DESIGN AND ANALYSIS OF ALGORITHMS

By : Akshith Shetty

UNIT-I	
	15 Hours
INTRODUCTION: What is an Algorithm? Fundamentals of Algorithmic Problem Solving **(Text Book-1: Chapter 1: 1.1 to 1.2)** **FUNDAMENTALS OF THE ALGORITHMS EFFICIENCY:** Analysis Framework, Asymptotic Notations and Basic efficiency classes, Mathematical Analysis of Non-recursive and Recursive Algorithms, **(Text Book-1: Chapter 2: 2.1 to 2.4)** **BRUTE FORCE:** Background, Selection Sort and Bubble sort, Sequential search and Brute-Force String Matching algorithms with complexity analysis, Exhaustive search **(Text Book-1: Chapter 3: 3.1, 3.2,3.4)** **DIVIDE AND CONQUER:** General Method, Merge sort, Quick sort, Binary Search algorithms with Complexity analysis **(Text Book-1: Chapter 4: 4.1 to 4.3)**	

Introduction

Algorithm : Algorithm is a sequence of unambiguous instructions for solving a problem for obtaining the desired output for any legitimate i/p in a finite amount of time.

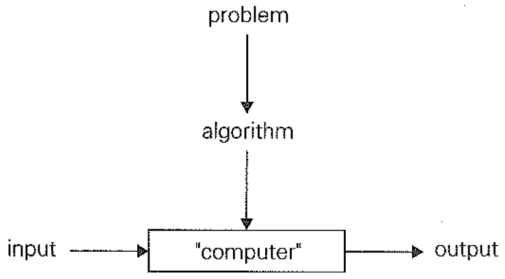


FIGURE 1.1 Notion of algorithm

Properties:

- No ambiguity in the instructions
- Range of input must be specified carefully
- Same algorithm can be expressed in different ways
- Several algorithm can might exist for solving same problem
- Algorithms can be based on different ideas and can solve a problem with different speeds.

Algorithms for calculating gcd of 2 numbers

Euclid's algorithm

- **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.

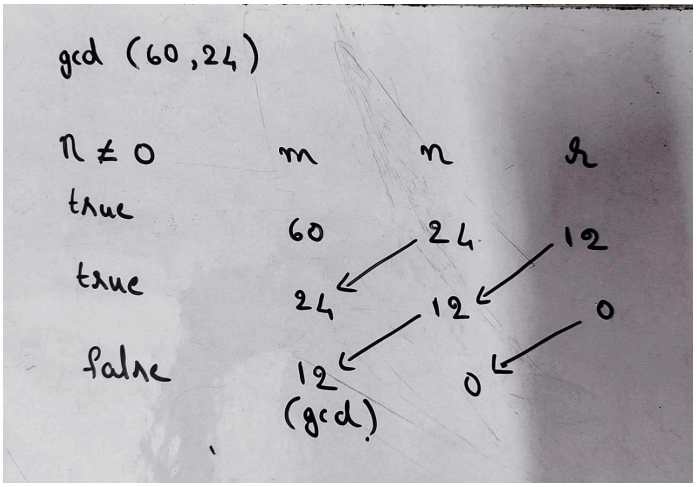
Pseudo-code:

```
//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 ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
    return m
```

or

```
    while(true) do
        //Step 1
        if n == 0
            return m
        //Step 2
        r ← m mod n
        //Step 3
        m ← n
        n ← r
```

Ex:



Consecutive integer checking algorithm

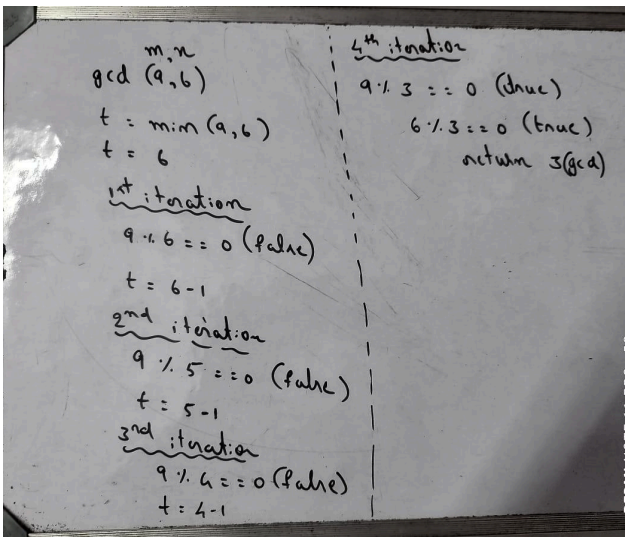
- **Step 1:** Assign the value of min{m, n} to t.
- **Step 2:** Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
- **Step 3:** Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
- **Step 4:** Decrease the value of t by 1. Go to Step 2.

