Kammavari Sangham (R) 1952
K. S. GROUP OF INSTITUTIONS
K. S. SCHOOL OF ENGINEERING AND MANAGEMENT
(Approved by AICTE, New Delhi; Affiliated to VTU, Belagavi, Karnataka; Accredited by NAAC)
www.kssem.edu.in
DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

Academic Year 2023-24

Study Material

Course Name : Analysis and Design of Algorithms

Course Code : BCS401

Course Coordinator : Mr. Ramesh Babu N
Assoc. Prof.,
Dept. of CS & BS,
KSSEM

Contents

Module 1

Syllabus

INTRODUCTION: What is an Algorithm?, Fundamentals of Algorithmic Problem Solving.
FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY: Analysis Framework, Asymptotic Notations and Basic Efficiency Classes, Mathematical Analysis of Non recursive Algorithms, Mathematical Analysis of Recursive Algorithms.
BRUTE FORCE APPROACHES: Selection Sort and Bubble Sort, Sequential Search and Brute Force String Matching.

Chapter 1 (Sections 1.1,1.2), Chapter 2(Sections 2.1,2.2,2.3,2.4), Chapter 3 (Section 3.1, 3.2)

INTRODUCTION

Why do you need to study algorithms?

a. Practical reasons
   - to know a standard set of important algorithms from different areas of computing.
   - to design new algorithms and analyze their efficiency.

b. Theoretical reasons
   - It is the core of computer science, and is relevant to most of science, business, and technology.
   - Study of algorithms enables person to develop analytical skills. Algorithms are precisely defined procedures for getting answers.

What is an Algorithm?

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

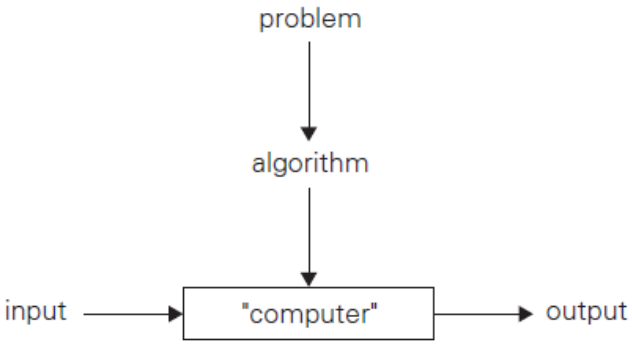


FIGURE 1.1 The notion of the algorithm.

Algorithms should have

- **Input**. Zero or more quantities are externally supplied.
- **Output**. At least one quantity is produced.
- **Definiteness**. Each instruction is clear and unambiguous. It must be perfectly clear what should be done.
- **Finiteness**. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **Effectiveness**. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper.

Important points:

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Example: Computing GCD of two numbers

Method 1: Euclid's algorithm

Euclid's algorithm is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n),$$

where $m \bmod n$ is the remainder of the division of m by n, until $m \bmod n$ is equal to 0. Since $\gcd(m, 0) = m$. the last value of m is also the greatest common divisor of the initial m and n.

For example, $\gcd(60, 24)$ can be computed as follows:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

Structured description of this algorithm in English.

Euclid's algorithm for computing $\gcd(m, n)$

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM *Euclid(m, n)*
//Computes $\gcd(m, n)$ by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return m

How do we know that Euclid's algorithm eventually comes to a stop?
The second integer of the pair gets smaller with each iteration and it cannot become negative.

The new value of *n* on the next iteration is *m* mod *n,* which is always smaller than *n.* Therefore, the value of the second integer eventually becomes 0, and the algorithm stops.

Method 2: Consecutive integer checking algorithm for computing gcd*(m, n)*
Step 1 Assign the value of min{*m, n*} to *t.*
Step 2 Divide *m* by *t.* If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
Step 3 Divide *n* by *t.* If the remainder of this division is 0, return the value of *t* as the answer and stop; otherwise, proceed to Step 4.
Step 4 Decrease the value of *t* by 1. Go to Step 2.

Example: for numbers 60 and 24, the algorithm will try first 24, then 23, and so on, until it reaches 12, where it stops.

This algorithm, does not work correctly when one of its input numbers is zero.

Method 3: Middle-school procedure
Step 1 Find the prime factors of *m*.
Step 2 Find the prime factors of *n*.
Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If *p* is a common factor occurring *pm* and p_n times in *m* and *n,* respectively, it should be repeated min{ p_m, p_n } times.)
Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Example: For the numbers 60 and 24, we get
60 = 2 . 2 . 3 . 5
24 = 2 . 2 . 2 . 3
gcd*(*60, 24*)* = 2 . 2 . 3 = 12.

How to find prime numbers? Using ***sieve of Eratosthenes***
How to find longest common sequences? Difficult.

Sieve of Eratosthenes
The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to *n*. Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples.

The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.
Example: n = 25, Prime numbers are: 2,3,5,7,11,13,17,19, 23

