restricted to the computation of the total frequency count of the program.

Let us estimate the frequency count of the statement $x = x + 2$ occurring in the following three program segments (A, B, C):

Program segment A

```
...
x = x + 2;
...
```

Program segment B

```
...
for k = 1 to n do
x = x + 2;
end
...
```

Program segment C

```
...
for j = 1 to n do
for x = 1 to n do
x = x + 2;
end
end
```

The frequency count of the statement in the program segment A is 1. In the program segment B, the frequency count of the statement is n, since the **for** loop in which the statement is embedded executes n ($n \geq 1$) times. In the program segment C, the statement is executed n^2 ($n \geq 1$) times since the statement is embedded in a nested **for** loop, executing n times each.

In apriori analysis, the frequency count f_i of each statement i of the program is computed and summed up to obtain the total frequency count $T = \sum_i f_i$.

The computation of the total frequency count of the program segments A, B, and C are shown in Tables 2.1, 2.2 and 2.3. It is well known that the opening statement of a **for** loop such as `for i = low_index` to `up_index` executes `((up_index -low_index +1) +1` times and the statements within the loop are executed `(up_index-low_index)+1` times. In the

Table 2.1 *Total frequency count of program segment A*

Program statements	Frequency count
...	
`x = x + 2;`	1
...	
Total frequency count	1

Table 2.2 *Total frequency count of program segment B*

Program statements	Frequency count
...	
for `k = 1 to n` **do**	$(n + 1)$
`x = x + 2;`	n
end	n
...	
Total frequency count	$3n + 1$

Table 2.3 *Total frequency count of program segment C*

Program statements	Frequency count
...	
for `j = 1 to n` **do**	$(n + 1)$
for `k = 1 to n` **do**	$\sum_{j=1}^{n}(n+1) = (n+1)n$
`x = x + 2;`	n^2
end	$\sum_{j=1}^{n} n = n^2$
end	n
...	
Total frequency count	$3n^2 + 3n + 1$

case of nested **for** loops, it is easier to compute the frequency counts of the embedded statements making judicious use of the following fundamental mathematical formulae:

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Observe in Table 2.3 how the frequency count of the statement `for k = 1 to n do` is computed as

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

The total frequency counts of the program segments A, B and C given by 1, $(3n + 1)$ and $3n^2+3n+1$ respectively, are expressed as $O(1)$, $O(n)$ and $O(n^2)$ respectively. These notations mean that the orders of the magnitude of the total frequency counts are proportional to 1, n and n^2 respectively. The notation O has a mathematical definition as discussed in Sec. 2.3. These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner, one could also discuss about the space complexities of a program which is the amount of memory they require for their execution and its completion. The space complexities can also be expressed in terms of mathematical notations.

2.3 Asymptotic Notations

Apriori analysis employs the following notations to express the time complexity of algorithms. These are termed ***asymptotic notations*** since they are meaningful approximations of functions that represent the time or space complexity of a program.

Definition 2.1: $f(n) = O(g(n))$ (read as f of n is "big oh" of g of n), if there exists a positive integer n_0 and a positive number C such that $|f(n)| \leq C|g(n)|$, for all $n \geq n_0$.

Example

$f(n)$	$g(n)$	
$16n^3 + 78n^2 + 12n$	n^3	$f(n) = O(n^3)$
$34n - 90$	n	$f(n) = O(n)$
56	1	$f(n) = O(1)$

Here $g(n)$ is the upper bound of the function $f(n)$.

Definition 2.2: $f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number C such that $|f(n)| \geq C|g(n)|$, for all $n \geq n_0$.

Example

$f(n)$	$g(n)$	
$16n^3 + 8n^2 + 2$	n^3	$f(n) = \Omega(n^3)$
$24n + 9$	n	$f(n) = \Omega(n)$

Here $g(n)$ is the lower bound of the function $f(n)$.

Definition 2.3: $f(n) = \Theta(g(n))$ (read as f on n is theta of g of n) if there exist two positive constants c_1 and c_2, and a positive integer n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$.

Example

$f(n)$	$g(n)$	
$28n + 9$	n	$f(n) = \Theta(n)$ since $f(n) > 28n$ and $f(n) \leq 37n$ for $n \geq 1$
$16n^2 + 30n - 90$	n^2	$f(n) = \Theta(n^2)$
$7.2^n + 30n$	2^n	$f(n) = \Theta(2^n)$

From the definition it implies that the function $g(n)$ is both an upper bound and a lower bound for the function $f(n)$ for all values of n, $n \geq n_0$. This means that $f(n)$ is such that, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Definition 2.4: $f(n) = o(g(n))$ (read as f of n is "little oh" of g of n) if $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

Example

$f(n)$	$g(n)$	
$18n + 9$	n^2	$f(n) = o(n^2)$ since $f(n) = O(n^2)$ and $f(n) \neq \Omega(n^2)$ however, $f(n) \neq O(n)$.

Time Complexity of an Algorithm Using *O* Notation 2.4

O notation is widely used to compute the time complexity of algorithms. It can be gathered from its definition (Definition 2.1) that if $f(n) = O(g(n))$ then $g(n)$ acts as an upper bound for the function $f(n)$. $f(n)$ represents the computing time of the algorithm. When we say the time complexity of the algorithm is $O(g(n))$, we mean that its execution takes a time that is no more than constant times $g(n)$. Here n is a parameter that characterizes the input and/or output instances of the algorithm.

Algorithms reporting $O(1)$ time complexity indicate ***constant running time***. The time complexities of $O(n)$, $O(n^2)$ and $O(n^3)$ are called ***linear, quadratic*** and ***cubic*** time complexities respectively. ***O(logn)*** time complexity is referred to as ***logarithmic***. In general, time complexities of the type $O(n^k)$ are called ***polynomial time complexities***. In fact it can be shown that a polynomial $A(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 = O(n^m)$ (see Illustrative Problem 2.2). Time complexities such as $O(2^n)$, $O(3^n)$, in general $O(k^n)$ are called as ***exponential time complexities***.

Algorithms which report $O(\log n)$ time complexity are faster for sufficiently large n, than if they had reported $O(n)$. Similarly $O(n.\log n)$ is better than $O(n^2)$, but not as good as $O(n)$. Some of the commonly occurring time complexities in their ascending orders of magnitude are listed below:

$$O(1) \leq O(\log n) \leq O(n) \leq O(n.\log n) \leq O(n^2) \leq O(n^3) \leq O(2n)$$

Polynomial Vs Exponential Algorithms 2.5

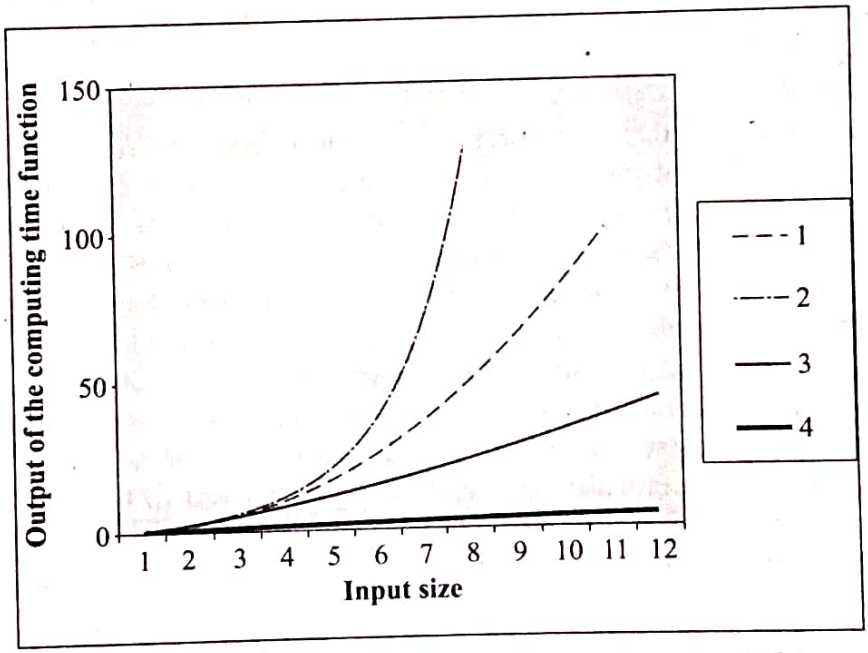
If n is the size of the input instance, then the number of operations for polynomial algorithms are of the form $P(n)$ where P is a polynomial. In terms of O notation, polynomial algorithms have time complexities of the form $O(n^k)$, where k is a constant.

In contrast, in the exponential algorithms the number of operations are of the form k^n. In terms of O notation, exponential algorithms have time complexities of the form $O(k^n)$, where k is a constant.

It is clear from the inequalities listed above that polynomial algorithms are a lot more efficient than exponential algorithms. From Table 2.4 it is seen that exponential algorithms can quickly get beyond the capacity of any sophisticated computer due to their rapid growth rate (Refer Fig. 2.1). Here, it is assumed that the computer takes 1 microsecond per operation. While the time complexity functions of n^2, n^3 can be executed in a reasonable time, one can never hope to finish the execution of exponential algorithms even if the fastest computer were to be employed. Thus if one were to find an algorithm for a problem that reduces from exponential to polynomial time then it is indeed a great accomplishment!

Table 2.4 *Comparison of polynomial and exponential algorithms*

Time complexity function \ Size	10	20	50
n^2	10^{-4} sec	4×10^{-4} sec	25×10^{-4} sec
n^3	10^{-3} sec	8×10^{-3} sec	125×10^{-3} sec
2^n	10^{-3} sec	1 sec	35 years
3n	6×10^{-2} sec	58 mins	2×10^3 centuries



1 : n^2 2: 2^n 3: $n.\log_2 n$ 4: $\log_2 n$

Fig. 2.1 *Growth rate of some computing time functions*

Average, Best and Worst Case Complexities 2.6

The time complexity of an algorithm is dependent on parameters associated with the input/ output instances of the problem. Very often the running time of the algorithm is expressed as a

function of the input size. In such a case it is fair enough to presume that larger the input size of the problem instances the larger is its running time. But such is not the case always. There are problems whose time complexity is dependent not just on the size of the input but on the nature of the input as well. Example 2.1 illustrates this point.

Example 2.1 **Algorithm:** To sequentially search for the first-occurring even number in the list of numbers given.

Input 1: –1, 3, 5, 7, –5, 7, 11, –13, 17, 71, 21, 9, 3, 1, 5, –23, –29, 33, 35, 37, 40

Input 2: 6, 17, 71, 21, 9, 3, 1, 5, –23, 3, 64, 7, –5, 7, 11, 33, 35, 37, –3, –7, 11

Input 3: 71, 21, 9, 3, 1, 5, –23, 3, 11, 33, 36, 37, –3, –7, 11, –5, 7, 11, –13, 17, 22

Let us determine the efficiency of the algorithm for the input instances presented in terms of the number of comparisons done before the first occurring even number is retrieved. Observe that all three input instances are of the same size.

In the case of Input 1, the first occurring even number occurs as the last element in the list. The algorithm would require 21 comparisons, equivalent to the size of the list, before it retrieves the element. On the other hand, in the case of Input 2 the first occurring even number shows up as the very first element of the list thereby calling for only one comparison before it is retrieved! If Input 2 is the ***best*** possible case that can happen for the quickest execution of the algorithm, then Input 1 is the ***worst*** possible case that can happen when the algorithm takes the longest possible time to complete. Generalizing, the time complexity of the algorithm in the best possible case would be expressed as $O(1)$ and in the worst possible case would be expressed as $O(n)$ where n is the size of the input.

This justifies the statement that the running time of algorithms are not just dependent on the size of the input but also on its nature. That input instances (or instances) for which the algorithm takes the maximum possible time is called the ***worst case*** and the time complexity in such a case is referred to as the ***worst case time complexity***. That input instances for which the algorithm takes the minimum possible time is called the *best case* and the time complexity in such a case is referred to as the ***best case time complexity***. All other input instances which are neither of the two are categorized as the ***average cases*** and the time complexity of the algorithm in such cases is referred to as the ***average case complexity***. Input 3 is an example of an average case since it is neither the best case nor the worst case. By and large, analyzing the average case behaviour of algorithms is harder and mathematically involved when compared to their worst case and best case counterparts. Also such an analysis can be misleading if the input instances are not chosen at random or appropriately to cover all possible cases that may arise when the algorithm is put to practice.