Pseudo-code

```
t = min(m, n)  // Step 1

while (true) do
    if m mod t = 0 then  // Step 2
        if n mod t = 0 then  // Step 3
            return t  // t is the GCD, stop and return
    t = t - 1  // Step 4
```

Ex:



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 Pn times in m and n, respectively, it should be repeated min{Pm, Pn} times.)
- **Step 4:** Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Note : Middle-school procedure does not qualify, in the form presented, as a legitimate algorithm.This is because they require a list of prime numbers

Sieve of Eratosthenes(Algorithm for generating consecutive primes)

15.1 Sieve of Eratosthenes | Challenge | C++ Placement Course

ALGORITHM *Sieve*(n)

```
//Implements the sieve of Eratosthenes
//Input: An integer n ≥ 2
//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
```

Code snippet for finding floor square root of a given number

Program To Find Square Root/Floor Of Square Root Of A Number | FREE DSA Course…

Algorithm

1. **Base Case:** If N is 0, return 0; if N is 1, return 1.
2. **Initialization:** Set the lower bound to 0 and the upper bound to N/2.
3. **Binary Search Loop:** While the lower bound is less than or equal to the upper bound, perform the following steps else go to step 6.
4. **Mid-point Calculation:** Calculate the midpoint value.
    a. If the square of the mid-point is equal to N, return the midpoint value.
    b. If not, proceed to the next steps.
5. **Adjusting Bounds:**
    a. If the square of the mid-point is less than N, update the lower bound to one greater than the midpoint and store the midpoint in the *result* variable.
    b. If the square of the mid-point is greater than N, update the upper bound to one less than the midpoint.
6. Return the value stored in the *result* variable as the floor value of the square root of N.

Code snippet

```
int sqrt(int n){
   if(n == 1 || n == 0)
      return n;

   int l = 2,h=n/2,res,m;

   while(l<=h){
      m = (l+h)/2;
```

```
        if(m*m == n)
            return m;
        else if(m*m<n){
            res = m;
            l = m+1;
        }
        else
            h = m-1;
    }
    return res;
}
```