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

ALGORITHM *Sieve(n)*
//Implements the sieve of Eratosthenes
//Input: A positive integer $n > 1$
//Output: Array L of all prime numbers less than or equal to n

```
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0          //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0     //mark element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

An algorithm can be specified in
1) Simple English
2) Graphical representation like flow chart
3) Programming language like C/C++/Java
4) Combination of above methods.

Fundamentals of Algorithmic Problem Solving

We can consider algorithms to be procedural solutions to problems.
These solutions are not answers but specific instructions for getting answers.

Sequence of steps one typically goes through in designing and analysing an algorithm.

Understanding the Problem

To understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.

Knowledge of existing algorithms, their strengths and weaknesses enables us to use one of the algorithms or create a new algorithm to solve a problem.

A correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

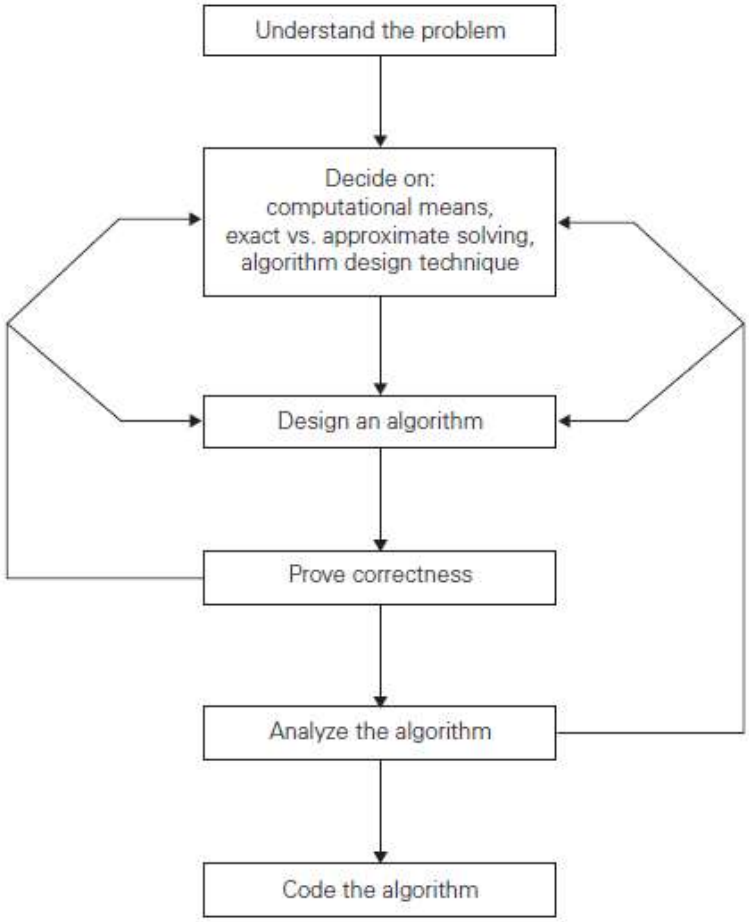


FIGURE 1.2 Algorithm design and analysis process.

Ascertaining the Capabilities of the Computational Device
Understand the computational device the algorithm is intended for.

The majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine.

Sequential algorithms: instructions are executed one after another, one operation at a time.
Parallel algorithms: newer computers can execute operations concurrently, i.e., in parallel.

Should you worry about the speed and amount of memory of a computer?
If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve.

If problems that are very complex by their nature, or have to process huge volumes of data, or deal with applications where the time is critical, it is necessary to know the speed and memory available on a computer system.

Choosing between Exact and Approximate Problem Solving
The next principal decision is to choose between solving the problem exactly or solving it approximately.

exact algorithm: gives exact output
approximation algorithm: gives approximate output

There are important problems that simply cannot be solved exactly for most of their instances. Example: extracting square roots,

Available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.

Algorithm Design Techniques
What is an algorithm design technique?
An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
Example: divide and conquer, decrease and conquer etc..

Understanding existing techniques helps in solving problems. They provide guidance for designing algorithms for new problems.

Designing an Algorithm and Data Structures
Some design techniques can be simply inapplicable to the problem in question.
Sometimes, several techniques need to be combined to solve a problem.

Choose data structures appropriate for the operations performed by the algorithm.
Example: the sieve of Eratosthenes, would run longer if we used a linked list instead of an array in its implementation.

Methods of Specifying an Algorithm

a. **Natural language** has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.

b. **Pseudocode** is a mixture of a natural language and programming language like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more brief algorithm descriptions.

   Declarations of variables is sometimes omitted and indentation is used to show the scope of such statements as for, if, and while. Other symbols used are, an arrow "←" for the assignment operation and two slashes "//" for comments.

c. **Flowchart**, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms.

Algorithm needs to be converted into a computer program written in a particular computer language.

Proving an Algorithm's Correctness

It is necessary to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

Example:
The correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n).*

The simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

Tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively.

To show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

Analyzing an Algorithm

There are two kinds of algorithm efficiency:
time efficiency, indicating how fast the algorithm runs, and
space efficiency, indicating how much extra memory it uses.

Simplicity: depends on the reader.
Example: Euclid's algorithm is simpler than the middle-school procedure for computing gcd*(m, n)*,

Coding an Algorithm

Algorithms are implemented as computer programs.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The term "analysis of algorithms" is usually used in a narrower, technical sense to mean an investigation of an algorithm's efficiency with respect to two resources: running time and memory space.

Analysis Framework

We outline a general framework for analysing the efficiency of algorithms.
there are two kinds of efficiency:

- ***Time efficiency***, also called ***time complexity***, indicates how fast an algorithm in question runs.
- ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Now the amount of extra space required by an algorithm is typically not of as much concern as compared to earlier days when memory was costly.

Measuring an Input's Size

Almost all algorithms run longer on larger inputs. Example, it takes longer to sort larger arrays.

Investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

Choosing n, will be easy for problems of sorting, searching etc.. i.e. size of list.

For the problem of evaluating a polynomial $p(x) = a_n x_n + \ldots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.

In case of matrix multiplication, the choice of a parameter indicating an input size.
The first is the matrix order n.
The other would be total number of elements N in the matrices being multiplied.

Units for Measuring Running Time

Count the number of times the algorithm's ***basic operation*** is executed.

Basic operation:

The operation contributing the most to the total running time.

It can be found the in the algorithm's innermost loop.

Example: for searching, basic operation will be **key comparison** operation.
In algorithms for mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division. Of the four,

the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together.

Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and
let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula,

$$T(n) \approx c_{op} . C(n).$$

The formula can give a reasonable estimate of the algorithm's running time.

Assuming that $C(n) = 1/2 . n(n-1)$, how much longer will the algorithm run if we double its input size?

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

Therefore,

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Four times.

Note that we were able to answer the last question without actually knowing the value of c_{op}.

Orders of Growth

The order of growth of basic operation count of an algorithm is an approximation of the time required to run a computer program as the input size n increases.

To know the efficiency of an algorithms, we will use the algorithm for large inputs n (as efficiency not evident for small number of inputs).

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

The function growing the slowest among these is the logarithmic($\log_2 n$) function. Both exponential function (2^n) and the factorial function ($n!$) grow very fast that their values become astronomically large even for rather small values of *n*.

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

Consider how they react to, say, a twofold increase in the value of their argument n. The function log2 n increases in value by just 1 (because $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$).

The linear function increases twofold, the linearithmic function $n \log_2 n$ increases slightly more than twofold.

The quadratic function n^2 and cubic function n^3 increase fourfold and eightfold, respectively (because $(2n)^2 = 4n^2$ and $(2n)^3 = 8n^3$); the value of 2^n gets squared (because $2^{2n} = (2^n)^2$); and n! increases much more than that (yes, even mathematics refuses to cooperate to give a neat answer for n!).

Worst-Case, Best-Case, and Average-Case Efficiencies

Algorithms efficiency is measured by size of the input and on the specifics of a particular input.

Example: Sequential search

Searches for a given item (some search key *K*) in a list of *n* elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

ALGORITHM SequentialSearch(A[0..n − 1], K)
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
// or −1 if there are no matching elements
i ←0
while i < n **and** A[i] != K **do**
    i ←i + 1
if i < n **return** i
else return −1

The running time of this algorithm can be quite different for the same list size *n*.

Worst Case:
The ***worst-case efficiency*** of an algorithm is its efficiency for the worst-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the longest among all possible inputs of that size.

Find what kind of inputs yield the largest value of the basic operation's count *C(n)* among all possible inputs of size *n* and then compute this worst-case value $C_{worst}(n)$.

Worst case guarantees that for any instance of size *n*, the running time will not exceed $C_{worst}(n)$.

Best case:
The best-case efficiency of an algorithm is its efficiency for the best-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the fastest among all possible inputs of that size.

Determine the kind of inputs for which the count *C(n)* will be the smallest among all possible inputs of size *n*.

Average-case efficiency

Provides algorithm's behaviour on a "typical" or "random" input.

To analyse the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

For Linear Search:
Worst Case: when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n: $C_{worst}(n) = n$.

Best Case: inputs for sequential search are lists of size *n* with their first element equal to a search key; accordingly, $C_{best}(n) = 1$ for this algorithm.

Average Case:
The standard assumptions are that

(a) the probability of a successful search is equal to p $(0 \leq p \leq 1)$ and

(b) the probability of the first match occurring in the i^{th} position of the list is the same for every *i*.

We can find the average number of key comparisons $C_{avg}(n)$ as follows.

In the case of a successful search, the probability of the first match occurring in the i^{th} position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i.

In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1-p)$.
Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

For successful search:
if $p = 1$ (the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$; that is, the algorithm will inspect, on average, about half of the list's elements.

For unsuccessful search:
If $p = 0$ (the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

Amortized Efficiency
In some situation a single operation can be expensive, but the total time for an entire sequence of n such operations is always significantly better than the worst-case efficiency of that single operation multiplied by n. (discovered by the American computer scientist Robert Tarjan)

Summary

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

Asymptotic Notations and Basic Efficiency Classes

To compare and rank orders of growth, computer scientists use three notations: O(big oh), Ω(big omega), Θ (big theta).

Let *t (n)* and *g(n)* can be any nonnegative functions defined on the set of natural numbers. And *t (n)* will be an algorithm's running time (indicated by its basic operation count *C(n)*), and *g(n)* will be some simple function to compare the count with.

Informal Introduction

Informally, *O(g(n))* is the set of all functions with a lower or same order of growth as *g(n)* (to within a constant multiple, as *n* goes to infinity).
Examples:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

The first two functions are linear and hence have a lower order of growth than $g(n) = n^2$, while the last one is quadratic and hence has the same order of growth as n^2.

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

Indeed, the functions n^3 and $0.00001n^3$ are both cubic and hence have a higher order of growth than n^2, and so has the fourth-degree polynomial $n^4 + n + 1$.

The second notation, Ω *(g(n))*, stands for the set of all functions with a higher or same order of growth as *g(n)*.

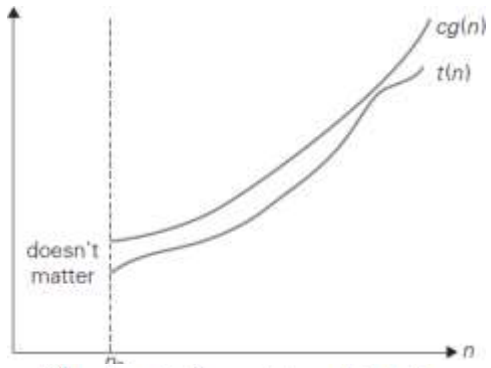
$$n^3 \in \Omega(n^2), \qquad \frac{1}{2}n(n-1) \in \Omega(n^2), \qquad \text{but } 100n + 5 \notin \Omega(n^2)$$

Finally, Θ *(g(n))* is the set of all functions that have the same order of growth as *g(n)*. Thus, every quadratic function $an^2 + bn + c$ with $a > 0$ is in $\Theta(n^2)$.

Big-O notation

Definition: A function t(n) is said to be in O(g(n)),denoted t(n) ∈ O(g(n)), if t(n) is bounded above bysome constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$t(n) \le cg(n)$ for all $n \ge n_0$



Big-oh notation: $t(n) \in O(g(n))$.

Strategies to prove Big-O: Sometimes the easiest way to prove that f(n) = O(g(n)) is to take c to be the sum of the positive coefficients of f(n). We can usually ignore the negative coefficients.

Example: To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$,

$$5n^2 + 3n + 20 \leq 5n^2 + 3n^2 + 20n^2 = 28n^2,$$

thus $5n^2 + 3n + 20 = O(n^2)$.

Example 2: To prove **$100n + 5 \in O(n^2)$**
$100n + 5 \leq 105n^2$. *($c=105, n_0=1$)*

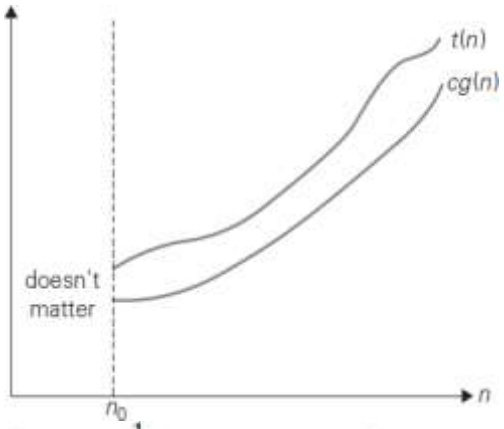
Example 3: To prove **$n^2 + n = O(n^3)$**
Take c = 1+1=2, if $n \geq n_0=1$, then $n^2 + n = O(n^3)$

Exercise i) Prove $3n+2=O(n)$ ii) Prove $1000n^2+100n-6 = O(n^2)$

Omega notation

Definition: A function t(n) is said to be in Ω(g(n)),denoted ***t(n)*∈ *Ω(g(n))*,** if t(n) is bounded below bysome positive constant multiple of g(n) for all largen,i.e., if there exist some positive constant c and some nonnegative integer n_0 such that ***t(n)* ≥ *c g(n)*** for all $n \geq n_0$.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$: $n^3 \geq n^2$ for all $n \geq 0$, i.e., we can select c = 1 and $n_0 = 0$.



Example: $n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$, but $100n + 5 \notin \Omega(n^2)$.

Example: To prove $n^3 + 4n^2 = \Omega(n^2)$

We see that, if $n \geq 0$, $n^3+4n^2 \geq n^3 \geq n^2$;

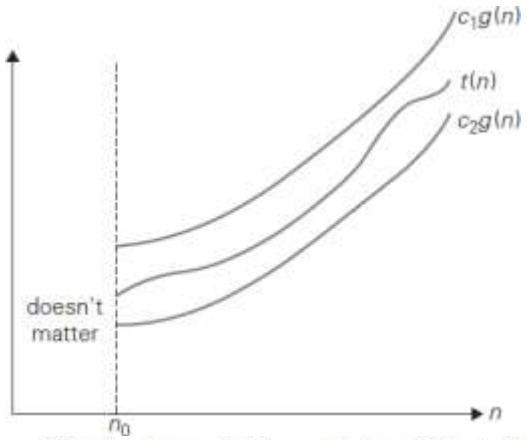
Therefore $n^3+4n^2 \geq 1n^2$ for all $n \geq 0$.

Thus, we have shown that $n^3+4n^2 = \Omega(n^2)$ where $c = 1$ & $n_0=0$

Theta notation

A function t(n) is said to be in **Θ(g(n)),** denoted **t(n) ∈ Θ (g(n)),** if t (n) is bounded both above and belowby some positive constant multiples ofg(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some non negative integer n_0 such that

$c_2g(n) \leq t(n) \leq c_1g(n)$ for all $n \geq n_0$.



Big-theta notation: $t(n) \in \Theta(g(n))$

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

When $n \geq 1$,

$$n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$$

When $n \geq 0$,

$$n^2 \leq n^2 + 5n + 7$$

Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Strategies for Ω and Θ

- Proving that a f(n) = Ω(g(n)) often requires more thought.
  - Quite often, we have to pick c < 1.
  - A good strategy is to pick a value of c which you think will work, and determinewhich value of n_0 is needed.
  - We can sometimes simplify by ignoring terms of f(n) with the positive coefficients.
- The following theorem shows us that proving f(n) = Θ(g(n)) is nothing new:

  Theorem: f(n) = Θ(g(n)) if and only iff(n) = O(g(n)) and f(n) = Ω(g(n)).

Thus, we just apply the previous two strategies.

Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$

Notice that if $n \geq 1$,

$$\frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n^2 = \frac{7}{2}n^2$$

Thus,

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

Also, when $n \geq 0$,

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n$$

So

$$\frac{1}{2}n^2 + 3n = \Omega(n^2)$$

Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,

$$\frac{1}{2}n^2 + 3n = \Theta(n^2)$$

We need to find positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0$$

Dividing by n^2, we get

$$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

$c_1 \le \frac{1}{2} - \frac{3}{n}$ holds for $n \ge 10$ and $c_1 = 1/5$

$\frac{1}{2} - \frac{3}{n} \le c_2$ holds for $n \ge 10$ and $c_2 = 1$.

Thus, if $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 10$, then for all $n \ge n_0$,

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0.$$

Thus we have shown that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Theorem: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.(The analogous assertions are true for the Ω and Θ notations as well.)

Proof: The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer $n_1$such that $t_1(n) \le c_1 g_1(n)$ for all $n \ge n_1$.

Similarly, since $t_2(n) \in O(g_2(n))$, $t_2(n) \le c_2 g_2(n)$ for all $n \ge n_2$.

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3\, g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

Little Oh

The function **f(n)= o(g(n))** [i.e f of n is a little oh of g of n] if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example: The function $3n + 2 = o(n^2)$ since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \ne o(n)$. $6 * 2^n + n^2 \ne o(2^n)$. □

For comparing the order of growth limit is used

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n),\\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n),\\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).\end{cases}$$

If the case-1 holds good in the above limit, we represent it by little-oh.

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1)\in\Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=0$, we can use the so-called ***little-oh notation***: $\log_2 n\in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.) ■

Basic asymptotic efficiency Classes

Class	*Name*	*Comments*
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.

n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Mathematical Analysis of Non recursive Algorithms

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

Summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Analysis of algorithm MaxElement

1. Measure of an input's size here is the number of elements in the array, i.e., n.
2. Basic operation: A[i]> maxval (executed the most number of times)
3. The number of comparisons will be the same for all arrays of size n, there is no need to distinguish among the worst, average, and best cases here.

   Let C(n) the number of times this comparison is executed for an array of size n. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive.
   Hence C(n) is defined as

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

   Hence the order of growth is Θ(n).

EXAMPLE 2 Consider the element uniqueness problem: check whether all the elements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Analysis of algorithm UniqueElements

1. Measure of the input's size here is *n*, the number of elements in the array.
2. Basic operation: $A[i] == A[j]$ (executed the most number of times)
3. The number of element comparisons depends on input instance. Hence need to find best case and works case.

   Worst case inputs: arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. Example: 10 20 30 40 50 and 10 20 30 40 40.

   Basic operation count C(n) worst is defined as:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2).

EXAMPLE 3 Given two n × n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C = AB. By definition, C is an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:

where C[i, j]= A[i, 0]B[0, j]+ . . . + A[i, k]B[k, j]+ . . . + A[i, n − 1]B[n − 1, j]

for every pair of indices 0 ≤ i, j ≤ n – 1 .

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Analysis of MartixMultiplication

1. Measure an input's size by matrix order n.
2. Basic operation: Multiplication or addition can be considered as basic operation as both statements will be executed same number of times in

   C[i, j]←C[i, j]+ A[i, k] ∗ B[k, j].
3. The basic operation count does not vary on input instance. Hence find worst case only.
4. Let us set up a sum for the total number of multiplications M(n) executed by the algorithm.
   Multiplication operation is executed once in the innermost loop(k) for an iteration. The innermost loop(k) is executed for j = 0 to n-1 and enclosing loop(i) also executes for I = 0 to n-1 times. Hence total number of multiplications M(n) and the order of growth is:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

EXAMPLE 4 The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

The basic operation is count=count + 1 repeats $\lfloor \log_2 n \rfloor + 1$ no. of times.

Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1 Compute the factorial function F(n) = n! for an arbitrary nonnegative integer n.

Since n!= 1 . . . . . (n − 1) . n = (n − 1)! . n for n ≥ 1 and 0!= 1 by definition,

we can compute F(n) = F(n − 1) . n with the following recursive algorithm.

ALGORITHM $F(n)$

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) * n
```

Analysis of algorithm

1. Measurement of input size is n.
2. Basic operation: multiplication F(n-1) * n.
3. Let M(n)denote total number of multiplications required to compute factorial of n.
4. The factorial function F(n) itself; it is defined by the recurrence:

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

Recurrence relation and initial condition for the algorithm's number of multiplications M(n):

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

Simplify the recurrence relation M(n) using method of *backward substitutions*.

$$\begin{aligned} M(n) &= M(n-1)+1 && \text{substitute } M(n-1)=M(n-2)+1 \\ &= [M(n-2)+1]+1 = M(n-2)+2 && \text{substitute } M(n-2)=M(n-3)+1 \\ &= [M(n-3)+1]+2 = M(n-3)+3. \end{aligned}$$

i^{th} term

$$M(n) = M(n - i) + i.$$

substitute i = n in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n-1)+1 = \cdots = M(n-i)+i = \cdots = M(n-n)+n = n.$$

EXAMPLE 2 Tower of Hanoi

There are n disks on first peg. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. Constraints: only one disk can be moved at any point of time and larger disk cannot be placed on smaller disk.