Worst case analysis is appropriate when the response time of the algorithm is critical. For example, in the case of a nuclear power plant controller, it is critical to know of the maximum limit of the system response time regardless of the input instance that is to be handled by the system. The algorithms designed cannot have a running time that exceeds this response time limit.

On the other hand in the case of applications where the input instances may be wide and varied and there is no knowing beforehand of the kind of input instance that has to be worked on, it is prudent to choose algorithms with good average case behaviour.

2.7 Analyzing Recursive Programs

Recursion is an important concept in computer science. Many algorithms can best be described in terms of recursion.

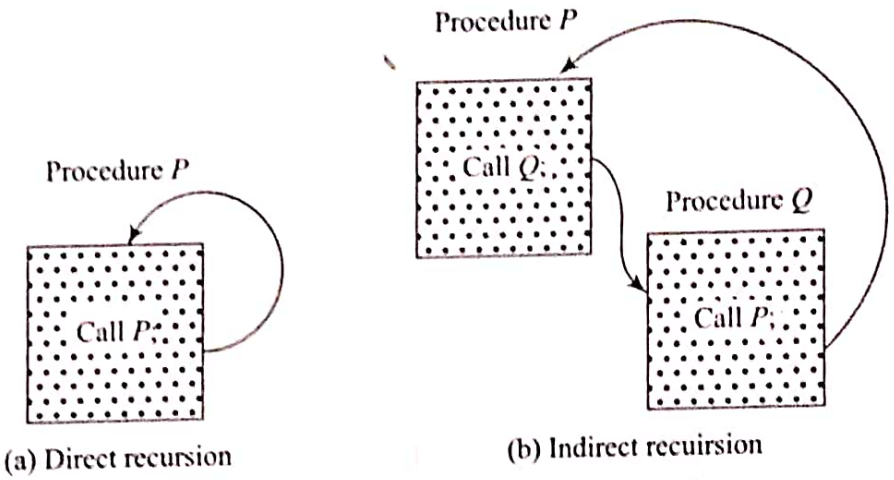
Recursive procedures

If P is a procedure containing a call statement to itself (Fig. 2.2(a)) or to another procedure that results in a call to itself (Fig. 2.2(b)), then the procedure P is said to be a ***recursive procedure***. In the former case it is termed ***direct recursion*** and in the latter case it is termed ***indirect recursion***.

Extending the concept to programming can yield program functions or programs themselves that are recursively defined. In such cases they are referred to as ***recursive functions*** and ***recursive programs*** respectively.

Extending the concept to mathematics would yield what are called ***recurrence relations***.

In order that the recursively defined function may not run into an infinite loop it is essential that the following properties are satisfied by any recursive procedure:



(a) Direct recursion (b) Indirect recuirsion

Fig. 2.2 *Skeletal recursive procedures*

(i) There must be criteria, one or more, called the *base criteria* or simply *base case(s)*, where the procedure does not call itself either directly or indirectly.
(ii) Each time the procedure calls itself directly or indirectly, it must be closer to the base criteria.

Example 2.2 illustrates a recursive procedure and Example 2.3 a recurrence relation. Example 2.4 describes the Tower of Hanoi puzzle which is a classic example for the application of recursion and recurrence relation.

Example 2.2 A recursive procedure to compute factorial of a number n is shown below:

$n! = 1$, if $n = 1$ (base criterion)
$n! = n.\ (n-1)!$, if $n > 1$

Note the recursion in the definition of factorial function(!). $n!$ calls $(n-1)!$ for its definition. A pseudo-code recursive function for computation of $n!$ is shown below:

```
function factorial(n)
1-2.   if (n = 1) then factorial = 1;
or     else
3.     factorial = n* factorial(n-1);
and    end factorial.
```

Example 2.3 A recurrence relation $S(n)$ is defined as below:

$S(n) = 0$, if $n = 1$ (base criterion)
$= S(n/2) + 1$, if $n > 1$

Example 2.4 *The Tower of Hanoi puzzle*

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 188
There are three Pegs, Source (S), Intermediary (I) and Destination (D). Peg S contains a set o
disks stacked to resemble a tower, with the largest disk at the bottom and the smallest at the to
Figure 2.3 illustrates the initial configuration of the Pegs for 6 disks. The objective is to transfe
the entire tower of disks in Peg S, to Peg D, maintaining the same order of the disks. Also onl
one disk can be moved at a time and never can a larger disk be placed on a smaller disk durin
the transfer. The I Peg is for intermediate use during the transfer.
A simple solution to the problem, for $N = 3$ disk is given by the following transfers of disks:

1. Transfer disk from Peg S to Peg D
2. Transfer disk from Peg S to Peg I
3. Transfer disk from Peg D to Peg I
4. Transfer disk from Peg S to Peg D
5. Transfer disk from Peg I to Peg S
6. Transfer disk from Peg I to Peg D
7. Transfer disk from Peg S to Peg D

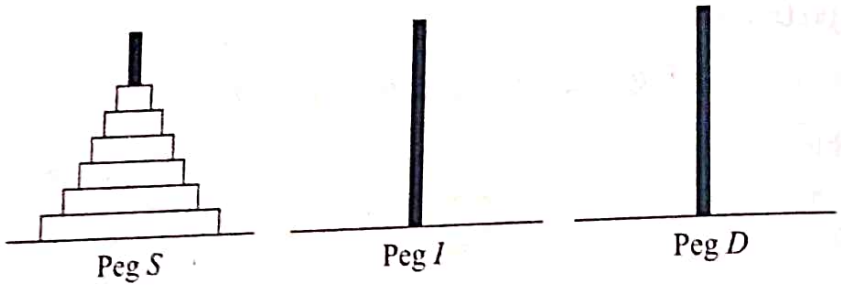
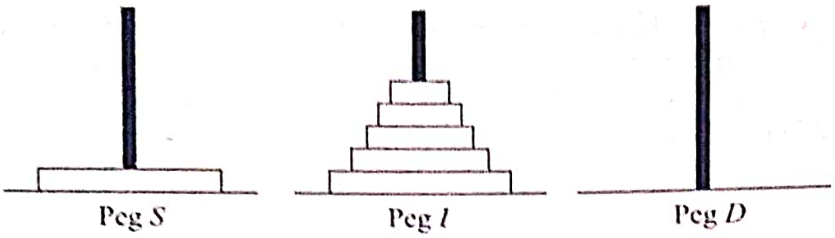


Fig. 2.3 *Tower of Hanoi puzzle (initial configuration)*

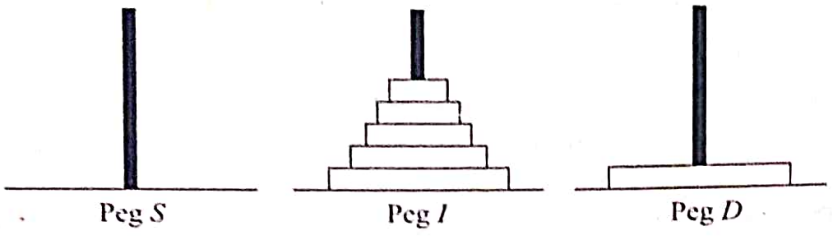
The solution to the puzzle calls for an application of recursive functions and recurrence relation
A skeletal recursive procedure for the solution of the problem for N number of disks, is a
follows:

1. Move the top N-1 disks from Peg S to Peg I (using D as an intermediary Peg)
2. Move the bottom disk from Peg S to Peg D
3. Move N-1 disks from Peg I to Peg D (using Peg S as an intermediary Peg)

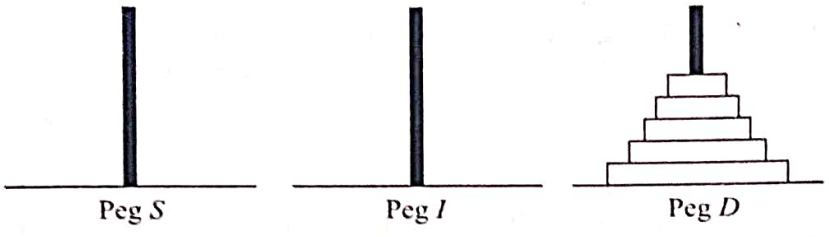
A pictorial representation of the skeletal recursive procedure for $N = 6$ disks is shown in Fig. 2.
Function `TRANSFER` illustrates the recursive function for the solution of the problem.



Move the top (N-1) disks (5 disks) from
Peg S to Peg I using Peg D as the intermediate Peg



Move the bottom disk from Peg S to Peg D



Move (N-1) disks from Peg I to Peg D using Peg S as the intermediate Peg

Fig. 2.4 *Pictorial representation of the skeletal recursive procedure for Tower of Hanoi puzzle*

```
function TRANSFER(N, S, I, D)
/* N disks are to be transferred from peg S to peg D with
peg I as the intermediate peg*/
if N is 0 then exit();
else
{TRANSFER(N-1, S, D, I); /* transfer N-1 disks from peg S to
peg I with peg D as the intermediate peg*/
Transfer disk from S to D; /* move the disk which is the last
and the largest disk, from peg S to peg D*/
TRANSFER(N-1, I, S, D); /* transfer N-1 disks from peg I to
peg D with peg S as the intermediate peg*/}
end TRANSFER.
```

Apriori analysis of recursive functions

The apriori analysis of recursive functions is different from that of iterative functions. In the latter case as was seen in Sec. 2.2, the total frequency count of the programs were computed before approximating them using mathematical functions such as O. In the case of recursive functions we first formulate recurrence relations that define the behaviour of the function. The solution of the recurrence relation and its approximation using the conventional O or any other notation yields the resulting time complexity of the program.

To frame the recurrence relation, we associate an unknown time function $T(n)$ where n measures the size of the arguments to the procedure. We then get a recurrence relation for $T(n)$ in terms of $T(k)$ for various values of k.

Example 2.5 illustrates obtaining the recurrence relation for the recursive factorial function FACTORIAL(n) shown in Example 2.2.

Example 2.5 Let $T(n)$ be the running time of the recursive function FACTORIAL(n). The running times of lines 1 and 2 is $O(1)$. The running time for line 3 is given by $O(1) + T(n-1)$. Here $T(n-1)$ is the time complexity of the call to the recursive function FACTORIAL(n-1). Thus for some constants c, d,

$$\begin{aligned} T(n) &= c + T(n-1), && \text{if } n > 1 \\ &= d, && \text{if } n \leq 1 \end{aligned}$$

Example 2.6 derives the recurrence relation for the Tower of Hanoi puzzle.

Example 2.6 The recurrence relation for the Tower of Hanoi puzzle is derived as follows. Let $T(N)$ be the minimum number of transfers that are needed to solve the puzzle with N disks. From the function TRANSFER it is evident that for $N = 0$, no disks are transferred. Again for $N > 0$, two recursive calls each enabling the transfer of $(N-1)$ disks, and a single transfer of the last (largest) disk from peg S to peg D are done. Thus the recurrence relation is given by,

$$\begin{aligned} T(N) &= 0, && \text{if } N = 0 \\ &= 2.\ T(N-1) + 1, && \text{if } N > 0 \end{aligned}$$

Now what remains to be done is to solve the recurrence relation, in other words to solve for $T(n)$. Such a solution where $T(n)$ expresses itself in a form where no T occurs on the right side is termed as a ***closed form solution***, in conventional mathematics.

The general method of solution is to repeatedly replace terms $T(k)$ occurring on the right side of the recurrence relation, by the relation itself with appropriate change of parameters. The substitutions continue until one reaches a formula in which T does not appear on the right side. Quite often at this stage, it may be essential to sum a series which could be either an arithmetic progression or a geometric progression or some such series. Even if we cannot obtain a sum exactly, we could work to obtain at least a close upper bound on the sum, which could serve to act as an upper bound for $T(n)$.

Example 2.7 illustrates the solution of the recurrence relation for the function FACTORIAL(n) discussed in Example 2.5 and Example 2.8 illustrates the solution of the recurrence relation for the Tower of Hanoi puzzle, discussed in Example 2.6.