Fundamentals of Algorithmic Problem Solving

Sequence of steps in the process of design and analysis of algorithms

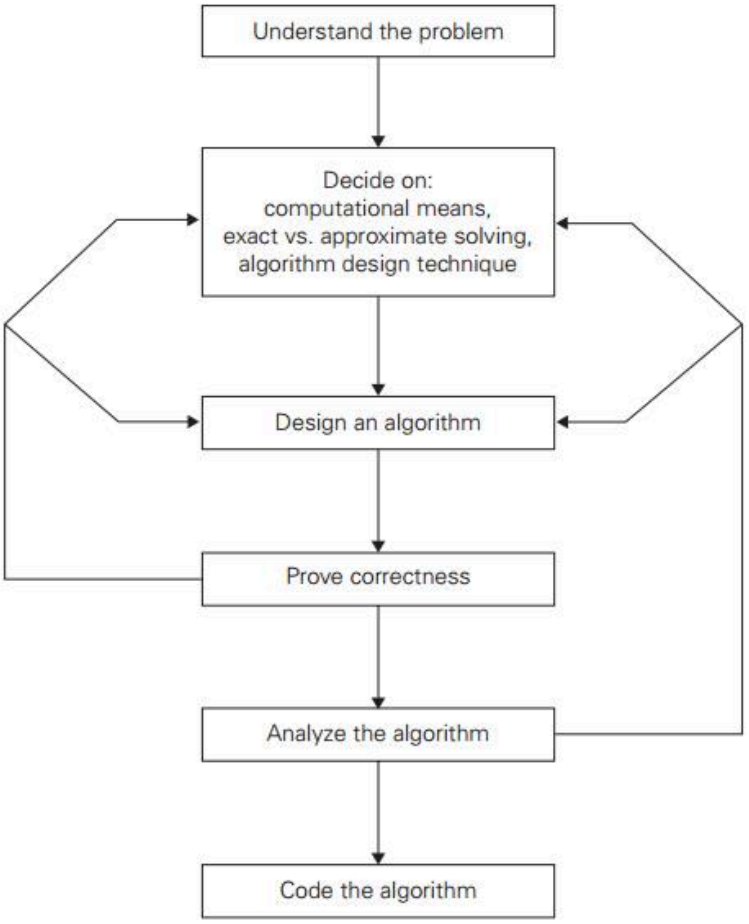


FIGURE 1.2 Algorithm design and analysis process.

Understanding the problem

- Before designing an algorithm, it is crucial to understand the problem by carefully reading its description and clarifying doubts through questions.

Decide on: Approximate vs exact problem solving

- The decision to use exact or approximate algorithms depends on factors like the complexity of the problem, the computational resources available.

Design an algorithm

- Algorithm design techniques provide general approaches for solving problems which are applicable across various computing domains.

Prove correctness

- Proving an algorithm's correctness involves demonstrating that the algorithm produces the expected result for all valid inputs

Analyse the algorithm

- Analyse the algorithm for desirable characteristics like efficiency (both in terms of time and space) simplicity, and generality.

Code the algorithm

- The process of coding an algorithm requires careful consideration to avoid incorrect or inefficient implementations, with testing being a crucial aspect.

Fundamentals of The Algorithms Efficiency:

Analysis Framework

- **Time Efficiency:**
  - Definition: Time efficiency refers to how quickly an algorithm can solve a problem or complete a task.
- **Space Efficiency:**
  - Definition: Space efficiency, refers to how much extra memory or storage space an algorithm requires to execute.

Units for Measuring Running time

- The basic operation is often the most time-consuming operation and present in the algorithm's innermost loop.
- Define the execution time of the basic operation as 'Cop' on a specific computer and the count as 'C(n)'.
- The running time 'T(n)' can be estimated using the formula:

$$\mathbf{T(n) \approx CopC(n).}$$

Orders of Growth

The order of growth of an algorithm refers to the rate at which its running time (or the number of operations) increases as a function of the input size.

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 function. It grows so slowly, in fact, that we should expect a program implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes.