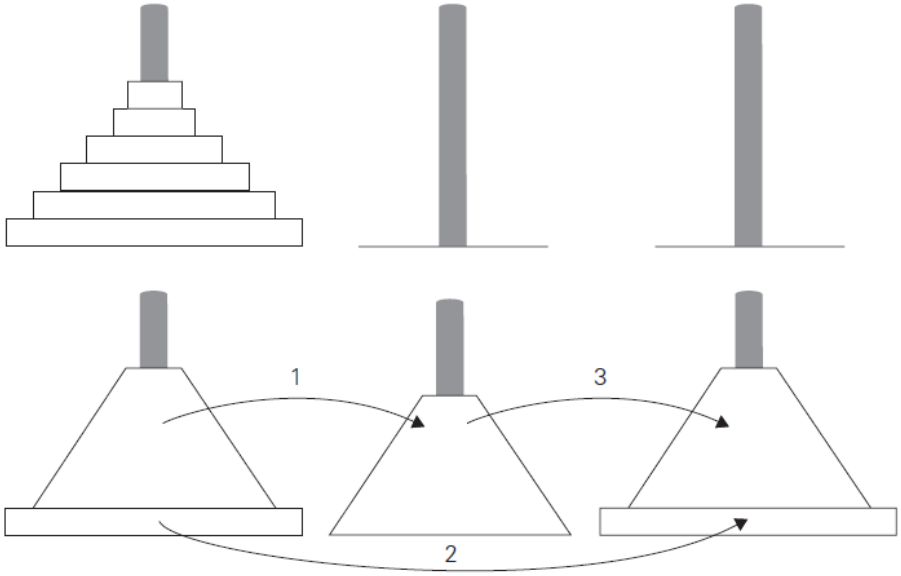


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

The problem has an elegant recursive solution

- To move n>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary),
  - we first move recursively n-1 disks from peg 1 to peg 2 (with peg 3 as auxiliary),
  - then move the largest disk directly from peg 1 to peg 3, and,
  - finally, move recursively n-1 disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- If n = 1, we move the single disk directly from the source peg to the destination peg.

Algorithm TowerOfHanoi(n, source, dest, aux)

```
If n == 1, then
        move disk from source to dest
else
{
        TowerOfHanoi (n - 1, source, aux, dest)
        move disk from source to dest
        TowerOfHanoi (n - 1, aux, dest, source)
}
```

Algorithm TowerofHanoi(n, source, temp, destination)
//Move n disks from source to destination
//Input: n, number of disks
//Output: n disks moved to destination
If n == 1 move disk from source to destination
else
        TowerofHanoi(n-1, source, destination, temp)
        move n^{th} disk from source to destination
        TowerofHanoi(n-1, temp, source, destination)

Analysis of TowerOfHanoi

Measurement of input size is n, no. of disks
Basic operation: moving disk from one peg to another
Let *M(n)* represent the total number of moves made. And it depends on *n* only.
Recurrence equation for M(n):

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1.$$

i.e. to move n disks from source to destination,

M(n-1) : move n-1 disk from source to temp
1 : move disk from source to destination
M(n-1) : move n-1 disk from temp to destination

Hence the equation M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1

Solve this recurrence by the same method of backward substitutions

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1.$$

after *i* substitutions, we get

$$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence

$$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Formula:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$

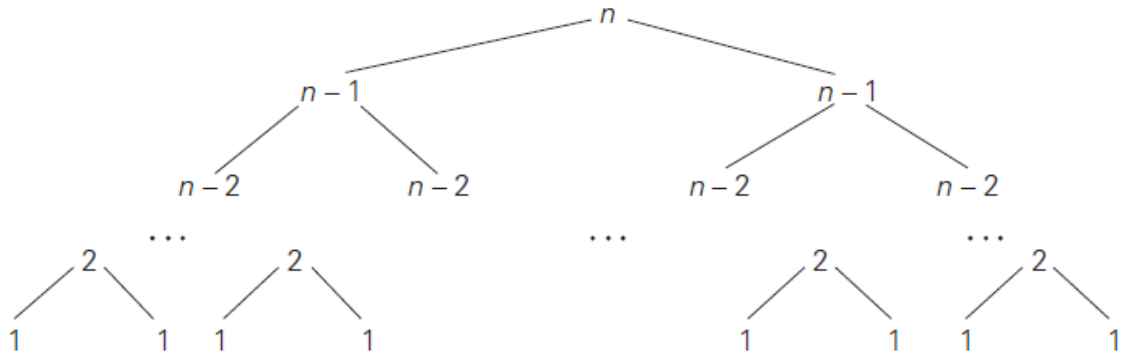


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

EXAMPLE 3 Find the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

Analysis of BinRec

1. Measurement of input size is n, the decimal number.
2. Basic operation: Addition
3. The algorithm's basic operation does not vary on input instance. Hence compute worst case analysis.

Let A(n) represent total number of additions required for n.

The number of additions made in computing $BinRec(\lfloor n/2 \rfloor)$ is $A(\lfloor n/2 \rfloor)$. plus one more addition is made by the algorithm to increase the returned value by 1.

The recurrence relation is as shown below.

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is
A(1) = 0.

Let $n = 2^k$

Modified recurrence relation is

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
&\dots \\
&= A(2^{k-i}) + i \\
&\dots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

BRUTE FORCE APPROACHES

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. Brute force approach is simple but has inferior efficiency.
Example: Selection sort, Bubble sort.

Selection Sort

Start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.

Find the smallest element in the remaining elements and swap it with second element. Repeat this for n-1 times where n is number of elements.

Generally, on the ith pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

After $n-1$ passes, the list is sorted.

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Example: sort the following elements in ascending order 89, 45, 68, 90, 29, 34, 17

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Analysis

The input size is given by the number of elements *n*; the basic operation is the key comparison $A[j\,] < A[min]$.

The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Bubble Sort

Bubble sort compares adjacent elements of the list and exchange them if first element is greater than second element compared. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted.

Pass i ($0 \le i \le n - 2$) of bubble sort can be represented by the following diagram.

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \cdots \le A_{n-1}$$
in their final positions

Example: sort the following elements in ascending order 89, 45, 68, 90, 29, 34, 17

89 ?↔ 45 68 90 29 34 17
45 89 ?↔ 68 90 29 34 17
45 68 89 ?↔ 90 ?↔ 29 34 17
45 68 89 29 90 ?↔ 34 17
45 68 89 29 34 90 ?↔ 17
45 68 89 29 34 17 | 90

45 ?↔ 68 ?↔ 89 ?↔ 29 34 17 | 90
45 68 29 89 ?↔ 34 17 | 90
45 68 29 34 89 ?↔ 17 | 90
45 68 29 34 17 | 89 90

etc.

FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

ALGORITHM *BubbleSort*$(A[0..n-1])$

```
//Sorts a given array by bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    for j ← 0 to n − 2 − i do
        if A[j + 1] < A[j] swap A[j] and A[j + 1]
```

Analysis

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

We can improve the crude version of bubble sort given above by exploiting the following observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm.

Though the new version runs faster on some inputs, it is still in $\Theta(n^2)$ in the worst and average cases.

Sequential Search(Enhanced)

The algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

First improvement to simple sequential search is to append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether ($i < n$).

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

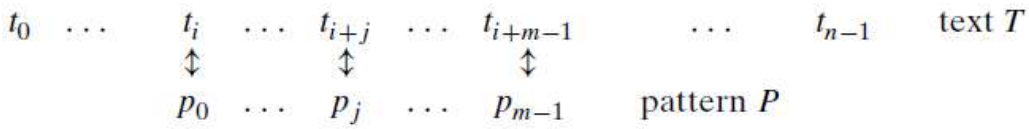
Improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

The algorithm remains linear in both the worst and average cases.

Brute Force String Matching

Given a string of *n* characters called the *text* and a string of m characters (m ≤ n) called the *pattern*, find a substring of the text that matches the pattern.

We want to find *i* the index of the leftmost character of the first matching substring in the text.



Example: Find Pattern = “NOT” in Text = “NOBODY_NOTICED_HIM”.

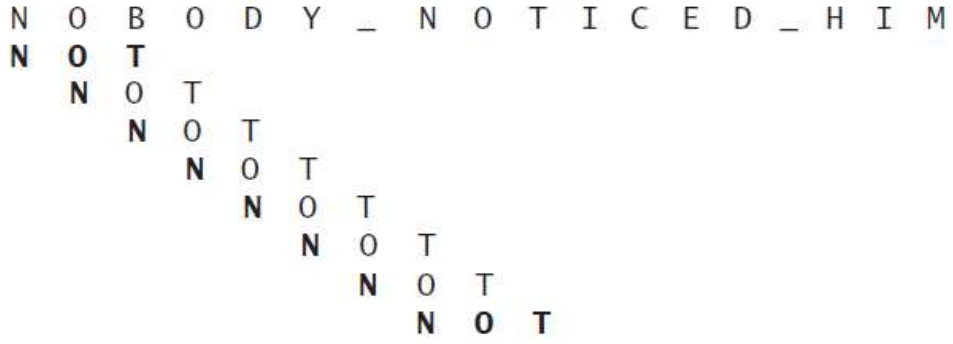


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Working

Align the pattern against the first *m* characters of the text and start matching the corresponding pairs of characters from left to right until either all the *m* pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.

Note that the last position in the text that can still be a beginning of a matching substring is $n - m$ (provided the text positions are indexed from 0 to $n - 1$).
Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

Analysis

Worst case: the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries.

Thus the algorithm makes $m(n - m + 1)$ character comparisons, which puts it in the $O(nm)$ class.

Module 2

Syllabus

BRUTE FORCE APPROACHES (contd..): Exhaustive Search (Travelling Salesman problem and Knapsack Problem).
DECREASE-AND-CONQUER: Insertion Sort, Topological Sorting.
DIVIDE AND CONQUER: Merge Sort, Quick Sort, Binary Tree Traversals, Multiplication of Large Integers and Strassen's Matrix Multiplication.

Chapter 3(Section 3.4), Chapter 4 (Sections 4.1,4.2), Chapter 5 (Section 5.1,5.2,5.3, 5.4)

BRUTE FORCE APPROACHES (contd..)

Exhaustive Search

Exhaustive search is simply a brute-force approach to combinatorial problems. Works by generating all possible combination of solutions that satisfy all the constraints and selecting the best solution.
Example: Travelling Salesman problem, Knapsack problem.

Travelling Salesman problem (TSP)

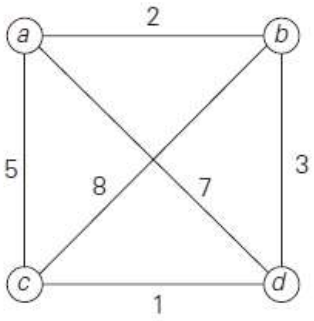
TSP means, find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

TSP is modelled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.

Then the problem can be stated as the problem of finding the shortest *Hamiltonian circuit* of the graph.
(A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805–1865)).

Hamiltonian circuit can also be defined as a sequence of $(n + 1)$ adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct.

We can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them.



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search.

The total number of permutations needed is still 1/2 * (n − 1)!.

Algorithm is suitable only for very small values of *n*.

Knapsack Problem

Given *n* items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack.