Example 2.7 Solution of the recurrence relation

$$\begin{aligned} T(n) &= c + T(n-1), && \textit{if } n > 1 \\ &= d, && \textit{if } n \leq 1 \end{aligned}$$

yields the following steps.

$$\begin{aligned} T(n) &= c + T(n-1) && \text{...(step 1)} \\ &= c + (c + T(n-2) \\ &= 2c + T(n-2) && \text{...(step 2)} \\ &= 2c + (c + T(n-3) \\ &= 3c + T(n-3) && \text{...(step 3)} \end{aligned}$$

In the *kth* step the recurrence relation is transformed as

$$T(n) = k \cdot c + T(n-k), \quad if \quad n > k, \qquad \text{...(step } k\text{)}$$

Finally when $(k = n - 1)$, we obtain

$$\begin{aligned} T(n) &= (n-1) \cdot c + T(1), \qquad \text{...(step } n-1\text{)} \\ &= (n-1)c + d \\ &= O(n) \end{aligned}$$

Observe how the recursive terms in the recurrence relation are replaced so as to move the relation closer to the base criterion viz., $T(n) = 1$, $n \leq 1$. The approximation of the closed form solution obtained viz., $T(n) = (n-1)c + d$ yields $O(n)$.

Example 2.8 Solution of the recurrence relation for the Tower of Hanoi puzzle,

$$\begin{aligned} T(N) &= 0, && \text{if } N = 0 \\ &= 2 \cdot T(N-1) + 1, && \text{if } N > 0 \end{aligned}$$

yields the following steps.

$$\begin{aligned} T(N) &= 2 \cdot T(N-1) && \text{...(step 1)} \\ &= 2 \cdot (2.T(N-2) + 1) + 1) \\ &= 2^2\, T(N-2) + 2 + 1 && \text{...(step 2)} \\ &= 2^2(2 \cdot T(N-3) + 1) + 2 + 1 \\ &= 2^3 \cdot T(N-3) + 2^2 + 2 + 1 && \text{..... (step 3)} \end{aligned}$$

In the *kth* step the recurrence relation is transformed as

$$T(N) = 2^k T(N-k) + 2^{(k-1)} + 2^{(k-2)} + \ldots\ 2^3 + 2^2 + 2 + 1, \qquad \text{.(step } k\text{)}$$

Finally when $(k = N)$, we obtain

$$\begin{aligned} T(N) &= 2^N T(0) + 2^{(N-1)} + 2^{(N-2)} + \ldots\ 2^3 + 2^2 + 2 + 1 && \text{...(step } N\text{)} \\ &= 2^N \cdot 0 + (2^N - 1) \\ &= 2^N - 1 \\ &= O(2^N) \end{aligned}$$

Summary

- When several algorithms can be designed for the solution of a problem, there arises the need to determine which among them is the best. The efficiency of a program or an algorithm is measured by computing its time and/or space complexities. The time complexity of an algorithm is a function of the running time of the algorithm and the space complexity is a function of the space required by it to run to completion.
- The time complexity of an algorithm can be measured using Apriori analysis or Posteriori testing. While the former is a theoretical approach that is general and machine independent, the latter is completely machine dependent.

CHAPTER

15 SEARCHING

15.1 Introduction

Search (or *Searching*) is a common place occurrence in every day life. Searching for a book in the library, searching for a subscriber's telephone number in the telephone directory, searching for one's name in the electoral rolls are some examples.

In the discipline of computer science, the problem of search has assumed enormous significance. It spans a variety of applications, rather disciplines, beginning from searching for a key in a list of data elements to searching for a solution to a problem in its *search space*. Innumerable problems exist where one searches for patterns - images, voice, text, hyper text, photographs etc., in a repository of data or patterns, for the solution of the problems concerned. A variety of search algorithms and procedures appropriate to the problem and the associated discipline exist in the literature.

In this chapter we enumerate search algorithms pertaining to the problem of looking for a key K in a list of data elements. When the list of data elements is represented as a linear list the search procedures of *linear search* or *sequential search, transpose sequential search, interpolation search, binary search* and *Fibonacci search* are applicable. When the list of data elements is represented using non linear data structures such as binary search trees or AVL trees or B trees etc., the appropriate *tree search* techniques unique to the data structure representation may be applied. Hash tables also promote efficient searching. Search techniques such as *breadth first search* and *depth first search* are applicable on graph data structures. In the case of data representing an index of a file or a group of ordered elements, *indexed sequential search* may be employed. This chapter discusses all the above mentioned search procedures.

15.2 Linear Search

A *linear search* or *sequential search* is one where a key K is searched for, in a linear list L of data elements. The list L is commonly represented using a sequential data structure such as an array. If L is ordered then the search is said to be an *ordered linear search* and if L is unordered then it is said to be *unordered linear search*.

Ordered linear search

Let $L = \{K_1, K_2, K_3, \ldots K_n\}$, $K_1 < K_2 < \ldots K_n$ be the list of ordered elements. To search for a key K in the list L, we undertake a linear search comparing K with each of the K_i. So long as $K > K_i$ comparing K with the data elements of the list L progresses. However, if $K \leq K_i$, then if $K = K_i$ then the search is done, other wise the search is unsuccessful implying K does not exist in the list L. It is easy to see how ordering the elements renders the search process to be efficient.

Algorithm 15.1 illustrates the working of ordered linear search.

Algorithm 15.1: Procedure for ordered linear search

```
procedure LINEAR_SEARCH_ORDERED(L, n, K)
                /* L[0:n-1] is a linear ordered list of data elements. K
                is the key to be searched for in the list. In case of
                unsuccessful search, the procedure prints the message "KEY
                not found" otherwise prints "KEY found" and returns the
                index i*/
i = 0;
while (( i < n) and (K > L[i])) do        /* search for X down the list*/
i = i + 1;
endwhile
if ( K = L[i]) then { print (" KEY found");
                      return (i);}   /*  Key K found. Return index i */
else
    print (" KEY not found");
end LINEAR_SEARCH_ORDERED.
```

Example 15.1 Consider an ordered list of elements $L[0:5] = \{16, 18, 56, 78, 90, 100\}$. Let us search for the key $K = 78$. Since K is greater than 16, 18, and 56, the search terminates at the fourth element when $K \leq (L[3] = 78)$ is true. At this point, since $K = L[3] = 78$, the search is successfully done. However in the case of searching for key $K = 67$, the search progresses until the condition $K \leq (L[3] = 78)$ is reached. At this point since $K \neq L[3]$, we deem the search to be unsuccessful.

Ordered linear search reports a time complexity of $O(n)$ in the worst case and $O(1)$ in the best case, in terms of key comparisons. However, it is essential that the elements are ordered before the search process is undertaken.

Unordered linear search

In this search, a key K is looked for in an unordered linear list $L = \{K_1, K_2, K_3, \ldots K_n\}$ of data elements. The method obviously of the 'brute force' kind, merely calls for a sequential search down the list looking for the key K.

Algorithm 15.2 illustrates the working of the unordered linear search.

Algorithm 15.2: Procedure unordered linear search

```
procedure LINEAR_SEARCH_UNORDERED(L, n, K)
                /* L[0:n-1] is a linear unordered list of data elements.
                K is the key to be searched for in the list. In case
                of unsuccessful search, the procedure prints the message
                "KEY not found" otherwise prints "KEY found" and returns
                the index i*/
```

```
i ← 0;
while (( i < n) and ( L[i] ≠ K)) do      /* search for X down the list*/
   i ← i + 1;
endwhile
if ( L[i]= K) then { print (" KEY found");
                     return (i);}        /*  Key K found. Return index i */
else
   print (" KEY not found");
end LINEAR_SEARCH_UNORDERED.
```

Example 15.2 Consider an unordered list $L[0:5] = \{ 23, 14, 98, 45, 67, 53\}$ of data elements. Let us search for the key $K = 53$. Obviously the search progresses down the list comparing key K with each of the elements in the list until it finds it as the last element in the list. In the case of searching for the key $K = 110$, the search progresses but falls off the list thereby deeming it to be an unsuccessful search.

Unordered linear search reports a worst case complexity of $O(n)$ and a best case complexity of $O(1)$ in terms of key comparisons. However, its average case performance in terms of key comparisons can only be inferior to that of ordered linear search.

$O(n/2)$ – average

Binary Search 15.5

In the previous section we discussed interpolation search which works on ordered lists and reports an average case complexity of $O(log_2\ log_2\ n)$. Another efficient search technique that operates on ordered lists is the *binary search* also known as *logarithmic search* or *bisection*.

A binary search searches for a key K in an ordered list $L = \{K_1, K_2, K_3, \ldots K_n\}, K_1 < K_2 < \ldots K_n$ of data elements, by halving the search list with each comparison until the key is either found or not found. The key K is first compared with the median element of the list viz., K_{mid}. For a sublist $\{K_i, K_{i+1}, K_{i+2}, \ldots K_j\}$, K_{mid} is obtained as the key occurring at the position *mid* which is computed as $mid = \left\lfloor \frac{(i+j)}{2} \right\rfloor$. The comparison of K with K_{mid} yields the following cases:

If $(K = K_{mid})$ then the binary search is done.

If $(K < K_{mid})$ then continue binary search in the sub list $\{K_i, K_{i+1}, K_{i+2}, \ldots K_{mid-1}\}$

If $(K > K_{mid})$ then continue binary search in the sub list $\{K_{mid+1}, K_{mid+2}, K_{mid+3}, \ldots K_j\}$

During the search process, each comparison of key K with K_{mid} of the respective sub lists results in the halving of the list. In other words with each comparison the search space is reduced to half its original length. It is this characteristic that renders the search process to be efficient. Contrast this with a sequential list where the entire list is involved in the search!

Binary search adopts the *Divide-and-Conquer* method of algorithm design. Divide-and-Conquer is an algorithm design technique where to solve a given problem, the problem is first recursively divided (*Divide*) into sub-problems (smaller problem instances). The sub-problems that are small enough are easily solved (*Conquer*) and the solutions combined to obtain the solution to the whole problem. Divide-and-Conquer has turned out to be a successful algorithm design technique with regard to many problems. In the case of binary search, the divide-and-conquer aspect of the technique breaks the list (problem) into two sub lists (sub-problems). However, the key is searched for only in one of the sublists hence with every division a portion of the list gets discounted.

Algorithm 15.5 illustrates a recursive procedure for binary search.

Decision tree for binary search

The binary search for a key K in the ordered list $L = \{K_1, K_2, K_3, \ldots K_n\}, K_1 < K_2 < \ldots K_n$ traces a binary decision tree. Figure 15.1 illustrates the decision tree for $n = 15$. The first element to be compared with K in the list $L = \{K_1, K_2, K_3, \ldots K_{15}\}$ is K_8 which becomes the root of the decision tree. If $K < K_8$ then the next element to be compared is K_4 which is the left child of the decision tree. For the other cases of comparisons it is easy to trace the tree by making use of the following characteristics:

(i) the indexes of the left and the right child nodes differ by the same amount from that of the parent node.

For example, in the decision tree shown in Fig. 15.1 the left and right child nodes of the node K_{12}, viz., K_{10} and K_{14} differ from their parent key index by the same amount.

This characteristic renders the search process to be *uniform* and therefore binary search is also termed as *uniform binary search*.

(ii) for n elements where $n = 2^t - 1$, the difference in the indexes of a parent node and its child nodes follows the sequence $2^0, 2^1, 2^2 \cdots$ from the leaf upwards.

For example, in Fig. 15.1 where $n = 15 = 2^4 - 1$, the difference in index of all the leaf nodes from their respective parent nodes is 2^0. The difference in index of all the nodes in level 3 from their respective parent nodes is 2^1 and so on.