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

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 of 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
```

"**Worst case efficiency**" refers to the measure of how an algorithm performs with the input that leads to the maximum possible running time.

In case of linear search the worst case is 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:

Cworst(n) = n

Worst case efficiency guarantees that for any instance of size n, the running time will not exceed Cworst(n),

"**Best case efficiency**" refers to the measure of how an algorithm performs with input that leads to the minimum possible running time.

For example, for sequential search, best-case inputs are lists of size n with their first elements equal to a search key; accordingly,

Cbest(n) = 1.

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either.For certain algorithms, their efficient handling of inputs close to the best-case scenario allows for practical advantages in real-world applications.

"**Average case efficiency**" refers to the measure of how an algorithm performs on an average or expected input, considering all possible inputs with equal probability.
Let us consider sequential search again. The standard assumptions are that

1. The probability of a successful search is equal to P ($0 \leq P \leq 1$)
2. The probability of the first match occurring in the ith position of the list is the same for every i.

In the case of a successful search, the probability of the first match occurring in the ith 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 is n with the probability of such a search being (1- p). Therefore,

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

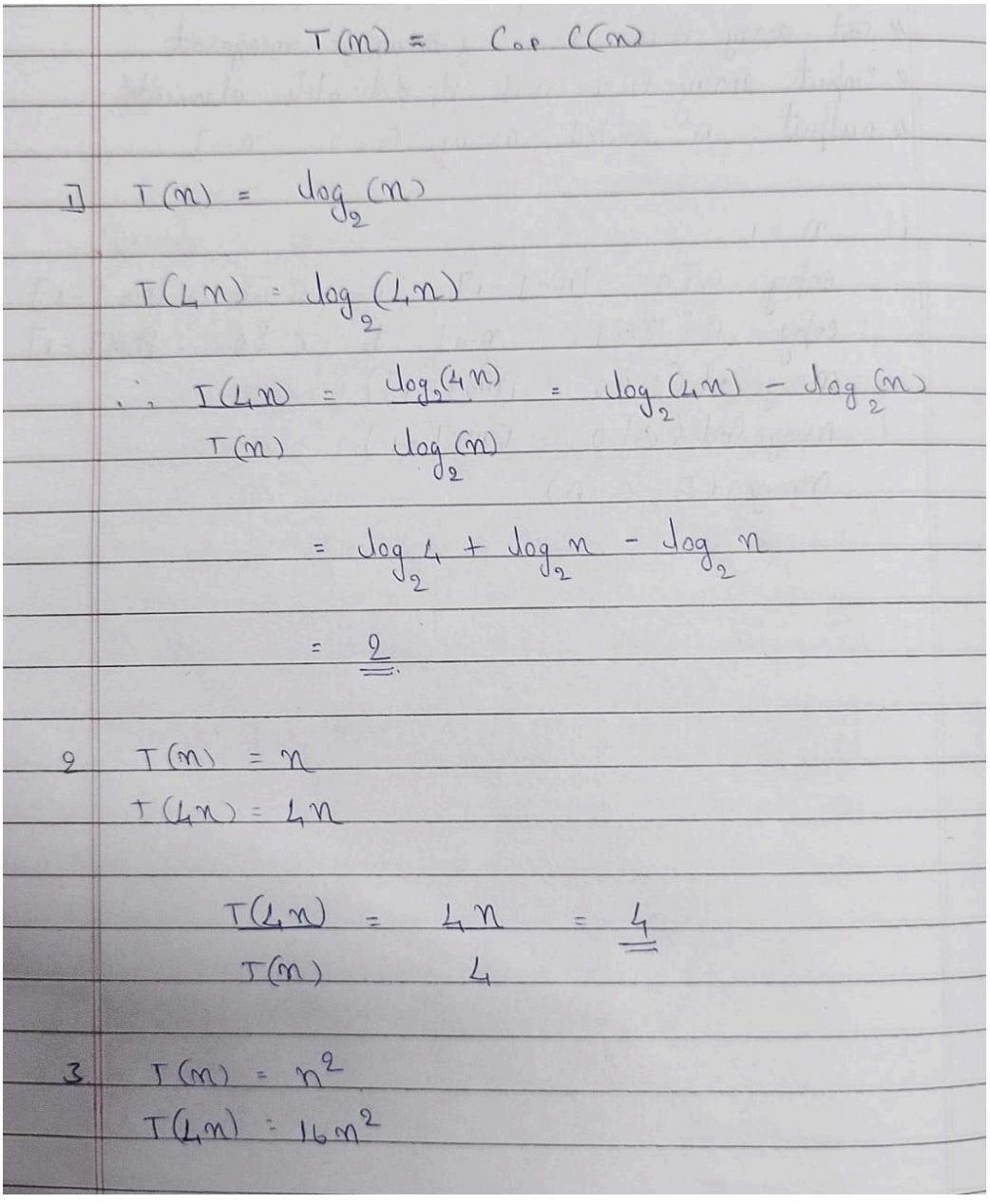
Summary of general framework for analysis of an algorithm

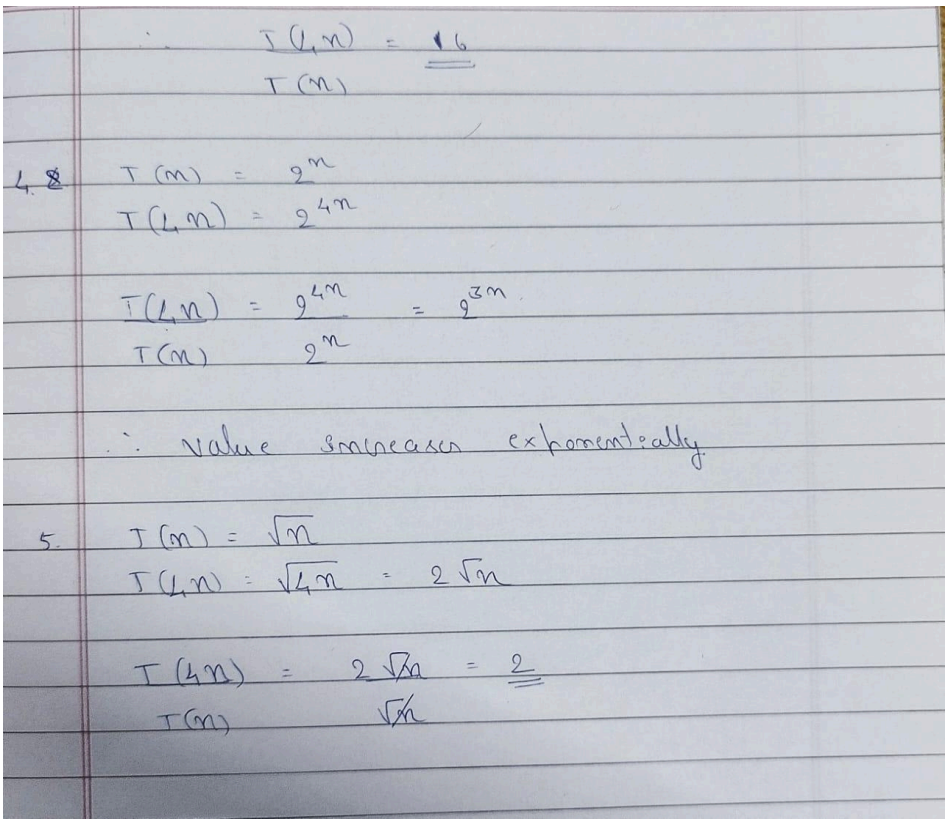