Application: transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

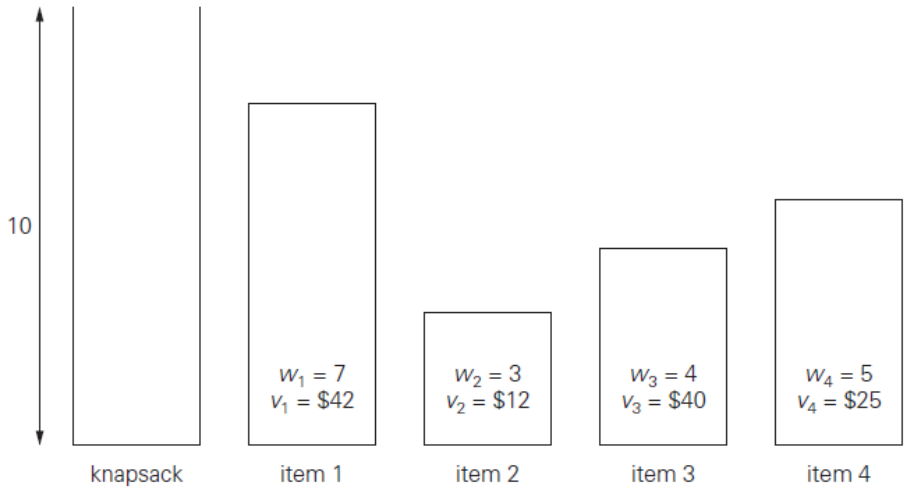


Figure 3.8 a: Instance of the knapsack problem.

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Figure 3.8 b: Its solution by exhaustive search. The information about the optimal selection is in bold.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

The number of subsets of an *n*-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm,

DECREASE-AND-CONQUER

The decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

The relationship between problem instances can be top down or bottom up.

- Top down relationship can be realised as recursive algorithm or sometimes non recursive algorithm.
- Bottom up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the *incremental approach*.

There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

Decrease-by-a-constant

The size of an instance is reduced by the same constant on each iteration of the algorithm.

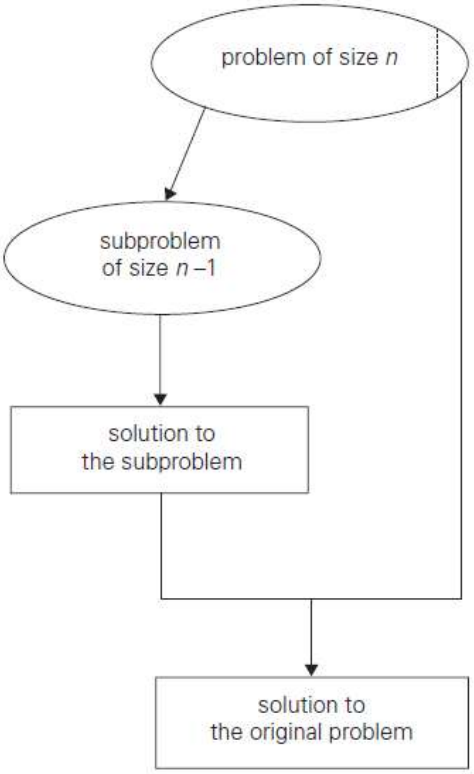


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Example: the exponentiation problem of computing a^n where $a != 0$ and n is a nonnegative integer.

function f (n) = a^n can be computed either "top down" by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

or "bottom up" by multiplying 1 by *a*, *n* times.

Decrease-by-a-constant-factor
reduces a problem instance by the same constant factor on each iteration of the algorithm.
Example: Binary Search

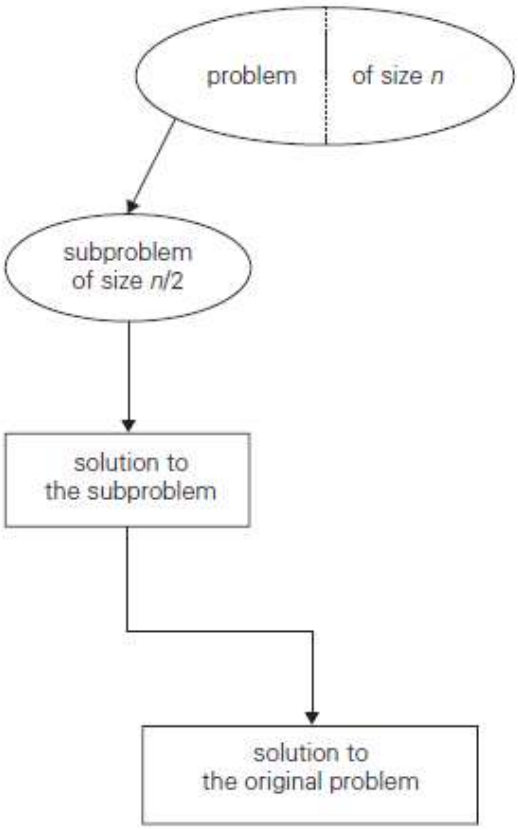


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Variable-size-decrease
The size-reduction pattern varies from one iteration of an algorithm to another.
Example: Euclid's algorithm

Insertion Sort

Insertion sort is an example of decrease-by-one technique.

Works by inserting a new element in its position in a sorted in each iteration. (Starts from second element).

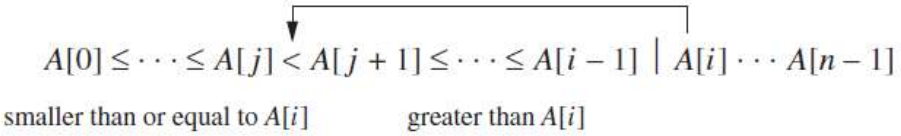


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

All the elements to left of A[i] is sorted and to its right are not sorted.

We need to place A[i] in its correct position in sorted list.
Step1: Compare A[i] with elements in sorted array starting from A[i-1].
Step2: Once correct position of A[i] is found, copy A[i] to temp variable, shift elements greater than A[i] to right (in sorted list), then copy temp to the correct position.
Step 3: Repeat Setp1 and Step 2 for i = 1 to n-1.

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Analysis of Insertion sort

Worst Case

Basic operation is A[j]> v in the algorithm.

The worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

In the worst case, insertion sort makes exactly the same number of comparisons as selection sort.

Best Case:

Input instance: sorted elements in increasing order.

The comparison $A[j\,] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \leq A[i]$ for every $i = 1, \ldots, n-1$.

The number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Topological Sorting

Topological sorting is a linear ordering of vertices in a Directed Acyclic Graph (DAG) such that for every directed edge uv, vertex u comes before vertex v in the ordering.

Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the course prerequisites are met.
(C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4.)

In which order should the student take the courses?
This problem is called topological sorting.

Application: instruction scheduling in program compilation, cell evaluation ordering in spreadsheet formulas, and resolving symbol dependencies in linkers.

Solution 1: Using DFS

Perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem.

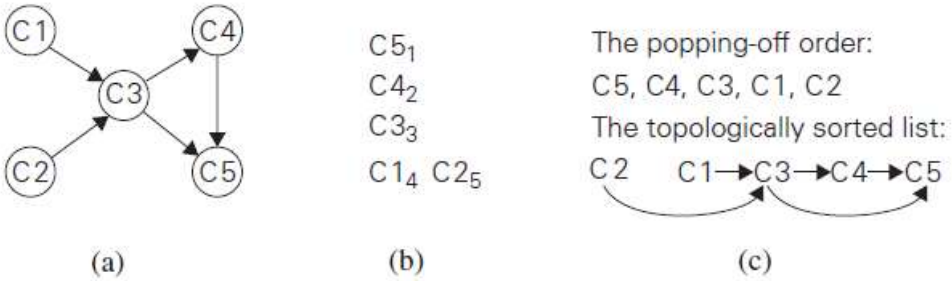


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Solution 2: Source-removal algorithm

Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.

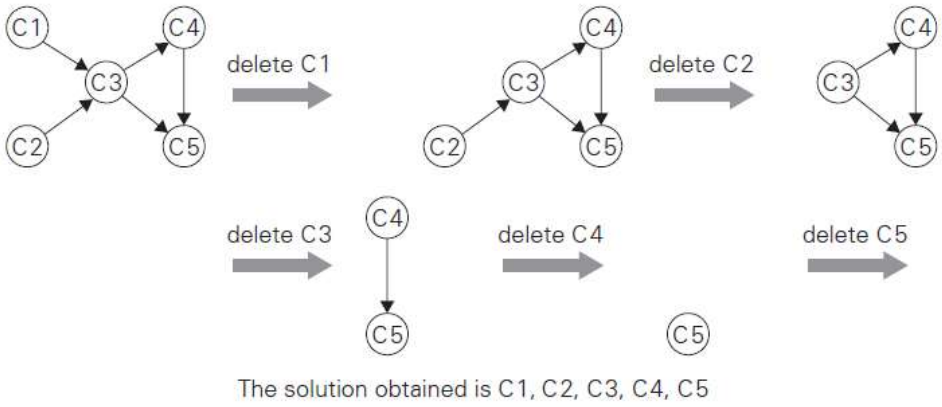


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Topological sort is an implementation of the decrease-(by one)-and-conquer technique.

The problem is modelled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements.

A directed graph, or digraph for short, is a graph with directions specified for all its edges. The adjacency matrix and adjacency lists are two means of representing a digraph.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs.

Types of edges:

- **tree edges** (ab, bc, de),
- **back edges** (ba) from vertices to their ancestors,
- **forward edges** (ac) from vertices to their descendants in the tree other than their children,
- **cross edges** (dc), which are none of the aforementioned types.

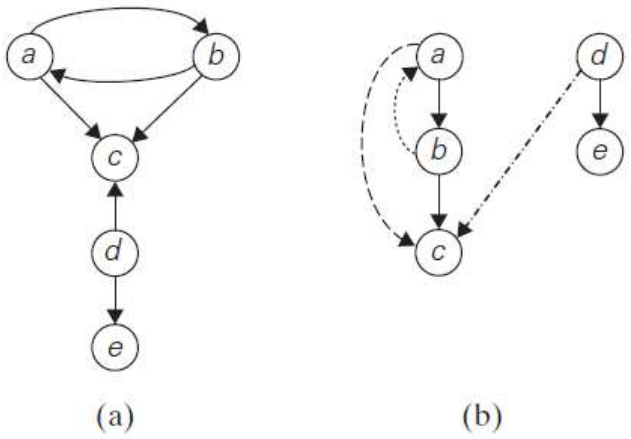


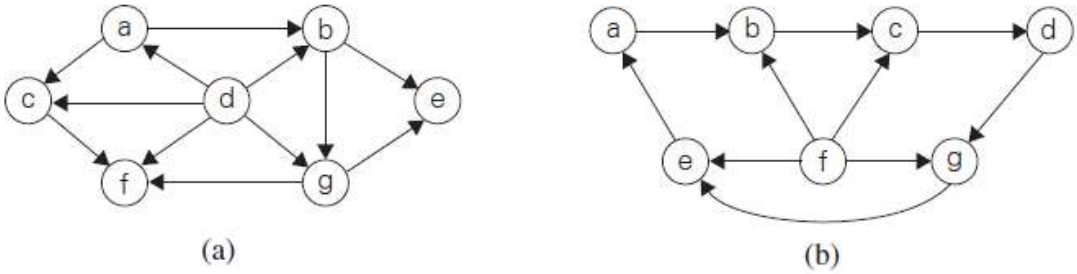
FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

The presence of a back edge indicates that the digraph has a directed cycle.

A directed cycle is a path that starts at a given vertex and returns to the same vertex, traversing each edge in the direction it is oriented.

A Directed Acyclic Graph (DAG) is a graph that is directed and contains no cycles.

Q) Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



DIVIDE AND CONQUER

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several sub-problems of the same type, ideally of about equal size.
2. The sub-problems are solved (typically recursively, though sometimes a different algorithm is employed, especially when sub-problems become small enough).
3. If necessary, the solutions to the sub-problems are combined to get a solution to the original problem.