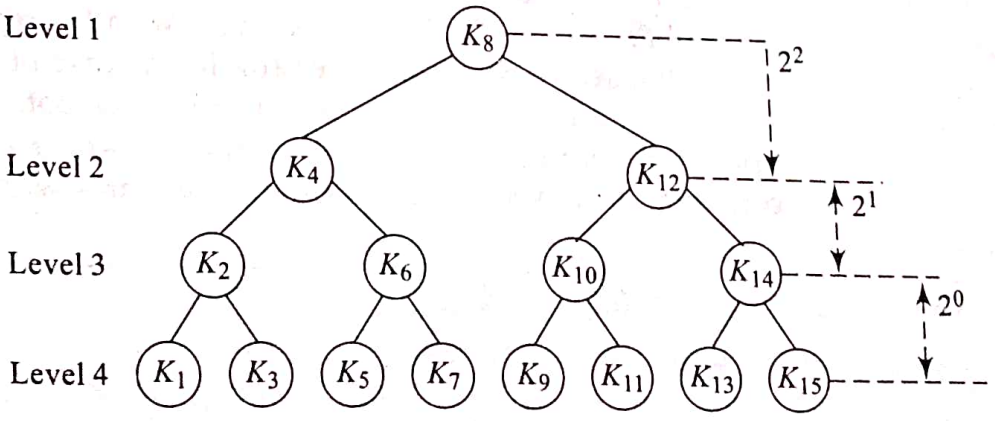


Fig. 15.1 *Decision tree for binary search*

Example 15.5 Consider an ordered list $L = \{K_1, K_2, K_3, \ldots K_{15}\} = \{12, 21, 34, 38, 45, 49, 67, 69, 78, 79, 82, 87, 93, 97, 99\}$. Let us search for the key $K = 21$ in the list L. The search process is illustrated in Fig. 15.2. K is first compared with $K_{mid} = K_{\lfloor \frac{1+15}{2} \rfloor} = K_8 = 69$. Since $K < K_{mid}$, the search continues in the sublist {12, 21, 34, 38, 45, 49, 67}. Now, K is compared with $K_{mid} = K_{\lfloor \frac{1+7}{2} \rfloor} = K_4 = 38$. Again $K < K_{mid}$, shrinks the search list to {12, 21, 34}. Now finally when K is compared with $K_{mid} = K_{\lfloor \frac{1+3}{2} \rfloor} = K_2 = 21$, the search is done. Thus in three comparisons we are able to search for the key K =21.

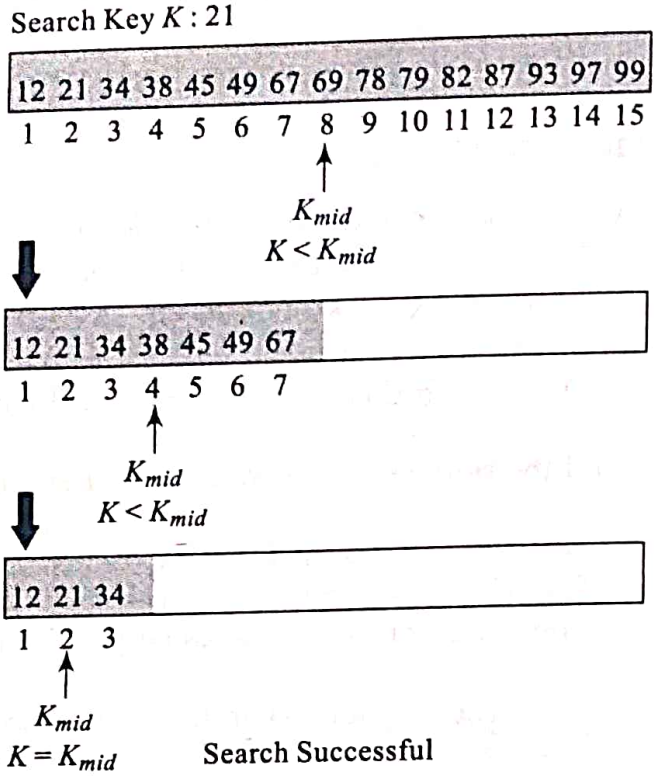


Fig. 15.2 *Binary search process (Example 15.5)*

Let us now search for the key $K = 75$. Proceeding in a similar manner, K is first compared with $K_8 = 69$. Since $K > K_8$, the search list is reduced to { 78, 79, 82, 87, 93, 97, 99}. Now $K < (K12 = 87)$, hence the search list is reduced further to { 78, 79, 82}. Comparing K with K_{10} reduces the search list to {78} which obviously yields the search to be unsuccessful. In the case of Algorithm 15.5, at this step of the search, the recursive call to `BINARY_SEARCH` would have both `low = high = 9`, resulting in `mid = 9`. Comparing K with K_{mid} results in the call to `binary_search(L, 9, 8, K)`. Since `(low > high)` condition is satisfied, the algorithm terminates with the `'key not found'` message.

Algorithm 15.5: Procedure for binary search

```
procedure binary_search(L, low, high, K)
                /*  L[low:high] is a linear ordered sublist of data
                elements. Initially low is set to 1 and high to n.   K
                is the key to be searched in the list.  */
```

```
if ( low > high)    then    {binary_search =0;
                             print("Key not found");
                             exit();}
else
    {
                                                /*  key K  not found*/
        mid = ⌊(low + high)/2⌋;
    case
      : K = L[mid]: { print ("Key found");
                      binary_search=mid;
                      return L[mid];}
      : K < L[mid]: binary_search = binary_search(L, low, mid-1, K);
      : K > L[mid]: binary_search = binary_search(L, mid+1, high, K);
    endcase
    }
end binary_search.
```

Considering the decision tree associated with binary search, it is easy to see that in the worst case the number of comparisons needed to search for a key would be determined by the height of the decision tree and is therefore given by $O(\log_2 n)$.

15.6 Fibonacci Search

The Fibonacci number sequence is given by { 0, 1, 1, 2, 3, 5, 8, 13, 21,...} and is generated by the following recurrence relation:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2}$$

It is interesting to note that the Fibonacci sequence finds an application in a search technique termed *Fibonacci search*. While binary search selects the median of the sublist as its next element for comparison, the Fibonacci search determines the next element of comparison as dictated by the Fibonacci number sequence.

Fibonacci search works only on ordered lists and for convenience of description we assume that the number of elements in the list is one less than a Fibonacci number, (i.e.) $n = F_k - 1$. It is easy to follow Fibonacci search once the decision tree is traced, which otherwise may look mysterious!

Decision tree for Fibonacci search

The decision tree for Fibonacci search satisfies the following characteristic:

If we consider a grandparent, parent and its child nodes and if the difference in index between the grandparent and the parent is F_k then

(i) if the parent is a left child node then the difference in index between the parent and its child nodes is F_{k-1}, whereas

(ii) if the parent is a right child node then the difference in index between the parent and the child nodes is F_{k-2}.

Let us consider an ordered list $L = \{K_1, K_2, K_3, \ldots K_n\}, K_1 < K_2 < \ldots K_n$ where $n = F_k - 1$. The Fibonacci search decision tree for $n = 20$ where $20 = (F_8 - 1)$ is shown in Fig. 15.3.

The root of the decision tree which is the first element in the list to be compared with key K during the search is that key K_i whose index i is the closest Fibonacci sequence number to n. In the case of $n = 20$, K_{13} is the root since the closest Fibonacci number to $n = 20$ is 13.

If $(K < K_{13})$ then the next key to be compared is K_8. If again $(K < K_8)$ then it would be K_5 and so on. Now it is easy to determine the other decision nodes making use of the characteristics mentioned above. Since child nodes differ from their parent by the same amount, it is easy to see that the right child of K_{13} should be K_{18} and that of K_8 should be K_{11} and so on. Consider the grandparent-parent combination, K_8 and K_{11} respectively, since K_{11} is the right child of its parent and the difference between K_8 and K_{11} is F_4, the same between K_{11} and its two child nodes should be F_2 which is 1. Hence the two child nodes of K_{11} are K_{10} and K_{12}. Similarly, considering the grandparent and parent combination of K_{18} and K_{16} where K_{16} is the left child of its parent and their difference is given by F_3, the two child nodes of K_{16} are given by K_{15} and K_{17} (difference is F_2) respectively.

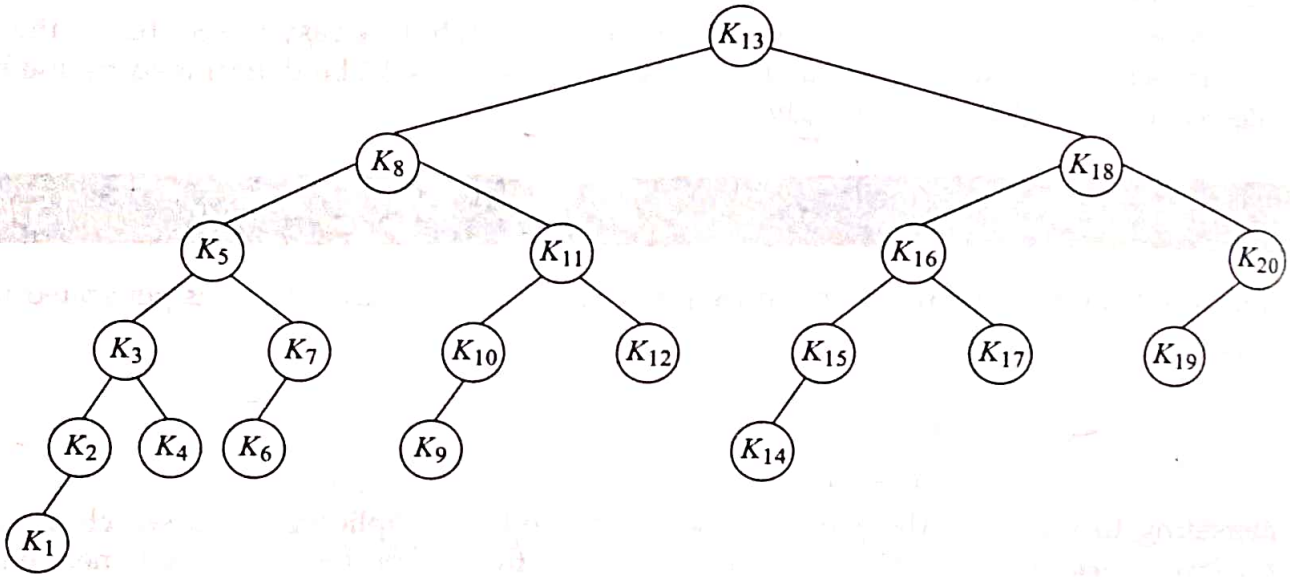


Fig. 15.3 *Decision tree for Fibonacci search*

Algorithm 15.6 illustrates the procedure for Fibonacci search. Here n, the number of data elements is such that

(i) $F_{k+1} > (n+1)$ and

(ii) $F_k + m = (n+1)$ for some $m \geq 0$, where F_{k+1} and F_k are two consecutive Fibonacci numbers.

Algorithm 15.6: Procedure for Fibonacci search

```
procedure FIBONACCI_SEARCH(L, n, K)
          /*  L[1:n] is a linear ordered (non decreasing) list of data
          elements. n is such that  F(k+1) > (n+1). Also F(k) + m = (n+1).
          K is the key to be searched in the list.  */
```

```
obtain the largest Fibonacci number F_k closest to n+1;
p = F_{k-1};
q = F_{k-2};
r = F_{k-3};
m = (n+1) - (p+q);
if ( K > L[p]) then p = p+m;
found = false;
while ( ( p ≠ 0) and ( not found) ) do
    case
    : K = L[p]:  {  print ("key found");     /* key found*/
                    found = true;
                 }
    : K < L[p]:     if (r = 0) then p = 0
                    else { p = p-r; t = q; q = r; r = t-r; }
    : K > L[p]:     if (q = 1) then p = 0
                    else { p = p+r; q = q-r; r = r-q }
    endcase
endwhile
if (found = false) then print ("key not found");
end FIBONACCI_SEARCH.
```

Example 15.6 Let us search for the key $K = 434$ in the ordered list $L = \{2, 4, 8, 9, 17, 36, 44, 55, 81, 84, 94, 116, 221, 256, 302, 356, 396, 401, 434, 536\}$. Here n ($n = 20$) the number of elements is such that (i) $F_9 > (n+1)$ and (ii) $F_8 + m = (n+1)$ where $m=0$ and $n=20$.