- Both time and space efficiencies are measured as functions of the algorithm's input size.
- **Time efficiency** is measured by counting the number of times the algorithm's basic operation is executed.
- **Space efficiency** is measured by counting the number of extra memory units consumed by the algorithm.

- 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.

For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

1) $log_2 n$
2) √n
3) n
4) n^2
5) n^3
6) 2^n





Asymptotic Notations and Basic Efficiency Classes

- In the following discussion, t(n) and g(n) can be any nonnegative functions defined on the set of natural numbers.
- In the context we are interested in, t(n) will be an algorithm's running time (usually indicated by its basic operation count C(n)),
- g(n) will be some simple function to compare the count with.

Informal Introduction

L-1.3: Asymptotic Notations | Big O | Big Omega | Theta Notations | Most Imp Topic Of …

Big-oh

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

Thus, to give a few examples, the following assertions are all true:

$$n \in O(n^2),\ 100n + 5 \in O(n^2),\ 1/2\ n(n-1) \in O(n^2)$$

$$n3 \notin O(n2),\ 0.00001n3 \notin O(n2),\ n4 + n + 1 \notin O(n2).$$

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 $n^4 + n + 1$.

Big-omega

The second notation, Ω(g(n)), stands for the set of all functions with a higher or same order of growth as g(n)(to within a constant multiple, as n goes to infinity).
For example,

$$n^3 \in \Omega(n2\ ),\ 1/2\ \Omega\ n(n-1) \in \Omega\ (n^2\ ),\ \text{but}\ 100n + 5 \notin \Omega(n^2\ ).$$

Big Theta

Finally,Θ (g(n)) is the set of all functions that have the same order of growth as g(n) (to within a constant multiple, as n goes to infinity). Thus, every quadratic function an^2 + bn + c with a > 0 is in Θ(n^2), but so are, among infinitely many others, n^2 + sin n and n^2 + log n. (Can you explain why?)