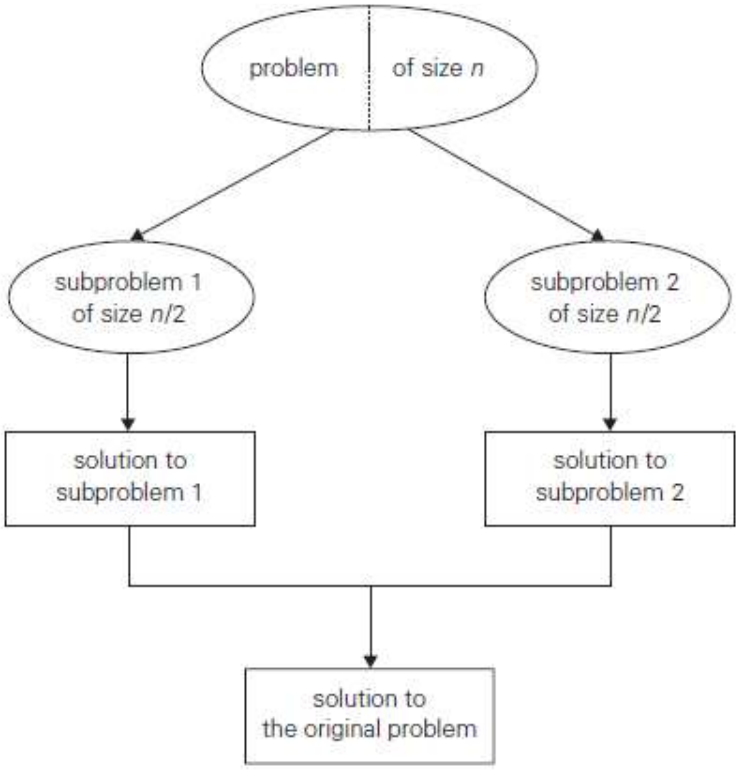


FIGURE 5.1 Divide-and-conquer technique (typical case).

General divide-and-conquer recurrence

Divide-and-conquer a problem's instance of size n is divided into two instances of size n/2. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.).

Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time T (n):

$$T(n) = aT(n/b) + f(n)$$

where f (n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions.

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

Merge Sort

Merge sort, sorts a given array A[0..n − 1] by dividing it into two halves A[0.._n/2_ − 1] and A[_n/2_..n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

The merging of two sorted arrays can be done as follows.

- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
- This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Example: Sort the following elements in ascending order: 8, 3, 2, 9, 7, 1, 5, 4

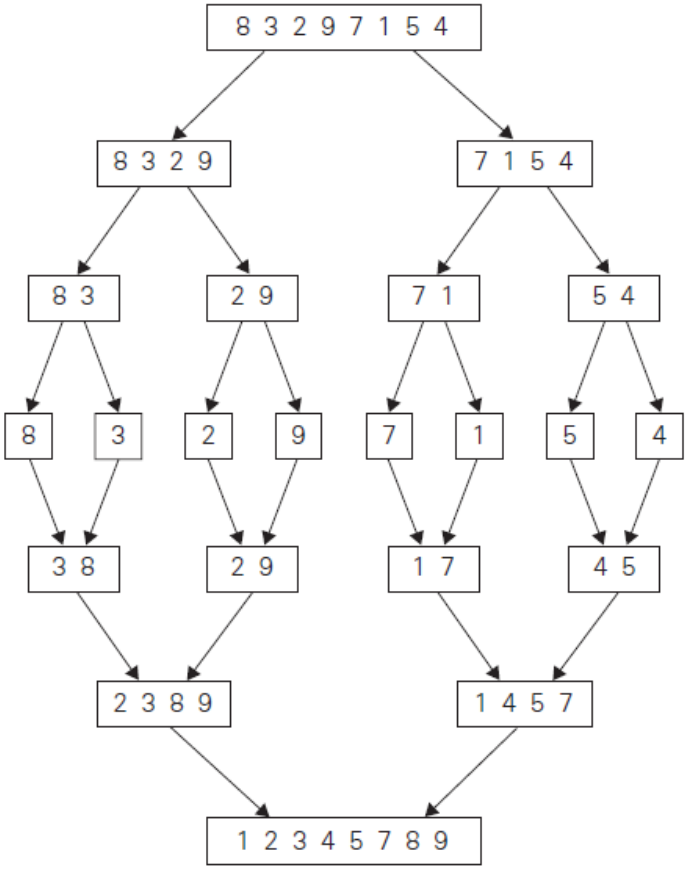


FIGURE 5.2 Example of mergesort operation.

Analysis:

Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made for B[i] ≤ C[j].

In the worst case, neither of the two arrays becomes empty before the other one contains just one element. Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence,

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Hence, according to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$

Quick Sort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike merge sort, which divides its input elements according to their position in the array, quicksort divides (or partitions) them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method).

In quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the sub-problems.

ALGORITHM *Quicksort*$(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//       indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ← Partition(A[l..r]) //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

Partitioning

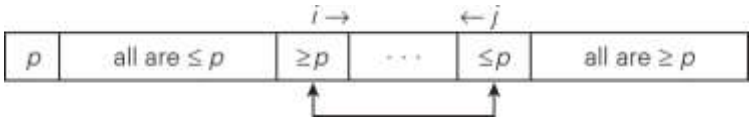
We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot. We use the sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.

Select the subarray's first element: p = A[l]. Now scan the subarray from both ends, comparing the subarray's elements to the pivot.

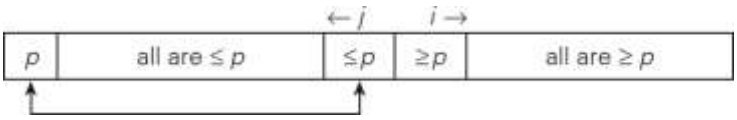
- The **left-to-right scan**, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
- The **right-to-left scan**, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or notthe scanning indices have crossed.

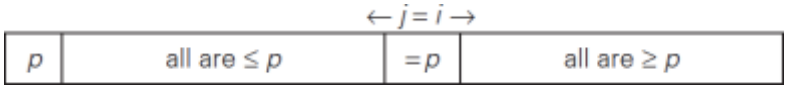
1. If scanning indices i and j have not crossed, i.e., i< j, we simply exchange A[i] and A[j] and resume the scans by incrementing I and decrementing j, respectively:



2. If the scanning indices have crossed over, i.e., i> j, we will have partitioned the subarray after exchanging the pivot with A[j]:



3. If the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned, with the split position s = i = j :



We can combine this with the case-2 by exchanging the pivot with A[j] whenever i ≥ j

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//        as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

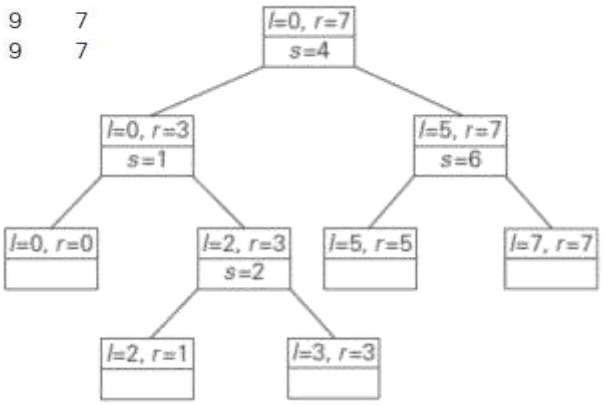
Note that index i can go out of the subarray's bounds in this pseudocode.

Example: quicksort operation.

(a) Array's transformations with pivots shown in bold.

(b) Tree of recursive calls to Quicksort with input values land r of subarray bounds and split position s of a partition obtained.

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9



Analysis

The number of key comparisons made before a partition is achieved is n + 1 if the scanning indices cross over and n if they coincide.

Best case:

All the splits happen in the middle of corresponding subarrays.
The number of key comparisons in the best case satisfies the recurrence.

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

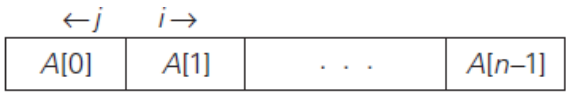
According to the Master Theorem $C_{best}(n) \in \_(n \log_2 n)$.

Solving using backward substitution:
Let $n = 2^k$, yields $C_{best}(n) = n \log_2 n$.

Worst Case:

All the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned.
Input instance is already sorted.

So, after making n + 1 comparisons to get to this partition and exchanging the pivot A[0] with itself, the algorithm will be left with the strictly increasing array A[1..n − 1] to sort. Continue until the last one A[n − 2..n − 1] has been processed.



The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Binary Tree Traversals

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root.

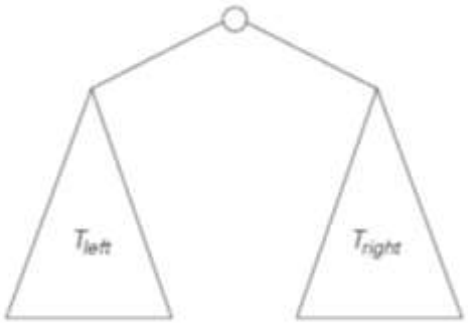


FIGURE 5.4 Standard representation of a binary tree.

Divide and conquer technique can be applied for traversal of binary tree and computing height of the tree.

Computing the height of a binary tree (Recursive algorithm)

Height is defined as the length of the longest path from the root to a leaf.
Hence, it can be computed as the maximum of the heights of the root's left and right subtrees plus 1.

Example:

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

Analysis

Measure the problem's instance size by the number of nodes *n(T)* in a given binary tree T .
Basic operation: finding max of left and right sub tree or addition.
Basic operation count: Number of comparisons made to compute the maximum of two numbers or the number of additions *A(n(T))* made by the algorithm are the same.

The *recurrence relation* for number of additions *A(n(T))*:

$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1 \quad \text{for } n(T) > 0,$$
$$A(0) = 0.$$

Addition is not the most frequently executed operation of this algorithm, it is Checking that the tree is not empty.

Example:
For the empty tree, comparison T = ∅ is executed once, number of additions is zero.
For a single-node tree, number of comparisons is 3, number of additions is one.

For ease of analysis of tree algorithms, replace empty subtree by special nodes.
The extra nodes (little squares in Figure 5.5) are called external; the original nodes (little circles) are called internal.

By definition, the extension of the empty binary tree is a single external node.

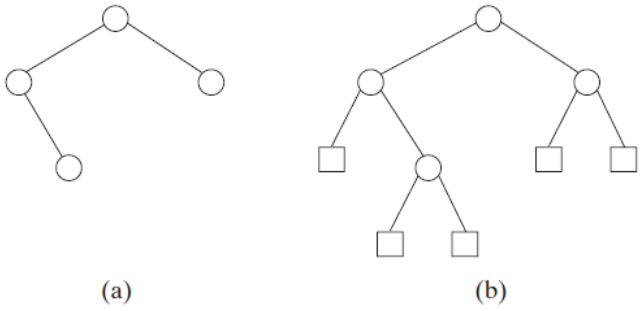


FIGURE 5.5 Binary tree (on the left) and its extension (on the right). Internal nodes are shown as circles; external nodes are shown as squares.

The Height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node.

Algorithm's efficiency, we need to know how many external nodes an extended binary tree with *n* internal nodes can have. The number of external nodes x is always 1 more than the number of internal nodes *n*: $x = n + 1$.

Returning to algorithm Height, the number of comparisons to check whether the tree is empty is

$$C(n) = n + x = 2n + 1,$$

and the number of additions is

$$A(n) = n.$$

Tree Traversal (Recursive)
Three traversals are preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees.

preorder traversal(NLR): the root is visited before the left and right subtrees are visited (in that order).
inorder traversal (LNR): the root is visited after visiting its left subtree but before visiting the right subtree.
postorder traversal(LRN): the root is visited after visiting the left and right subtrees (in that order).

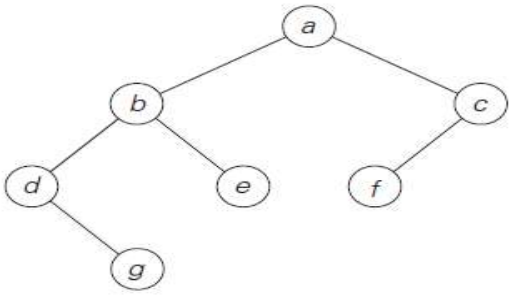


FIGURE 5.6 Binary tree and its traversals.

Multiplication of Large Integers

Some applications, like cryptography, require manipulation of integers that are over 100 decimal digits long.

conventional approach
for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the *n* digits of the second number for the total of n^2 digit multiplications.

Divide-and-conquer approach
multiply two-digit integers, say, 23 and 14.
These numbers can be represented as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \quad \text{and} \quad 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

let us multiply them:

$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0. \end{aligned}$$

but it uses the same four digit multiplications as the pen-and-pencil algorithm.

We can compute the middle term with just one-digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed.

$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$

For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the *a*'s digits and the sum of the *b*'s digits minus the sum of c_2 and c_0.

Apply this trick to multiplying two n-digit integers *a* and *b* where n is a positive even number.

Let us divide both numbers in the middle (using divide-and-conquer)
We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively.

In these notations, $a = a_1 a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1 b_0$ implies that $b = b_1 10^{n/2} + b0$.