The algorithm for Fibonacci search first obtains the largest Fibonacci number closest to $n+1$, (i.e.), F_8 in this case. It compares $K = 434$ with the data element with index F_7 (i.e.) $L[13] = 221$. Since $K > L[13]$, the search list is reduced to $L[14: 20]= \{256, 302, 356, 396, 401, 434, 536\}$. Now K compares itself with $L[18] = 401$. Since $K > L[18]$ the search list is further reduced to $L[19:20] = \{434, 536\}$. Now K is compared with $L[20]=536$. Since $K < L[20]$ is true it results in the search list $\{434\}$ which when searched yields the search key. The key is successfully found.

Following a similar procedure, searching for 66 in the list yields an unsuccessful search.

The detailed trace of Algorithm 15.6 for the search keys 434 and 66 is shown in Table 15.3

Table 15.3 *Trace of Algorithm 15.6 for the search keys 434 and 66*

Search key K	$<$ $K = L[p]$ $>$	t	p	q	r	Remarks
434			13	8	5	$n = 20$ $m = 0$ since $F_8 + 0 = n + 1$
	$K > L[13] = 221$		13	8	5	Since $K > L[p]$, $p = p+m$
	$K > L[13] = 221$		18	3	2	

(Conta

(*Contd.*)

	$K > L[18] = 401$ $K < L[20] = 536$ $K = L[19] = 434$	1	20 19	1 1	1 0	**Key is found**
66			13	8	5	$n = 20$ $m = 0$
	$K < L[13] = 221$	8	8	5	3	
	$K > L[8] = 55$		11	2	1	
	$K < L[11] = 94$	2	10	1	1	
	$K < L[10]=84$	1	9	1	0	
	$K < L[9] = 81$					Since $(r = 0)$, p is set to 0. **Key is not found**

An advantage of Fibonacci search over binary search is that while binary search involves division which is computationally expensive, during the selection of the next element for key comparison, Fibonacci search involves only addition and subtraction.

In this chapter we discuss the internal sorting techniques of Bubble Sort, Insertion Sort, Selection sort, Merge Sort, Shell sort, Quick Sort, Heap Sort and Radix Sort.

16.2 Bubble Sort

Bubble sort belongs to the family of *sorting by exchange* or *transposition*, where during the sorting process pairs of elements that are out of order are interchanged until the whole list is ordered. Given an unordered list $L = \{K_1, K_2, K_3, \ldots K_n\}$ bubble sort orders the elements in their ascending order (i.e.), $L = \{K_1, K_2, K_3, \ldots K_n\}, K_1 \leq K_2 \leq \ldots K_n$

Given the unordered list $L = \{K_1, K_2, K_3, \ldots K_n\}$, of keys, bubble sort compares pairs of elements K_i and K_j swapping them if $K_i > K_j$. At the end of the first pass of comparisons, the largest element in the list L moves to the last position in the list. In the next pass, the sublist $\{K_1, K_2, K_3, \ldots K_{n-1}\}$ is considered for sorting. Once again the pair wise comparison of elements in the sub list results in the next largest element floating to the last position of the sublist. Thus in $(n-1)$ passes where n is the number of elements in the list, the list L is sorted. The sorting is called bubble sorting for the reason that, with each pass the next largest element of the list floats or "bubbles" to its appropriate position in the sorted list.

Algorithm 16.1 illustrates the working of bubble sort.

Algorithm 16.1: Procedure for Bubble sort

```
procedure BUBBLE_SORT(L, n)
                /* L[1:n] is an unordered list of data elements to be
                sorted in the ascending order */

for i = 1 to n-1 do   /* n-1 passes*/
  for j = 1 to n-i do
    if ( L[j] > L[j+1]) swap(L[j],L[j+1]);
                                   /*  swap pair wise elements*/
  end      /* the next largest element "bubbles" to the last position*/
end
end BUBBLE_SORT.
```

Example 16.1 Let L = {92, 78, 34, 23, 56, 90, 17, 52, 67, 81, 18} be an unordered list. As the first step in the first pass of bubble sort, 92 is compared with 78. Since 92 > 78, the elements are swapped yielding the list { **78, 92**, 34, 23, 56, 90, 17, 52, 67, 81, 18}. The swapped elements are shown in bold. Now the pair 92 and 34 are compared resulting in a swap which yields the list {78, **34, 92**, 23, 56, 90, 17, 52, 67, 81, 18}. It is easy to see that at the end of pass one, the largest element of the list viz., 92 would have moved to the last position in the list. At the end of pass one, the list would be { 78, 34, 23, 56, 90, 17, 52, 67, 81, **18, 92**}.

In the second pass the list considered for sorting discounts the last element viz., 92 since 92 has found its appropriate position in the sorted list. At the end of the second pass, the next largest element viz., 90 would have moved to the end of the list. The partially sorted list at this point would be { 34, 23, 56, 78, 17, 52, 67, 81, **18, 90**, 92}. The elements shown in grey indicate elements discounted from the sorting process. In pass 10 the whole list would be completely sorted.

The trace of algorithm BUBBLE_SORT (Algorithm 16.1) over L is shown in Table 16.1. Here i keeps count of the passes and j keeps track of the pair wise element comparisons within a pass. The lower (l) and upper (u) bounds of the loop controlled by j in each pass is shown as $l..u$. Elements shown in grey and underlined in the list L at the end of pass i, indicate those discounted from the sorting process.

Table 16.1 *Trace of Algorithm 16.1 over the list L= {92, 78, 34, 23, 56, 90, 17, 52, 67, 81, 18}*

(Pass) i	j	List L at the end of Pass i
1	1..10	{ 78, 34, 23, 56, 90, 17, 52, 67, 81, 18, 92}
2	1..9	{ 34, 23, 56, 78, 17, 52, 67, 81, 18, 90, 92}.
3	1..8	{23, 34, 56, 17, 52, 67, 78, 18, 81, 90, 92}
4	1..7	{23, 34, 17, 52, 56, 67, 18, 78, 81, 90, 92}
5	1..6	{23, 17, 34, 52, 56, 18, 67, 78, 81, 90, 92}
6	1..5	{17, 23, 34, 52, 18, 56, 67, 78, 81, 90, 92}
7	1..4	{17, 23, 34, 18, 52, 56, 67, 78, 81, 90, 92}
8	1..3	{17, 23, 18, 34, 52, 56, 67, 78, 81, 90, 92}
9	1..2	{17, 18, 23, 34, 52, 56, 67, 78, 81, 90, 92}
10	1..1	{17, 18, 23, 34, 52, 56, 67, 78, 81, 90, 92}

Stability and performance analysis

Bubble sort is a stable sort since equal keys do not undergo swapping, as can be observed in Algorithm 16.1, and this contributes to the keys maintaining their relative orders of occurrence in the sorted list.

Example 16.2 Consider the unordered list $L = \{ 7^1, 7^2, 7^3, 6\}$. The repeating keys have been distinguished using their orders of occurrence as superscripts. The partially sorted lists at the end of each pass of the bubble sort algorithm are shown below:

Pass 1: $\{ 7^1, 7^2, 6, 7^3\}$
Pass 2: $\{ 7^1, 6, 7^2, 7^3\}$
Pass 3: $\{ 6, 7^1, 7^2, 7^3\}$

Observe how the equal keys 7^1, 7^2, 7^3 maintain their relative orders of occurrence in the sorted list as well, verifying the stability of bubble sort.

The time complexity of bubble sort in terms of key comparisons is given by $O(n^2)$. It is easy to see this since the procedure involves two loops with their total frequency count given by $O(n^2)$.

Insertion Sort 16.3

Insertion sort as the name indicates belongs to the family of *sorting by insertion* which is based on the principle that a new key K is *inserted* at its appropriate position in an already sorted sub list.

Given an unordered list $L = \{K_1, K_2, K_3, \ldots K_n\}$, insertion sort employs the principle of constructing the list $L = \{K_1, K_2, K_3, \ldots K_i, K, K_j, K_{j+1}, \ldots K_n\}$, $K_1 \le K_2 \le \ldots K_i$ and inserting a key K at its appropriate position by comparing it with its sorted sublist of predecessors $\{K_1, K_2, K_3, \ldots K_i\}$, $K_1 \le K_2 \le \ldots K_i$ for every key K ($K = K_i = 2, 3, \ldots, n$) belonging to the unordered list L.

In the first pass of insertion sort, K_2 is compared with its sorted sublist of predecessors viz., K_1. K_2 inserts itself at the appropriate position to obtain the sorted sublist $\{K_1, K_2\}$. In the second pass, K_3 compares itself with its sorted sublist of predecessors viz., $\{K_1, K_2\}$ to insert itself at its appropriate position yielding the sorted list $\{K_1, K_2, K_3\}$ and so on. In the $(n-1)^{th}$ pass, K_n compares itself with its sorted sublist of predecessors $\{K_1, K_2, \ldots K_{n-1}\}$ and having inserted itself at the appropriate position yields the final sorted list $L = \{K_1, K_2, K_3, \ldots K_i, \ldots K_j, \ldots K_n\}$, $K_1 \le K_2 \le \ldots K_i \le \ldots \le K_j \le \ldots K_n$. Since each key K finds its appropriate position in the sorted list, such a technique is referred to as *sinking* or *sifting* technique.

Algorithm 16.2 illustrates the working of Insertion sort. The **for** loop in the algorithm keeps count of the passes and the **while** loop implements the comparison of the key `key` with its sorted sublist of predecessors. So long as the preceding element in the sorted sublist is greater than `key` the swapping of the element pair is done. If the preceding element in the sorted sublist is less than or equal to `key`, then `key` is left at its current position and the current pass terminates.

Example 16.3 Let $L = \{16, 36, 4, 22, 100, 1, 54\}$ be an unordered list of elements. The various passes of the insertion sort procedure are shown below. The snapshots of the list before and after each pass is shown. The key chosen for insertion in each pass is shown in bold and the sorted sublist of predecessors against which the key is compared are shown in brackets.

Pass 1 (**Insert 36**)	{ [16] **36**, 4, 22, 100, 1, 54}
After Pass 1	{ [16 36] 4, 22, 100, 1, 54}
Pass 2 (**Insert 4**)	{ [16 36] **4**, 22, 100, 1, 54}
After Pass 2	{ [4 16 36] 22, 100, 1, 54}
Pass 3 (**Insert 22**)	{ [4 16 36] **22**, 100, 1, 54}
After Pass 3	{ [4 16 22 36] 100, 1, 54}
Pass 4 (**Insert 100**)	{ [4 16 22 36] **100**, 1, 54}
After Pass 4	{ [4 16 22 36 100] 1, 54}
Pass 5 (**Insert 1**)	{ [4 16 22 36 100] **1**, 54}
After Pass 5	{ [1 4 16 22 36 100] 54}
Pass 6 (**Insert 54**)	{ [1 4 16 22 36 100] **54**}
After Pass 6	{ [1 4 16 22 36 54 100]}

Algorithm 16.2: Procedure for Insertion sort

```
procedure INSERTION_SORT(L, n)
          /* L[1:n] is an unordered list of data elements to be sorted
          in the ascending order */
for i = 2 to n do                    /* n-1 passes*/
    key = L[i];                      /* key is the key to be inserted
                                     and position its location in the
                                     unordered list*/
```

```
        position = i;
                                /* compare key with its sorted
                                sublist of predecessors for insertion
                                at the appropriate position*/
        while (position > 1) and (L[position-1]> key) do
                L[position] = L[position-1];
                position = position -1;
                L[position] = key;
        end
end
end INSERTION_SORT.
```

Stability and performance analysis

Insertion sort is a stable sort. It is evident from the algorithm that the insertion of key K at its appropriate position in the sorted sublist affects the position index of the elements in the sublist so long as the elements in the sorted sublist are greater than K. When the elements are less than are equal to the key K, there is no displacement of elements and this contributes to retaining the original order of keys which are equal, in the sorted sublists.

Example 16.4 Consider the list $L = \{ 3^1, 1, 2^1, 3^2, 3^3, 2^2\}$ where the repeated keys have been superscripted with numbers indicative of their relative orders of occurrence. The keys for insertion are shown in bold and the sorted sublists are bracketed.