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** by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some non-negative integer n_0 such that

$$t\ (n) \leq cg(n)\ \text{for all}\ n \geq n_0.$$

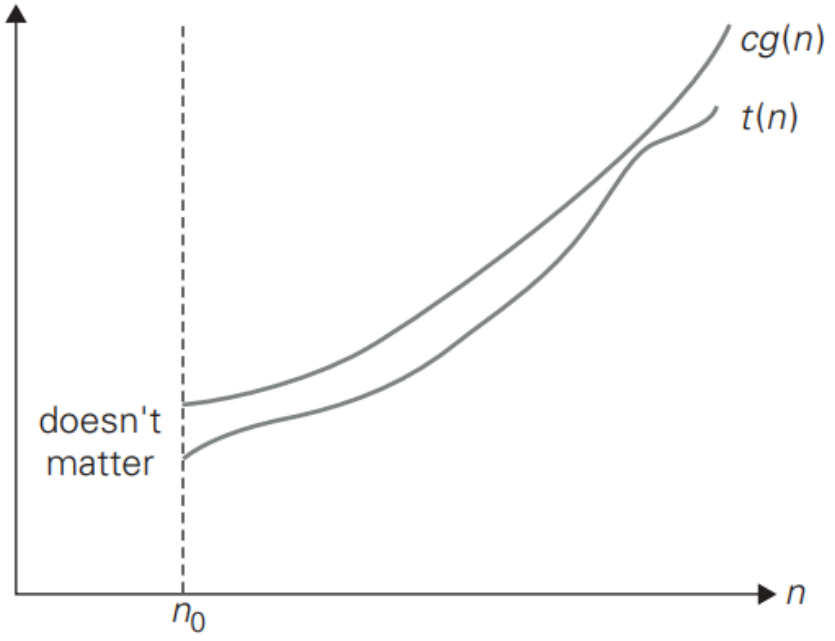


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

As an example, let us formally prove one of the assertions made in the introduction: 100n + 5 ∈ O(n^2).

Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5) = 101n \leq 101n^2 .$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively. Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also reason that

$$100n + 5 \leq 100n + 5n \text{ (for all } n \geq 1) = 105n$$

to complete the proof with c = 105 n_0=1

Ω - 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** by some positive 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

$$t(n) \geq cg(n) \text{ for all } n \geq n_0.$$

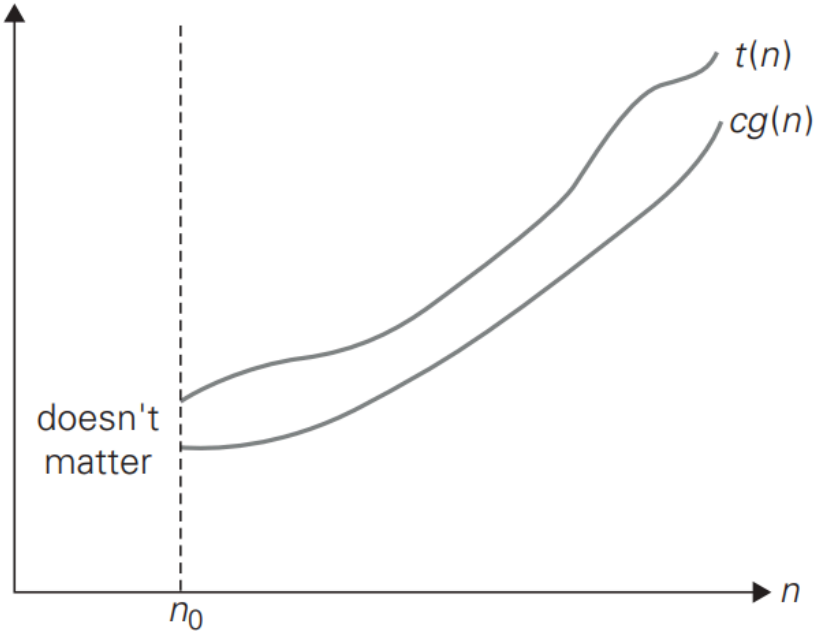


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Here is an example of the formal proof that

$$n^3 \in (n^2): n^3 \geq n^2 \text{ for all } n \geq 0$$

i.e., we can select c = 1 and n_0= 0

θ - notation

DEFINITION : 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 below** by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n0 such that

$$c2g(n) \leq t(n) \leq c1g(n) \text{ for all } n \geq n0$$

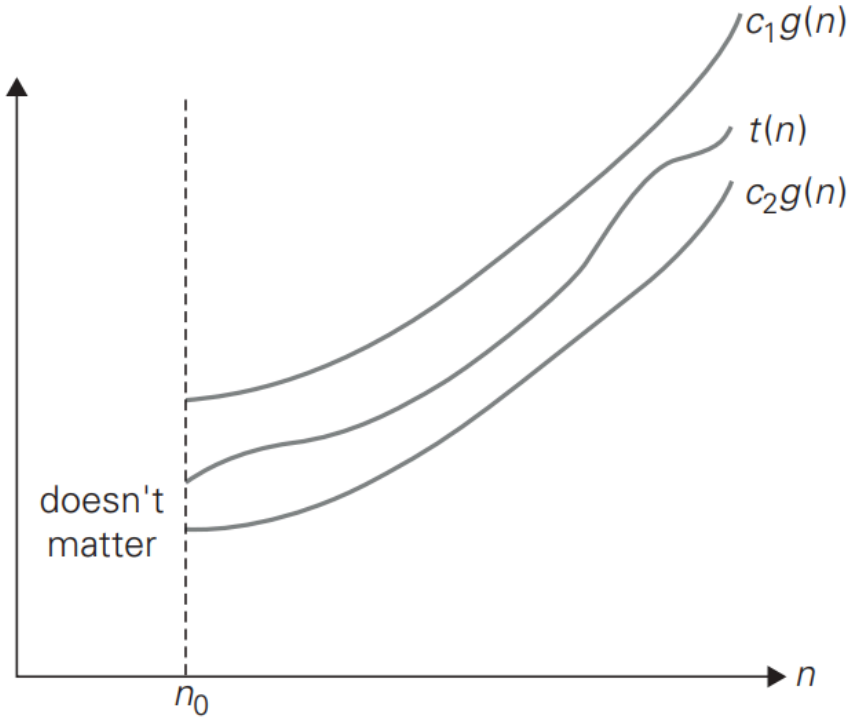


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

For example, let us prove that 1/2n(n − 1) ∈ θ(n^2).
First, we prove the right inequality (the upper bound):

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

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

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

Hence, we can select c2 = 1/4 , c1 = 1/2 , and n0 = 2.

Useful property involving the Asymptotic Notations

The following property, in particular, is useful in analysing algorithms that comprise two consecutively executed