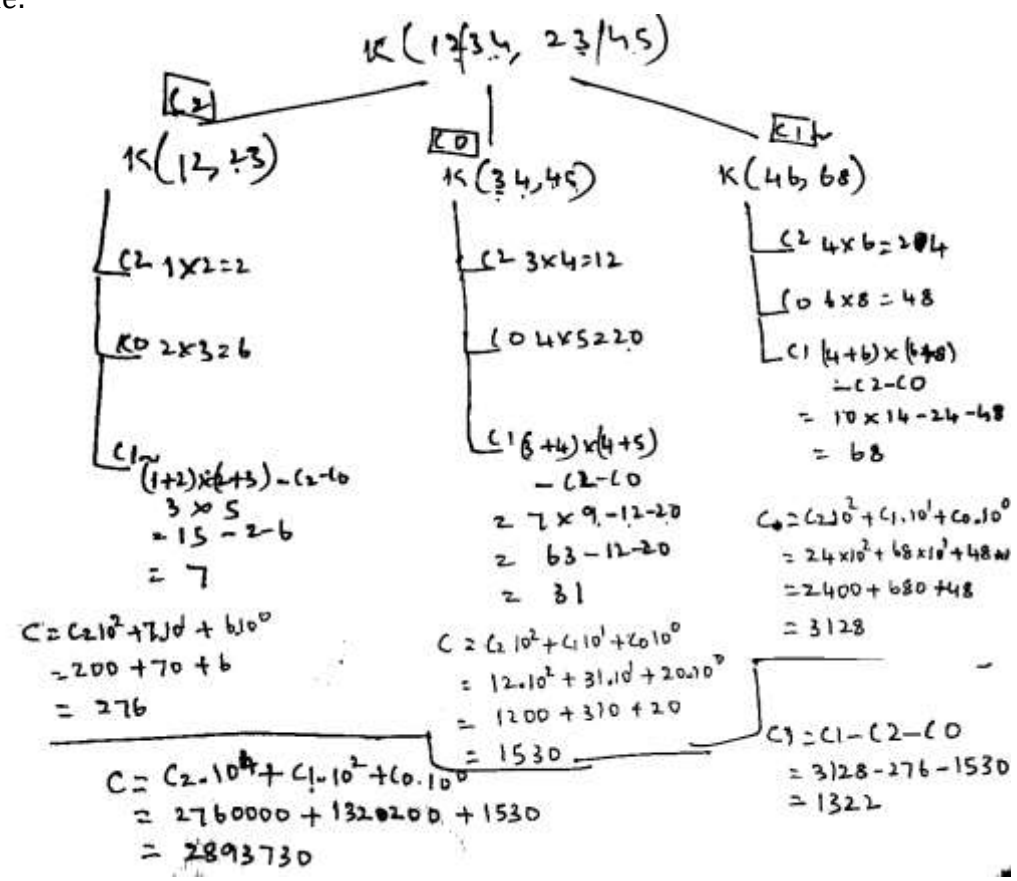
Taking advantage of the same trick we used for two-digit numbers, we get,

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,
$c_0 = a_0 * b_0$ is the product of their second halves,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Example:



Analysis

Multiplication of n-digit numbers requires three multiplications of *n/2* digit numbers, the recurrence for the number of multiplications M(n) is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2})$$
$$= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k.$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

Strassen's Matrix Multiplication

An algorithm was published by V. Strassen in 1969 for matrix multiplication using divide and conquer. We can find the product C of two 2 × 2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm.

This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

Where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions.

Analysis

If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n × n matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \cdots$$
$$= 7^iM(2^{k-i}) \cdots = 7^kM(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

which is smaller than n^3 required by the brute-force algorithm.

To multiply

- two matrices of order n>1, the algorithm needs to
  - multiply seven matrices of order *n/2* and
  - make 18 additions/subtractions of matrices of size *n/2*;
- when n = 1, no additions are made since two numbers are simply multiplied.

The number of additions A(n) made by Strassen's algorithm.

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0.$$

According to the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$.

The number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

Module-3

TRANSFORM-AND-CONQUER: Balanced Search Trees, Heaps and Heapsort.
SPACE-TIME TRADEOFFS: Sorting by Counting: Comparison counting sort, Input Enhancement in String Matching: Horspool's Algorithm.

Chapter 6 (Sections 6.3,6.4), Chapter 7 (Sections 7.1,7.2)

Module-3

TRANSFORM-AND-CONQUER

Transform-and-conquer methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

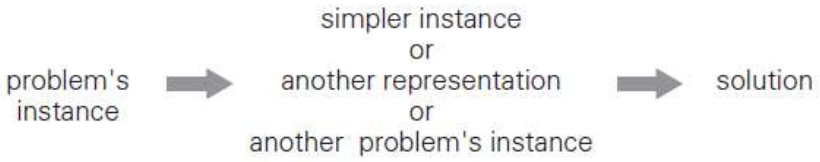


FIGURE 6.1 Transform-and-conquer strategy.

Balanced Search Trees

Binary Search Tree(BST) is a binary tree all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.

Binary search tree is an example of the representation-change technique.

BST time efficiency of searching, insertion, and deletion, which are all in $\theta(\log n)$, but only in the average case. In the worst case, these operations are in $\theta(n)$.

To improve the worst case efficiency of BST, following approaches can be used.

- The unbalanced binary search tree is transformed into a balanced one, such trees are called self-balancing.
  Example: AVL Tree, Red-black tree

  An AVL tree requires the difference between the heights of the left and right subtrees of every node never exceed 1 i.e {-1, 0, 1}.

  A red-black tree tolerates the height of one subtree being twice as large as the other subtree of the same node.

  Balance can be restored using rotations.

- The second approach is of the representation-change variety: allow more than one element in a node of a search tree.
  Example: 2-3 trees, 2-3-4 trees, and B-trees.

They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced.

AVL Trees

AVL trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis.

DEFINITION

An AVL tree is a binary search tree in which the balance factor of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1.

(The height of the empty tree is defined as -1. Of course, the balance factor can also be computed as the difference between the numbers of levels rather than the height difference of the node's left and right subtrees.)

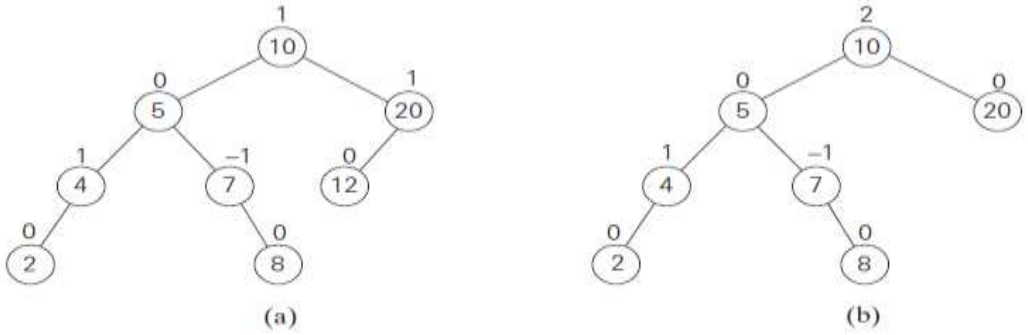


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

Operations

Inserting a new node, Deleting a node, Searching a node

Rotations

If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.

A **rotation** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2.

If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

Types of Rotations

i. R-rotation (single right rotation)
ii. L-rotation(single left rotation)
iii. LR rotation (double left-right rotation)
iv. RL-rotation (double right-left rotation)

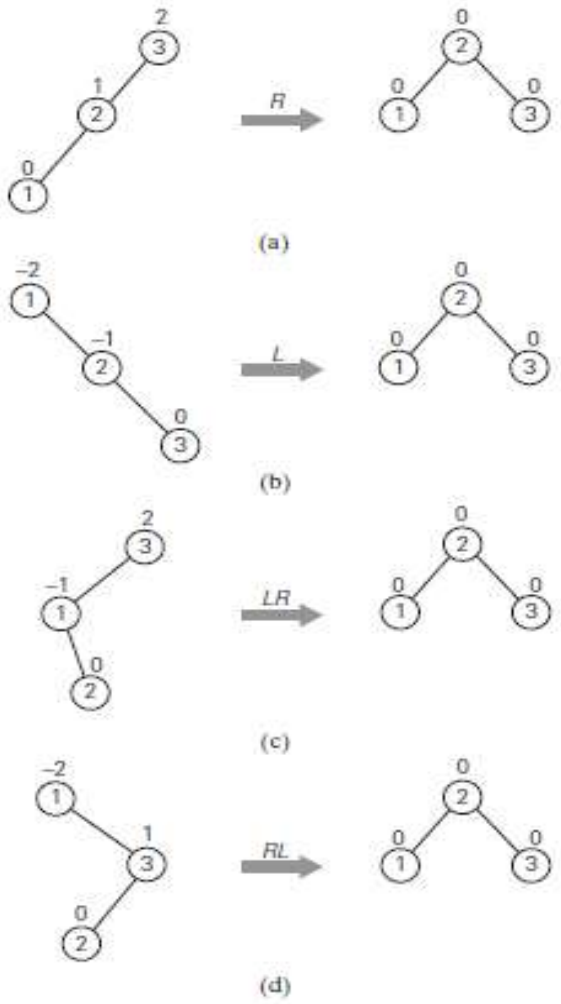
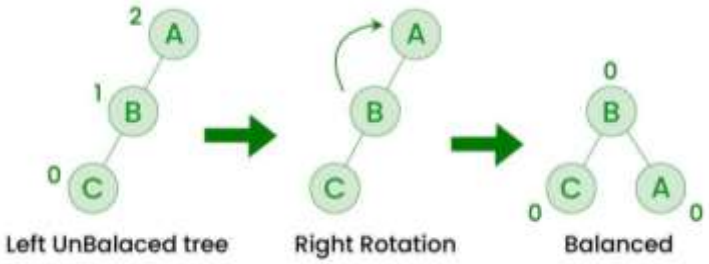


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

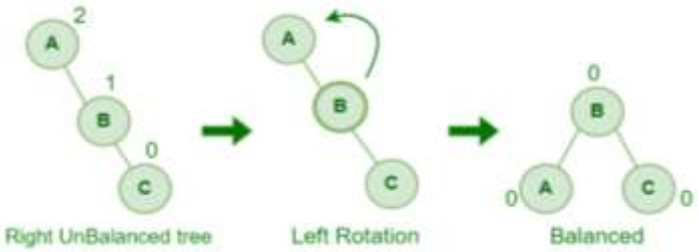
R-rotation (single right rotation)

Rotation is performed on the node with balance factor out or range (valid range: -1, 0, 1) by rotating the edge connecting the root and its left child in the binary tree in Figure 6.3a to the right.)

This rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

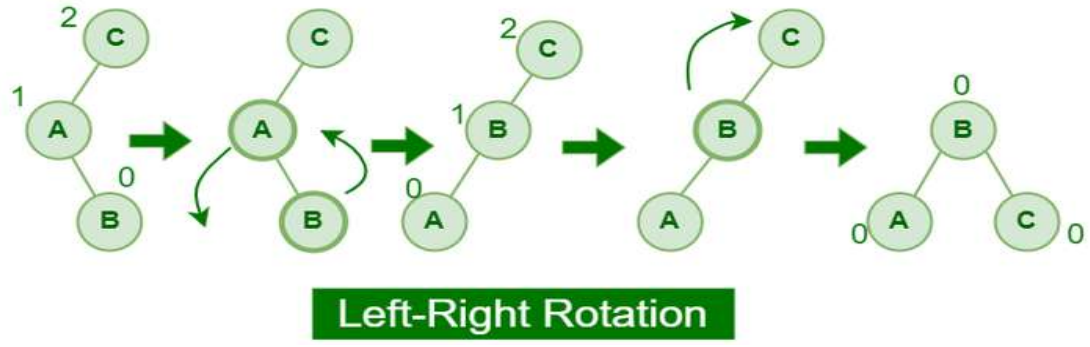


L-rotation(single left rotation)
is the mirror image of the single R-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of -1 before the insertion.

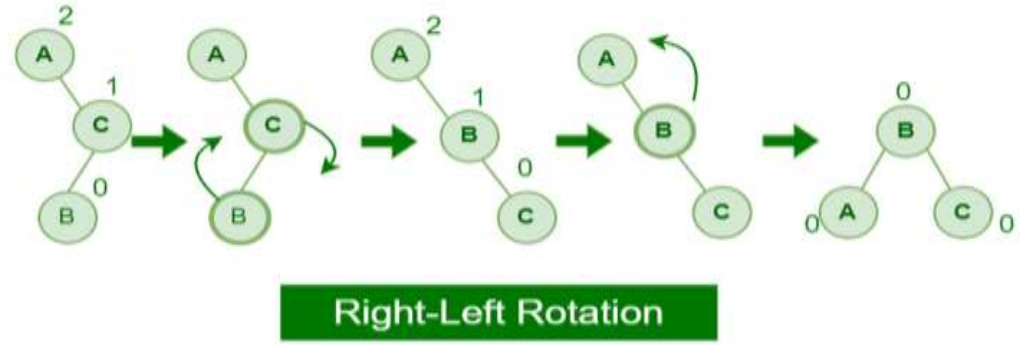


LR rotation (double left-right rotation)
It is a combination of two rotations: we perform the L-rotation of the left subtree of root r followed by the R-rotation of the new tree rooted at r.