The passes of the insertion sort are shown below:

Pass 1 **(Insert 1)**	$\{ [3^1]$ **1**, $2^1, 3^2, 3^3, 2^2\}$
After Pass 1	$\{ [1\ \ 3^1] 2^1, 3^2, 3^3, 2^2\}$
Pass 2 **(Insert 2)**	$\{ [1\ \ 3^1\ ]$ **2^1**, $3^2, 3^3, 2^2\}$
After Pass 2	$\{ [1\ \ 2^1\ \ 3^1] 3^2, 3^3, 2^2\ \}$
Pass 3 **(Insert 3)**	$\{ [1\ \ 2^1\ \ 3^1]$ **3^2**, $3^3, 2^2\ \}$
After Pass 3	$\{[1\ \ 2^1\ \ 3^1\ \ 3^2] 3^3, 2^2\ \}$
Pass 4 **(Insert 3)**	$\{[1\ \ 2^1\ \ 3^1\ \ 3^2]$ **3^3**, $2^2\ \}$
After Pass 4	$\{[1\ \ 2^1\ \ 3^1\ \ 3^2\ \ 3^3] 2^2\ \}$
Pass 5 **(Insert 2)**	$\{[1\ \ 2^1\ \ 3^1\ \ 3^2\ \ 3^3]$ **2^2**$\ \}$
After Pass 5	$\{[1\ \ 2^1\ \ 2^2\ \ 3^1\ \ 3^2\ \ 3^3\ ]\}$

The stability of insertion sort can be easily verified on this example. Observe how keys which are equal maintain their original relative orders of occurrence in the sorted list.

The worst case performance of insertion sort occurs when the elements in the list are already sorted in their descending order. It is easy to see that in such a case every key that is to be inserted has to move to the front of the list and therefore undertakes the maximum number of comparisons. Thus if the list $L = \{K_1, K_2, K_3, \ldots K_n\}$, $K_1 \geq K_2 \geq \ldots K_n$ is to be insertion sorted then the number of comparisons for the insertion of key K_i would be $(i-1)$ since K_i would swap positions with each of the $(i-1)$ keys occurring before it until it moves to position 1. Therefore the total number of comparisons for inserting each of the keys is given by

$$1+2+3+\ldots(n-1)=\frac{(n-1)(n)}{2}\approx O(n^2)$$

The best case complexity of insertion sort arises when the list is already sorted in the ascending order. In such a case the complexity in terms of comparisons is given by $O(n)$. The average case performance of insertion sort reports $O(n^2)$ complexity.

16.4 Selection Sort

Selection sort is built on the principle of repeated *selection* of elements satisfying a specific criterion to aid the sorting process.

The steps involved in the sorting process are listed below:

(i) Given an unordered list $L=\{K_1, K_2, K_3, \ldots K_j \ldots K_n\}$, select the minimum key K

(ii) Swap K with the element in the first position of the list L, viz., K_1. By doing so the minimum element of the list has secured its rightful position of number one in the sorted list. This step is termed pass 1.

(iii) Exclude the first element and select the minimum element K, from amongst the remaining elements of the list L. Swap K with the element in the second position of the list viz., K_2. This is termed pass 2.

(iv) Exclude the first two elements which have occupied their rightful positions in the sorted list L. Repeat the process of selecting the next minimum element and swapping it with the appropriate element, until the entire list L gets sorted in the ascending order. The entire sorting gets done in $(n-1)$ passes.

Selection sort can also undertake sorting in the descending order by selecting the *maximum element* instead of the minimum element and swapping it with the element in the *last position* of the list L.

Algorithm 16.3 illustrates the working of selection sort. The procedure `FIND_MINIMUM(L, i, n)` selects the minimum element from the array $L[i{:}n]$ and returns the position index of the minimum element to procedure `SELECTION_SORT`. The **`for`** `loop` in the `SELECTION_SORT` procedure represents the $(n-1)$ passes needed to sort the array $L[1{:}n]$ in the ascending order. Function **`swap`** swaps the elements input to it.

Algorithm 16.3: Procedure for Selection sort

```
procedure SELECTION_SORT(L, n)
                /* L[1:n] is an unordered list of data elements to be
                sorted in the ascending order */
for i = 1 to n-1 do                                  /* n-1 passes*/
 minimum_index = FIND_MINIMUM(L, i, n);              /* find minimum element
                   of the list L[i:n] and store the position index of
                   the element in minimum_index*/
 swap(L[i], L[minimum_index]);
end

end SELECTION_SORT
```

```
procedure FIND_MINIMUM(L, i, n)
          /* the position index of the minimum element in the array
             L[i : n] is returned*/
min_indx = i;
for j = i + 1 to n do
          if (L[j] < L[min_indx]) min_indx = j;
end
return (min_indx)
end FIND_MINIMUM
```

Example 16.5 Let L = {71, 17, 86, 100, 54, 27} be an unordered list of elements. Each pass of selection sort is traced below. The minimum element is shown in bold and the arrows indicate the swap of the elements concerned. The elements in gray indicate their exclusion in the passes concerned.

Pass	List L (During Pass)	List L (After Pass)
1	{71, **17**, 86, 100, 54, 27}	{17, 71, 86, 100, 54, 27}
2	{**17**, 71, 86, 100, 54, **27**}	{17, 27, 86, 100, 54, 71}
3	{**17**, **27**, 86, 100, **54**, 71}	{17, 27, 54, 100, 86, 71}
4	{**17**, **27**, **54**, 100, 86, **71**}	{17, 27, 54, 71, 86, 100}
5	{**17**, **27**, **54**, **71**, **86**, 100}	{17, 27, 54, 71, 86, 100} **(Sorted list)**

Stability and performance analysis

Selection sort is not stable. Example 16.6 illustrates a case. The computationally expensive portion of selection sort occurs when the minimum element has to be selected in each pass. The time complexity of FIND_MINIMUM procedure is $O(n)$. The time complexity of SELECTION_SORT procedure is therefore $O(n^2)$.

Example 16.6 Consider the list L = { 6^1, 6^2, 2}. The repeating keys have been superscripted with numbers indicative of their relative orders of occurrence. A trace of the selection sort procedure is shown below. The minimum element is shown in bold and the swapping is indicated by the curved arrow. The elements excluded from the pass are shown in gray.

Pass	List L (During Pass)	List L (After Pass)
1	{ 6^1, 6^2, **2** }	{ 2, 6^2, 6^1 }
2	{ 2, **6^2**, 6^1 }	{ 2, 6^2, 6^1 } **(Sorted list)**

The selection sort on the given list L is therefore not stable.

16.5 Merge Sort

Merging or *collating* is a process by which two ordered lists of elements are combined or merged into a single ordered list. *Merge sort* makes use of the principle of merge to sort an unordered list of elements and hence the name. In fact a variety of sorting algorithms belonging to the family of *sorting by merge* exist. Some of the well known external sorting algorithms belong to this class.

Two-way Merging

Two-way merging deals with the merging of two ordered lists.

Let $L_1 = \{a_1, a_2, \ldots a_i \ldots a_n\}\ a_1 \le a_2 \le \ldots \le a_i \le \ldots a_n$ and $L_2 = \{b_1, b_2, \ldots b_j \ldots b_m\}\ b_1 \le b_2 \le \ldots \le b_j \le \ldots b_m$ be two ordered lists. Merging combines the two lists into a single list L by making use of the following cases of comparison between the keys a_i and b_j belonging to L_1 and L_2 respectively:

A1. If ($a_i < b_j$) then drop a_i into the list L

A2. If ($a_i > b_j$) then drop b_j into the list L

A3. If ($a_i = b_j$) then drop both a_i and b_j into the list L

In the case of **A1**, once a_i is dropped into the list L the next comparison of b_j proceeds with a_{i+1}. In the case of **A2**, once b_j is dropped into the list L the next comparison of a_i proceeds with b_{j+1}. In the case of **A3** the next comparison proceeds with a_{i+1} and b_{j+1}. At the end of merge, list L contains $(n+m)$ ordered elements.

The series of comparisons between pairs of elements from the lists L_1 and L_2 and the dropping of the relatively smaller elements into the list L proceeds until one of the following cases happens:

B1. L_1 gets exhausted earlier to that of L_2. In such a case, the remaining elements in list L_2 are dropped into the list L in the order of their occurrence in L_2 and the merge is done.

B2. L_2 gets exhausted earlier to that of L_1. In such a case the remaining elements in list L_1 are dropped into the list L in the order of their occurrence in L_1 and the merge is done.

B3. Both L_1 and L_2 are exhausted, in which case merge is done.

Example 16.7 Consider the two ordered lists $L_1 = \{4, 6, 7, 8\}$ and $L_2 = \{3, 5, 6\}$. Let us merge the two lists to get the ordered list L. L contains 7 elements in all. Figure 16.1 illustrates the snapshots of the merge process. Observe how when the elements 6, 6 are compared both the elements drop into the list L. Also note how list L_2 gets exhausted earlier to L_1 resulting in all the remaining elements of list L_1 getting flushed into list L.

Algorithm 16.4 illustrates the procedure for merge. Here the two ordered lists to be merged are given as $(x_1, x_2, \ldots x_t)$ and $(x_{t+1}, x_{t+2}, \ldots x_n)$ to enable reuse of the algorithm for merge sort to be discussed subsequently. The input parameters to procedure `MERGE` is given as `(x, first, mid, last)` where `first` is the starting index of the first list, `mid` the index related to the end/beginning of the first and second list respectively and `last` the ending index of the second list. The call to merge the two lists, $(x_1, x_2, \ldots x_t)$ and $(x_{t+1}, x_{t+2}, \ldots x_n)$ would be `MERGE(x, 1, t, n)`. While the first `while` loop in the procedure performs the pair wise comparison of elements in the two lists as discussed in cases **A1-A3**, the second `while` loop takes care of the case **B1** and the third loop that of the case **B2**. Case **B3** is inherently taken care of in the first `while` loop.

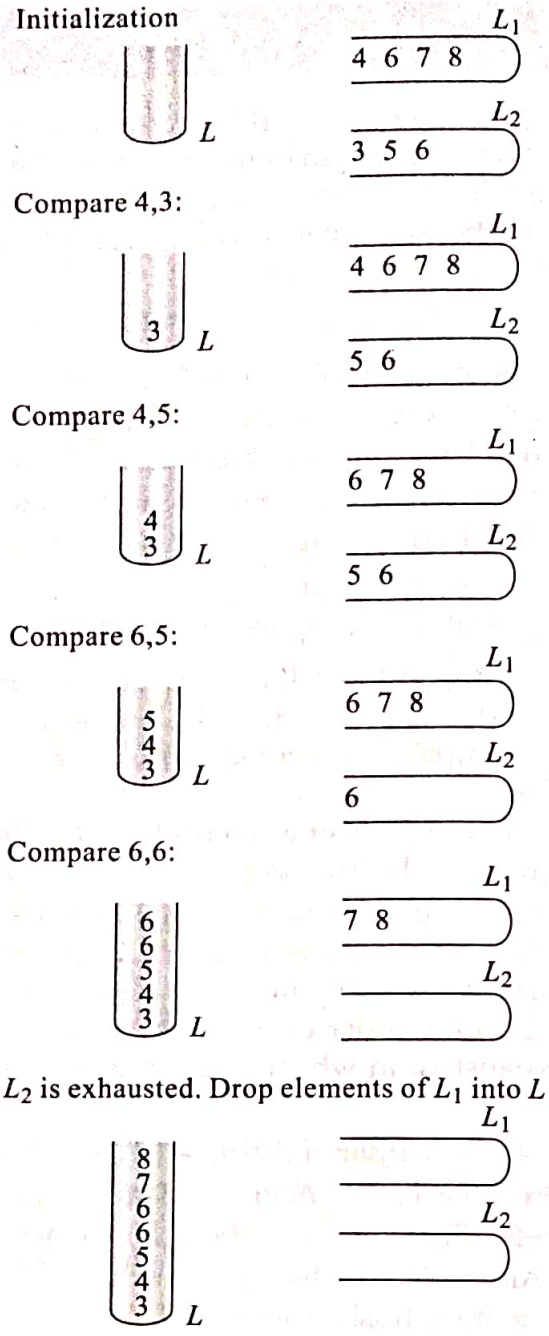


Fig. 16.1 *Two-way merge*

Performance analysis

The first `while` loop in Algorithm 16.4 executes at most `(last-first+1)` times and plays a significant role in the time complexity of the algorithm. The rest of the `while` loops only move the elements of the unexhausted lists into the list L. The complexity of the first `while` loop and hence the algorithm is given by O`(last-first+1)`. In the case of merging two lists $(x_1, x_2, \ldots x_t)$, $(x_{t+1}, x_{t+2}, \ldots x_n)$ where the number of elements in the two lists sums to n, the time complexity of `MERGE` is given by $O(n)$.