It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



RL-rotation (double right-left rotation)
A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Construction of AVL Tree

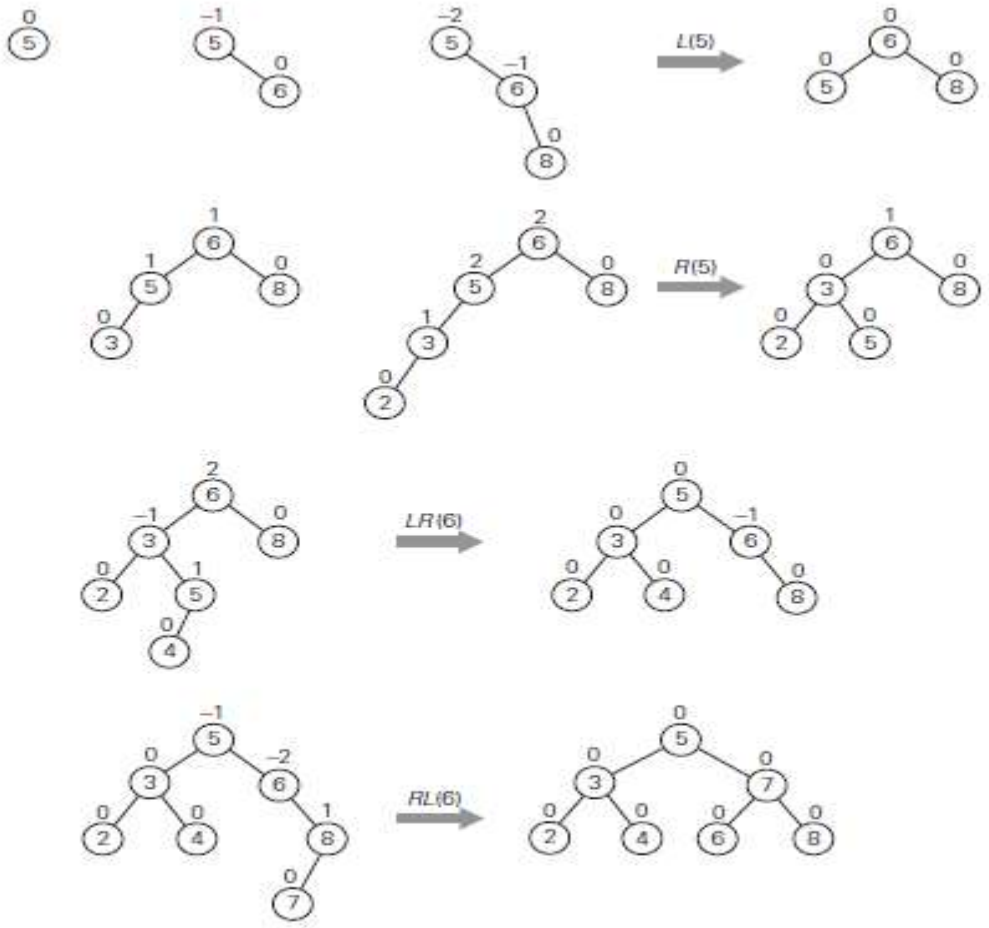
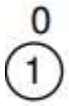


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

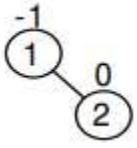
Example 2: Construct AVL Tree for 1, 2, 3, 4, 5, 6, 7

Step 1

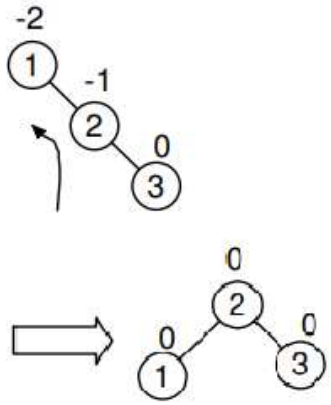


No rebalancing required

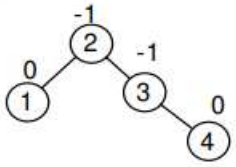
Step 2



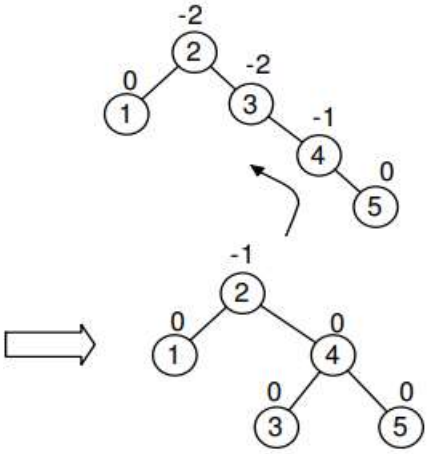
Step 3



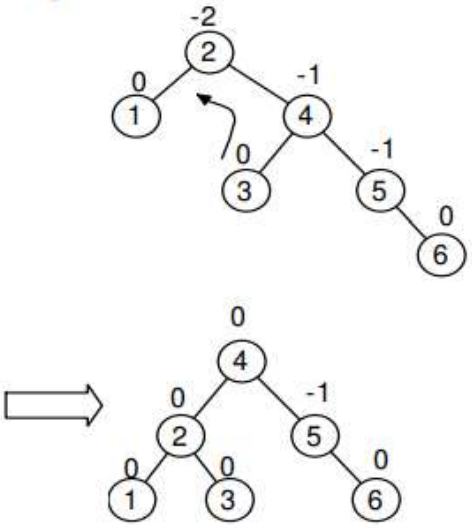
Step 4

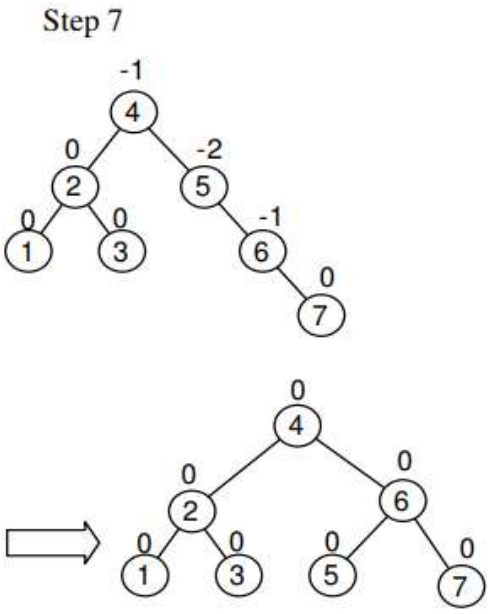


Step 5



Step 6





Efficiency:
Operations of searching, insertion and deletion of node are *θ(log n)* in the worst case.

The drawbacks of AVL trees are frequent rotations and the need to maintain balances for its nodes.

2-3 Trees

2-3 Trees is introduced by the U.S. computer scientist John Hopcroft in 1970.

A **2-3 tree** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

A **2-node** contains a single key K and has two children: the left child serves as the root of a subtree whose keys are less than K, and the right child serves as the root of a subtree whose keys are greater than K. (same as Binary Search Tree)

A **3-node** contains two ordered keys K1 and K2 (K1<K2) and has three children. The leftmost child serves as the root of a subtree with keys less than K1, the middle child serves as the root of a subtree with keys between K1 and K2, and the rightmost child serves as the root of a subtree with keys greater than K2.

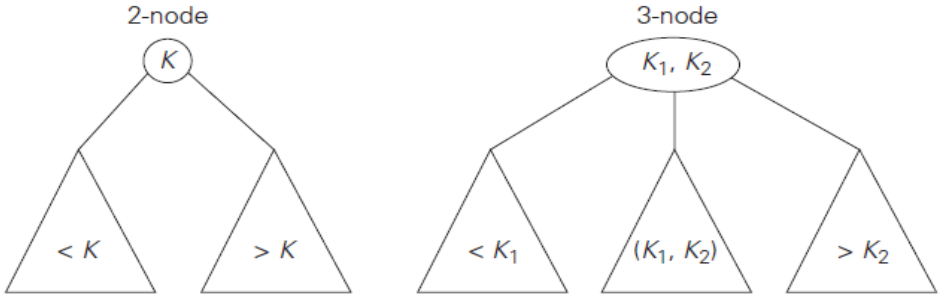


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

The 2-3 tree is that all its leaves must be on the same level. In other words, a 2-3 tree is always perfectly height-balanced: the length of a path from the root to a leaf is the same for every leaf.

Searching for a given key K in a 2-3 tree:

- If the root is a 2-node, we act as if it were a binary search tree: we either stop if K is equal to the root's key or continue the search in the left or right subtree if K is, respectively, smaller or larger than the root's key.
- If the root is a 3-node, we know after no more than two key comparisons whether the search can be stopped (if K is equal to one of the root's keys) or in which of the root's three subtrees it needs to be continued.

Inserting a new key in a 2-3 tree:
First of all, we always insert a new key K in a leaf, except for the empty tree. The appropriate leaf is found by performing a search for K.

If the leaf in question is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger than the node's old key. (Same as Binary Search Tree)

If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key.)

Example: Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

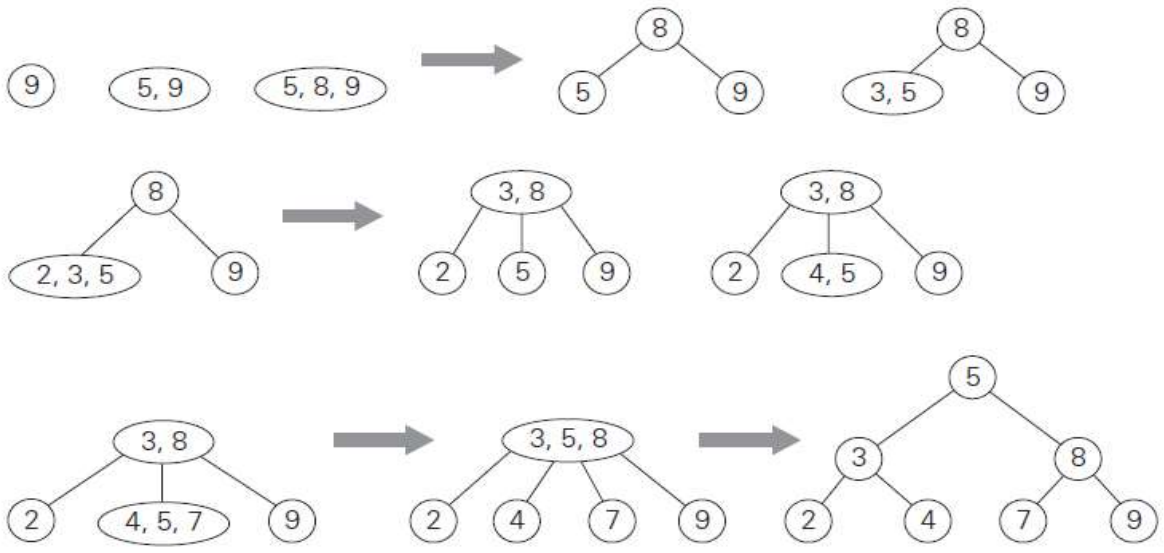


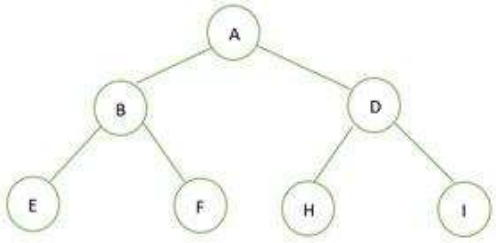
FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

Efficiency

The efficiency of 2-3 tree depends on height of the tree.

Upper bound for tree height:

A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes.



No. of nodes

1 ---------- 2^0

2 ---------- 2^1

4 ---------- 2^2

... 2^h

Therefore, for any 2-3 tree of height h with n nodes, we get the inequality

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$

and hence

$$h \leq \log_2(n+1) - 1.$$

Formula

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Lower bound for tree height:

A 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with n nodes,

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$$

and hence

$$h \geq \log_3(n+1) - 1.$$

These lower and upper bounds on height h,

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1,$$

imply that the time efficiencies of searching, insertion, and deletion are all in ***θ(log n)*** in both the worst and average case.