k-way merging

The two-way merge principle could be extended to k ordered lists in which case it is termed as *k-way merging*. Here k ordered lists

$$L_1 = \{a_{11}, a_{12}, \ldots a_{1i} \ldots a_{1n_1}\},\ a_{11} \le a_{12} \le \ldots \le a_{1i} \le \ldots a_{1n_1},$$

$$L_2 = \{a_{21}, a_{22}, \ldots a_{2i} \ldots a_{2n_2}\},\ a_{21} \le a_{22} \le \ldots \le a_{2i} \le \ldots a_{2n_2}$$

$$L_k = \{a_{k1}, a_{k2}, \ldots a_{ki} \ldots a_{kn_k}\},\ a_{k1} \le a_{k2} \le \ldots \le a_{ki} \le \ldots a_{kn_k}$$

each comprising $n_1, n_2, \ldots n_k$ number of elements are merged into a single ordered list L comprising $(n_1 + n_2 + \ldots n_k)$ number of elements. At every stage of comparison, k keys a_{ij}, one from each list, are compared before the smallest of the keys are dropped into the list L. Cases **A1- A3** and **B1 – B3** discussed in Sec. 16.5 with regard to two-way merge, hold good in this case as well but as extended to k lists. Illustrative Problem 16.3 discusses an example k-way merge.

Non recursive merge sort procedure

Given a list $L = \{K_1, K_2, K_3, \ldots K_n\}$ of unordered elements, merge sort sorts the list making use of procedure MERGE repeatedly over several passes.

The non recursive version of merge sort merely treats the list L of n elements as n independent ordered lists of one element each. In pass one, the n singleton lists are pair wise merged. At the end of pass 1, the merged lists would have a size of 2 elements each. In pass 2, the lists of size 2 are pair wise merged to obtain ordered lists of size 4 and so on. In the i^{th} pass the lists of size $2^{(i-1)}$ are merged to obtain ordered lists of size $2^{(i)}$.

During the passes, if any of the lists are unable to find a pair for their respective merge operation, then they are simply carried forward to the next pass.

Example 16.8 Consider the list L = {12, 56, 1, 34, 89, 78, 43, 10} to be merge sorted using its non recursive formulation. Figure 16.2 illustrates the pair wise merging undertaken in each of the passes. The sublists in each pass are shown in brackets. Observe how pass 1 treats the list L as 8 ordered sublists of one element each and at the end of merge sort, pass 3 obtains a single list of size 8 which is the final sorted list.

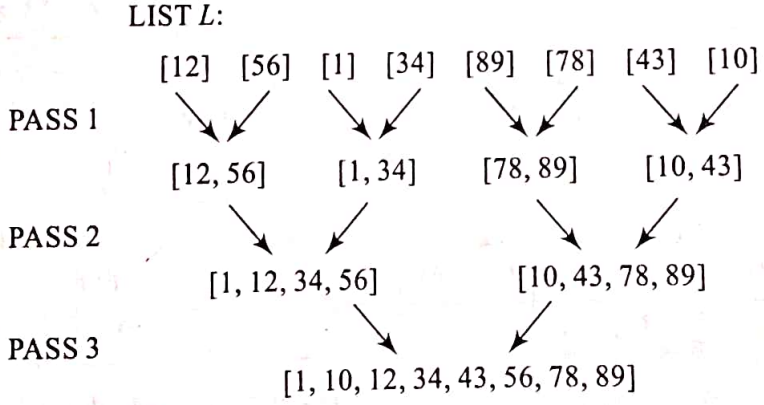


Fig. 16.2 *Non recursive merge sort of list L = { 12, 56, 1, 34, 89, 78, 43, 10} (Example 16.8)*

Performance analysis Merge sort proceeds by running several passes over the list that is to be sorted. In pass 1 sublists of size 1 are merged, in pass 2 sublists of size 2 are merged and in the i^{th} pass sublists of size $2^{(i-1)}$ are merged. Thus one could expect a total of passes over the list. With the merge operation commanding $O(n)$ time complexity, each pass of merge sort takes $O(n)$ time. The time complexity of merge sort therefore turns out to be $O(n.\log_2 n)$.

Stability Merge sort is a stable sort since the original relative orders of occurrence of repeating keys are maintained in the sorted list. Illustrative Problem 16.4 demonstrates the stability of the sort over a list.

Recursive merge sort procedure

The recursive merge sort procedure is built on the design principle of Divide and Conquer. Here, the original unordered list of elements is recursively divided roughly into two sublists until the sublists are small enough where a merge operation is done before they are combined to yield the final sorted list.

Algorithm 16.5 illustrates the recursive merge sort procedure. The procedure makes use of MERGE (Algorithm 16.4) for its merging operation.

Example 16.9 Let us merge sort the list $L = \{ 12, 56, 1, 34, 89, 78, 43, 10\}$ using Algorithm 16.5. The tree of recursive calls demonstrating the working of the procedure on the list L is shown in Fig. 16.3. The list is recursively divided into two sublists to be merge sorted before they are merged to obtain the final sorted list. Each rectangular node of the tree indicates a procedure call to `MERGE_SORT` with the parameters to the call inscribed inside the box. Beneath the parameter list is shown the output sublist obtained at the end of the execution of the procedure call.

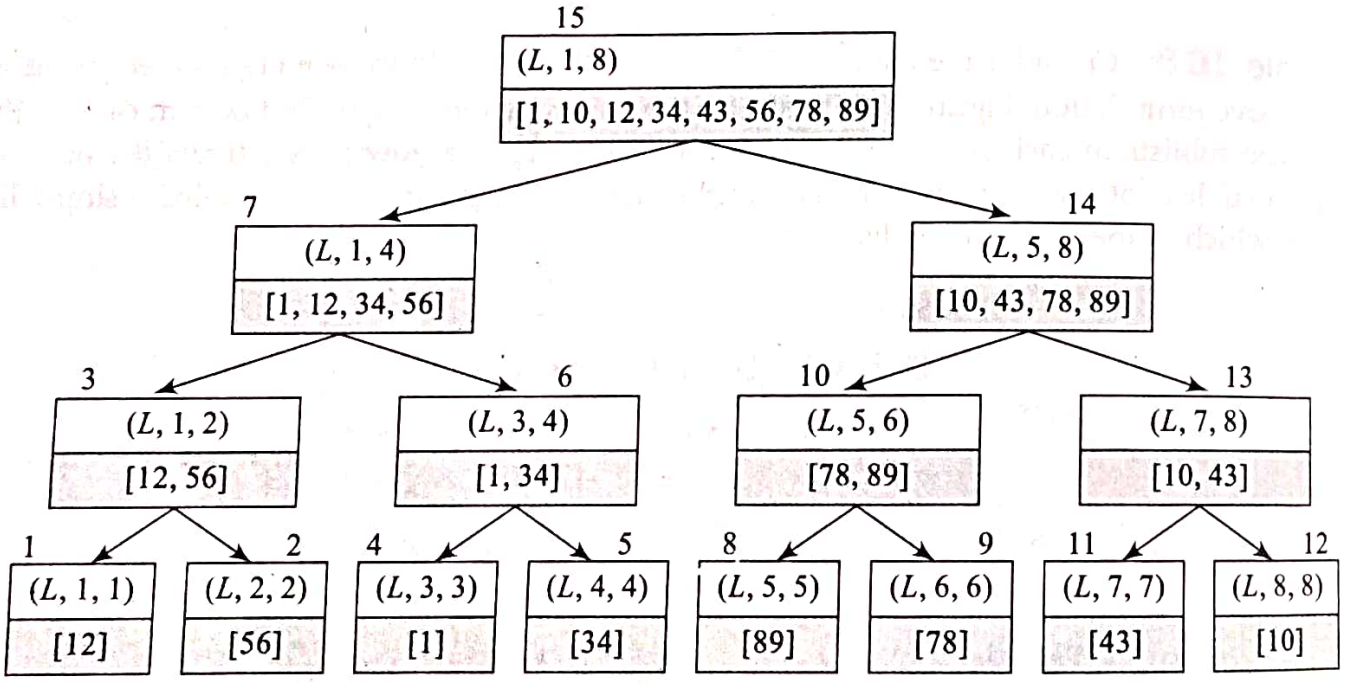


Fig. 16.3 *Tree of recursive calls illustrating recursive merge sort of list L = { 12, 56, 1, 34, 89, 78, 43, 10} (Example 16.9)*

The invocation of MERGE_SORT (L, 1, 8) generates two other calls viz., MERGE_SORT (L, 1, 4) and MERGE_SORT (L, 5, 8) and so on leading to the construction of the tree. Down the tree, the procedure calls MERGE_SORT (L, 1, 1) and MERGE_SORT (L, 2, 2) in that order, are the first to terminate releasing the lists [12] and [56] respectively. This triggers the MERGE (L, 1, 2) procedure yielding the sublist [12, 56] as the output of the procedure call MERGE_SORT (L, 1, 2). Observe [12, 56] inscribed in the rectangular box 3 which corresponds to the procedure call MERGE_SORT (L, 1, 2). Proceeding in a similar fashion, it is easy to build the tree and obtain the sorted sublists resulting out of each of the calls. The number marked over each rectangular node indicates the order of execution of the recursive procedure calls to MERGE_SORT. With MERGE_SORT (L, 1, 4) yielding the sorted sublist [1, 12, 34, 56] and MERGE_SORT (L, 5, 8) yielding [10, 43, 78, 89], the execution of the call MERGE (L, 1, 4, 8) terminates the call to MERGE_SORT (L, 1, 8) resulting in the sorted list [1, 10, 12, 34, 43, 56, 78, 89].

Performance analysis Recursive merge sort follows a Divide and Conquer principle of algorithm design. Let $T(n)$ be the time complexity of MERGE_SORT where n is the size of the list. The recurrence relation for the time complexity of the algorithm is given by

$$T(n) = 2.T\left(\frac{n}{2}\right) + O(n), \quad n \geq 2$$
$$= d$$

Here $T\left(\frac{n}{2}\right)$ is the time complexity for each of the two recursive calls to MERGE_SORT over a list of size $n/2$ and d is a constant. $O(n)$ is the time complexity of merge. Framing the recurrence relation as

$$T(n) = 2.T\left(\frac{n}{2}\right) + c.n, \quad n \geq 2$$
$$= d$$

where c is a constant and solving the relation yields the time complexity $T(n) = O(n.\log_2 n)$ (see Illustrative Problem 16.5).

Quick Sort 16.7

Quick sort procedure formulated by C.A.R. Hoare belongs to the family of *sorting by exchange or transposition* where elements that are out of order are exchanged amongst themselves to obtain the sorted list.

The procedure works on the principle of *partitioning* the unordered list into two sublists at every stage of the sorting process based on what is called a *pivot element*. The two sublists occur to the left and right of the pivot element. The pivot element determines its appropriate position in the sorted list and is therefore freed of its participation in the subsequent stages of the sorting process. Again each of the sublists are partitioned against their respective pivot elements until no more partitioning can be called for. At this stage all the elements would have determined their appropriate positions in the sorted list and quick sort is done.

Partitioning

Consider an unordered list $L = \{K_1, K_2, K_3, \ldots K_n\}$. How does partitioning occur? Let us choose K_1 to be the pivot element. Now K_1 compares itself with each of the keys on a left to right encounter looking for the first key K_i, $K_i \geq K$. Again K compares itself with each of the keys on a right to left encounter looking for the first key K_j, $K_j \leq K$. If K_i and K_j are such that $i < j$, then K_i and K_j are exchanged. Figure 16.6(a) illustrates the process of exchange.

Now K moves ahead from position index i on a left to right encounter looking for a key K_s, $K_s \geq K$. Again as before, K moves on a right to left encounter beginning from position index j looking for a key K_t, $K_t \leq K$. As before if $s < t$, then K_s and K_t are exchanged and the process repeats

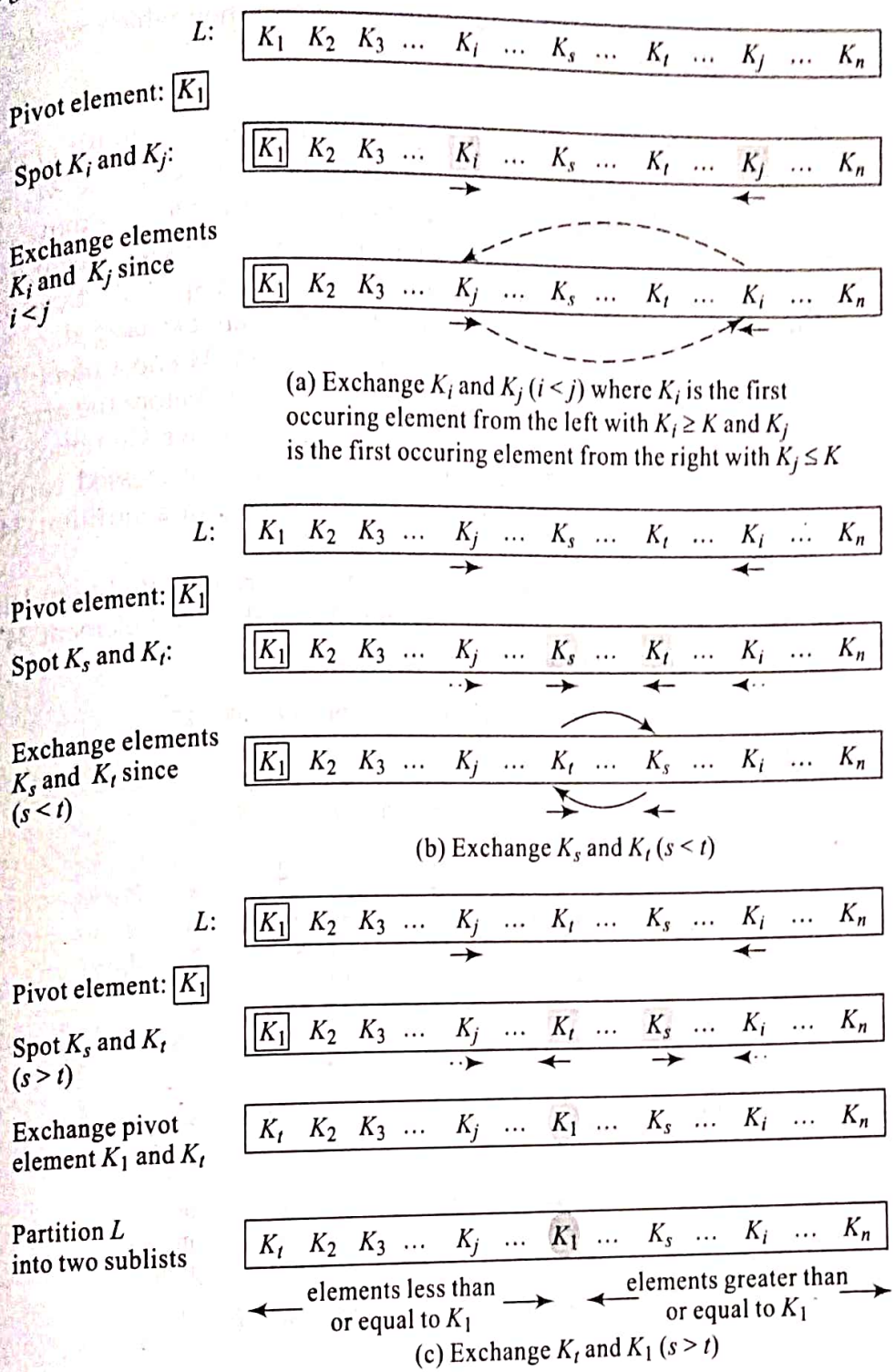


Fig. 16.6 *Partitioning in Quick Sort*

(Fig. 16.6(b)). If $s > t$, then K exchanges itself with K_t -the key which is smaller of K_s and K_t. At this stage a *partition* is said to occur. The pivot element K which has now exchanged position with K_t is the median around which the list partitions itself or splits itself into two. Figure 16.6(c) illustrates partition. Now what do we observe about the partitioned sublists and the pivot element?

(i) The sublist occurring to the left of the pivot element K (now at position t) has all its elements less than or equal to K and the sublist occurring to the right of the pivot element K has all its elements greater than or equal to K.

(ii) The pivot element has settled down to its appropriate position which would turn out to be its rank in the sorted list.

Example 16.12 Let $L = \{34, 26, 1, 45, 18, 78, 12, 89, 27\}$ be an unordered list of elements. We now demonstrate the process of partitioning on the above list. Let us choose 34 as the pivot element. Figure 16.7 illustrates the snap shots of partitioning the list. Here 34 moves left to right looking for the first element that is greater than or equal to it and spots 45. Again moving from right to left looking for the first element less than or equal to 34, it spots 27. Since the position index of 45 is less than that of 27 (arrows face each other), they are exchanged.

Proceeding from the points where the moves were last stopped, 34 encounters 78 during its left to right move and encounters 12 during its right to left move. As before the arrows face each other resulting in an exchange of 78 and 12. In the next lap of the move we notice the elements 78 and 12 are spotted again but this time note that the arrows have crossed each other. This implies that the position index of 78 is greater than that of 12 calling for a partition. 34 exchanges position with 12 and the list is partitioned into two as shown.

It may be seen that all elements less than or equal to 34 have accumulated to its left and those greater than or equal to 34 have accumulated to its right. Again the pivot element 34 has settled down at position index 6 which is its rank in the sorted list.

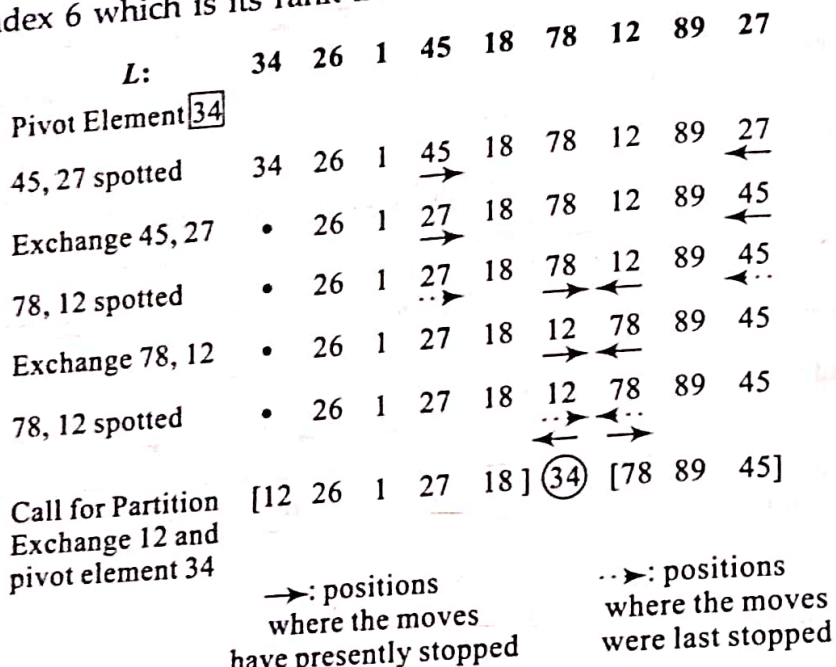


Fig. 16.7 *Partitioning a list (Example 16.12)*

Quick sort procedure

Once the method behind partitioning is known, quick sort is nothing but repeated partitioning until every pivot element settles down to its appropriate position thereby sorting the list.

Algorithm 16.8 illustrates the quick sort procedure. The algorithm employs the Divide and Conquer principle by exploiting procedure `PARTITION` (Algorithm 16.7) to partition the list into two sublists and recursively calling procedure `QUICK_SORT` to sort the two sublists.

Procedure `PARTITION` partitions the list `L[first:last]` at the position `loc` where the pivot element settles down.

Algorithm 16.7: Procedure for Partition

```
procedure PARTITION(L, first, last, loc )
          /* L[first:last] is the list to be partitioned. loc is the
             position where the pivot element finally settles down*/

left = first;
right = last+1;
pivot_elt = L[first];  /* set the pivot element to the first
                          element in list L*/
while (left < right) do
    repeat
        left = left+1;     /* pivot element moves left to right*/
    until L[left] ≥ pivot_elt;
    repeat
        right = right -1; /* pivot element moves right to left*/
    until L[right] ≤ pivot_elt;
    if (left < right) then swap(L[left], L[right]); /*arrows face each
        other*/

end
loc = right
swap(L[first], L[right]);      /* arrows have crossed each other - exchange
                                  pivot element L[first]  with L[right]*/

end PARTITION.
```

Example 16.13 Let us quick sort the list $L = \{5, 1, 26, 15, 76, 34, 15\}$. The various phases of the sorting process are shown in Fig. 16.8. When the partitioned sublists contain only one element then no sorting is done. Also in phase 4 of Fig. 16.8 observe how the pivot element 34 exchanges with itself. The final sorted list is {1, 5, 15, 15, 26, 34, 76}.

Algorithm 16.8: Procedure for Quick Sort

```
procedure QUICK_SORT(L, first, last )
          /* L[first:last] is the unordered list of elements to be
             quick sorted. The call to the procedure to sort the
             list L[1:n] would be QUICK_SORT(L, 1, n)*/

if (first < last) then
 { PARTITION(L, first, last, loc) ;   /* partition the list into two
                                         sublists at loc*/
   QUICK_SORT(L, first, loc-1 );      /* quick sort the sublist
                                         L[first,loc-1]*/

   QUICK_SORT(L, loc+1, last );       /* quick sort the sublist
                                         L[loc+1, last]*/

 }
end QUICK_SORT.
```

Stability and performance analysis

Quick sort is not a stable sort. During the partitioning process keys which are equal are subject to exchange and hence undergo changes in their relative orders of occurrence in the sorted list.

L: {5, 1, 26, 15, 76, 34, 15}

Phase 1: Pivot element [5]

• 1 26 15 76 34 15

List L after partition [1] (5) [26 15 76 34 15]

Phase 2: List [1] needs no quick sort.

Quick sort list [26 15 76 34 15]

Pivot element [26]

• 15 76 34 15

• 15 15 34 76

List L after partition (1) (5) [15 15] (26) [34 76]

Phase 3: Quick sort list [15, 15]

Pivot element: [15]

• 15

List L after partition (1) (5) [15] (15) (26) [34 76]

Phase 4: List [15] needs no quick sort

Quick Sort [34, 76]

Pivot element [34]

• 76

List L after partition (1) (5) (15) (15) (26) (34) [76]

The final sorted list: {1, 5, 15, 15, 26, 34, 76}

Fig. 16.8 *Snapshots of the quick sort process (Example 16.13)*

Example 16.14 Let us quick sort the list $L = \{5^1, 5^2, 5^3\}$ where the superscripts indicate the relative orders of their occurrence in the list. Figure 16.9 illustrates the sorting process. It can be easily seen that quick sort is not stable.

Quick sort reports a worst case performance when the list is already sorted in its ascending order (see Illustrative Problem 16.6). The worst case time complexity of the algorithm is given by $O(n^2)$. However, quick sort reports a good average case complexity of $O(n \log n)$.

L: $\{5^1\ 5^2\ 5^3\}$

Phase1: Pivot element [5^1]

• 5^2 5^3

• 5^3 5^2

List L after portion: [5^3] (5^1) [5^2]

The final sorted list $L = \{5^3\ 5^1\ 5^2\}$

Quick sort is unstable

Fig. 16.9 *Stability of Quick Sort*