Heaps and Heapsort

DEFINITION A **heap** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

- The **shape property**—the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
- The **parental dominance** or heap property—the key in each node is greater than or equal to the keys in its children.

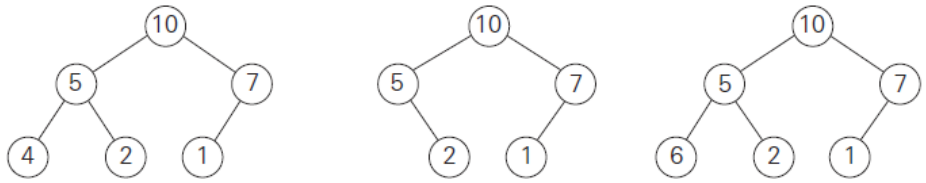


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

The first tree is a heap. The second one is not a heap, because the tree's shape property is violated. And the third one is not a heap, because the parental dominance fails for the node with key 5.

Note

- The key values in a heap are ordered top down; i.e., a sequence of values on any path from the root to a leaf is decreasing.
- There is no left-to-right order in key values; i.e., there is no relationship among key values for nodes either on the same level of the tree or, more generally, in the left and right subtrees of the same node.

List of important properties of heaps.

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
    a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lfloor n/2 \rfloor$ positions;

b. the children of a key in the array's parental position i ($1 \le i \le \lfloor n/2 \rfloor$) will be in positions 2i and 2i + 1, and, correspondingly, the parent of a key in position i ($2 \le i \le n$) will be in position $\lfloor i/2 \rfloor$.

A heap implemented as an array H[1..n] in which every element in position i in the first half of the array is greater than or equal to the elements in positions 2i and 2i + 1, i.e.,

$H[i] \ge \max\{H[2i], H[2i + 1]\}$ for $i = 1, \ldots, \lfloor n/2 \rfloor$

Heap construction (for a given list of keys)

Bottom-up Heap construction

It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree.

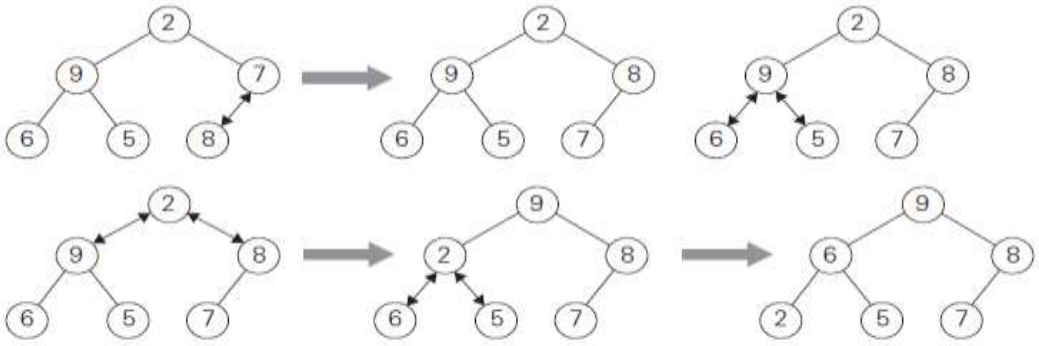


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

Working of algorithm

- The algorithm checks whether parental dominance holds for key of all the nodes (starting from last parent node to root node).
- If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position.
- This process continues until the parental dominance for K is satisfied.
- The algorithm stops after this is done for the root of the tree.

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

Analysis

Worst Case:

Let n = 2^k - 1 so that a heap's tree is full, i.e., the largest possible number of nodes occurs on each level. Let h be the height of the tree.

According to property 1, h = $\lfloor \log_2 n \rfloor$ or just $\lfloor log_2(n + 1) \rfloor$= k − 1 for the specific values of n we are considering.

Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm.

Since moving a node to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required.

The total number of key comparisons involving a key on level i will be 2(h − i).

Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^{h} i2^i$ or by mathematical induction on h.

Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than **2n** comparisons.

Top Down Heap construction

How can we insert a new key *K* into a heap.

a. First, attach a new node with key *K* in it after the last leaf of the existing heap.
b. Then shift *K* up to its appropriate place in the new heap as follows:
   - Compare *K* with its parent's key: if the latter is greater than or equal to *K,* stop (the structure is a heap)
   - otherwise, swap these two keys and compare *K* with its new parent.
   - This swapping continues until *K* is not greater than its last parent or it reaches the root.

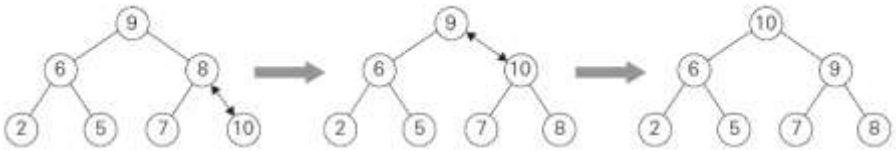


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Analysis:

The insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2 n$, the time efficiency of insertion is in $O(\log n)$.

Maximum Key Deletion (root element)

Deleting the root's key from a heap can be done with the following algorithm,

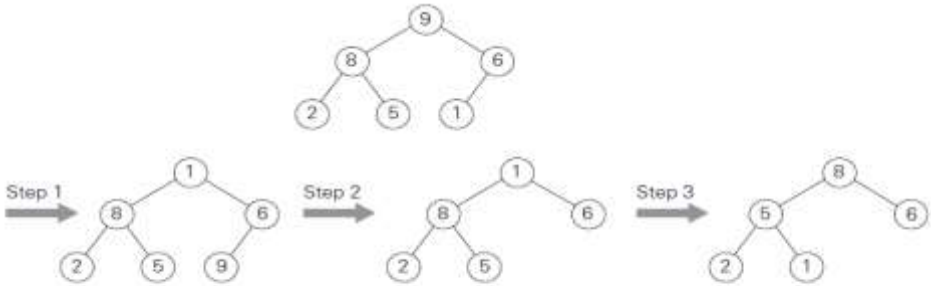


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key *K* of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting *K* down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for *K*: if it holds, we are done; if not, swap *K* with the larger of its children and repeat this operation until the parental dominance condition holds for *K* in its new position.

The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1.

Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in $O(\log n)$ as well.

Heap Sort

heapsort—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order. Heapsort is traced on a specific input in Figure 6.14.

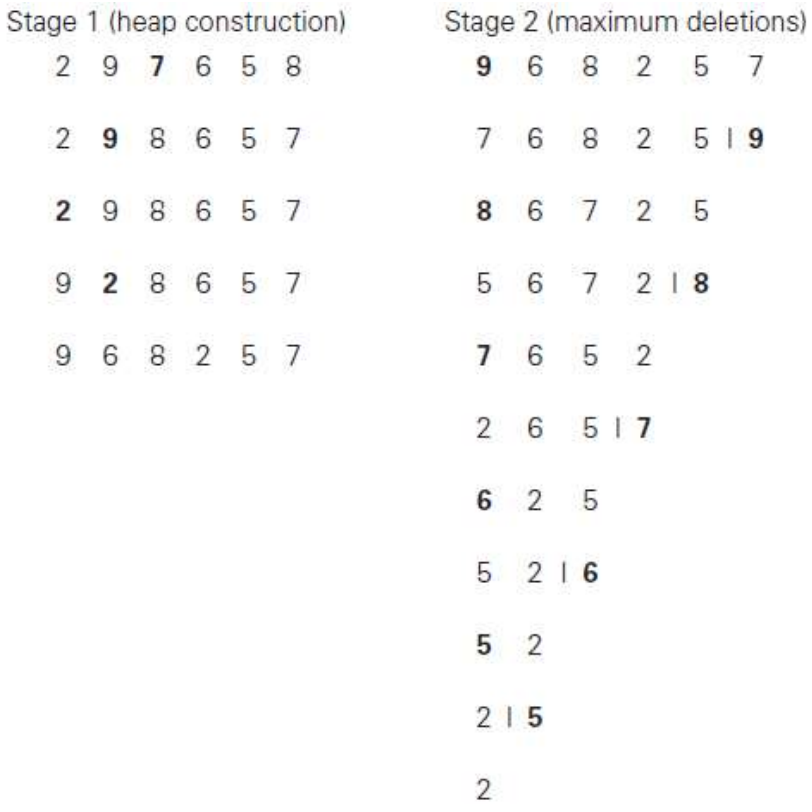


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

Analysis

Stage 1 (heap construction): the algorithm is in *O(n).*

Stage 2 (maximum deletions): The number of key comparisons, *C(n),* needed for eliminating the root keys from the heaps of diminishing sizes from *n* to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n \log_2 n.$$

This means that $C(n) \in O(n \log n)$ for the second stage of heapsort.

For both stages, we get $O(n) + O(n \log n) = O(n \log n)$.

In Heapsort elements are sorted in-place, i.e., it does not require any extra storage.

SPACE-TIME TRADEOFFS

To reduce the total time taken by an algorithm, it is good to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. This approach is called ***input enhancement***.

We will discuss counting methods for sorting and Horspool algorithm based on input enhancement.

Prestructuring technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data.
Example: hashing and indexing with B-trees

Dynamic programming This strategy is based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in question is then obtained.

The two resources—time and space can align to bring an algorithmic solution that minimizes both the running time and the space consumed.
Example:
Time efficiency of the two principal traversal algorithms—depth-first search and breadth-first search—depends on the data structure used for representing graphs:

- $\Theta(n^2)$ for the adjacency matrix representation and
- $\Theta(n + m)$ for the adjacency list representation,

where n and m are the numbers of vertices and edges, respectively.

Sorting by Counting

Applying the input-enhancement technique, to the sorting problem.

Comparison counting sort

The idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table. These numbers will indicate the positions of the elements in the sorted.

These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array.

This algorithm is called ***comparison counting sort*** (Figure 7.1).

Array $A[0..5]$		62	31	84	96	19	47
Initially	*Count* []	0	0	0	0	0	0
After pass $i = 0$	*Count* []	3	0	1	1	0	0
After pass $i = 1$	*Count* []		1	2	2	0	1
After pass $i = 2$	*Count* []			4	3	0	1
After pass $i = 3$	*Count* []				5	0	1
After pass $i = 4$	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

FIGURE 7.1 Example of sorting by comparison counting.

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

Analysis

Input size: n

Basic operation: $A[i]<A[j]$

Basic operation Count:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Time efficiency: $\Theta(n^2)$.

Drawback:

The counting idea does work productively in a situation in which elements to be sorted belong to a known small set of values.

Input Enhancement in String Matching

The problem of string matching requires finding an occurrence of a given string of *m* characters called the ***pattern*** in a longer string of *n* characters called the ***text***.

The worst-case efficiency of the brute-force algorithm for string matching is in the *O(nm)* class.

Most of faster algorithms exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text.

Horspool's Algorithm

Example, searching for the pattern BARBER in some text:

$$\begin{array}{ccccccccc} s_0 & \ldots & & & c & \ldots & s_{n-1} \\ & & \text{B} & \text{A} & \text{R} & \text{B} & \text{E} & \text{R} & \end{array}$$

Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text.

a. If all the pattern's characters match successfully, a matching substring is found.
b. If a mismatch occurs, we need to shift the pattern to the right using shift table.

Precompute shift sizes for all characters that can be found in the input text and store them in a table
(including, for natural language texts, the space, punctuation symbols, and other special characters).

The table's entries will indicate the shift sizes computed by the formula,

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

Creating Shift Table

1. Initialize all the entries to the pattern's length *m* and
2. scan the pattern left to right repeating the following step *m* − 1 times:
    a. for the *j* th character of the pattern *(*0 ≤ *j* ≤ *m* − 2*)*,
        i. overwrite its entry in the table with *m* − 1− *j* , which is the character's distance to the last character of the pattern.

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

ALGORITHM *ShiftTable*$(P[0..m-1])$

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
//          filled with shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

Horspool's algorithm

Step 1 For a given pattern of length *m* and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern

a. reaches beyond the last character of the text.

b. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all *m* characters are matched (then stop) or a mismatching pair is encountered.

c. In case of mismatch, retrieve the entry *t (c)* from the *c*'s column of the shift table where *c* is the text's character currently aligned against the last character of the pattern, and shift the pattern by *t (c)* characters to the right along the text.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])      //generate Table of shifts
i ← m − 1                    //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                    //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R               B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                   B A R B E R
```

■

Efficiency

The worst-case efficiency of Horspool's algorithm is in *O(nm).*
For random texts, it is in Θ*(n)*.

References:

1) Introduction to the Design and Analysis of Algorithms, By Anany Levitin, 3rd Edition (Indian), 2017, Pearson.
2) https://www.geeksforgeeks.org/introduction-to-avl-tree/

*****