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 non-negative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

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. ■

This implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part.

Using limits for comparing order of growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

L'Hopital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

Derivatives of Logs

Derivative of Common Logarithm:

$$\frac{d}{dx}\log_a(x) = \frac{1}{x \ln(a)}$$

Derivative of Natural Logarithm:

$$\frac{d}{dx}\ln(x) = \frac{1}{x}$$

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.) ■

Big - o vs little - o : [Little o vs Big O Notation - YouTube]

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n!\in\Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

Important Summation Formulas

1. $\sum_{i=l}^{u}1=\underbrace{1+1+\cdots+1}_{u-l+1\text{ times}}=u-l+1$ (l, u are integer limits, $l\le u$); $\sum_{i=1}^{n}1=n$
2. $\sum_{i=1}^{n}i=1+2+\cdots+n=\frac{n(n+1)}{2}\approx\frac{1}{2}n^2$
3. $\sum_{i=1}^{n}i^2=1^2+2^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}\approx\frac{1}{3}n^3$
4. $\sum_{i=1}^{n}i^k=1^k+2^k+\cdots+n^k\approx\frac{1}{k+1}n^{k+1}$
5. $\sum_{i=0}^{n}a^i=1+a+\cdots+a^n=\frac{a^{n+1}-1}{a-1}$ $(a\ne1)$; $\sum_{i=0}^{n}2^i=2^{n+1}-1$
6. $\sum_{i=1}^{n}i2^i=1\cdot2+2\cdot2^2+\cdots+n2^n=(n-1)2^{n+1}+2$
7. $\sum_{i=1}^{n}\frac{1}{i}=1+\frac{1}{2}+\cdots+\frac{1}{n}\approx\ln n+\gamma$, where $\gamma\approx0.5772\ldots$ (Euler's constant)
8. $\sum_{i=1}^{n}\lg i\approx n\lg n$

Sum Manipulation Rules

1. $\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$
4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Mathematical Analysis of Non-recursive and Recursive Algorithms

General Plan for Analysing 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.

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 a 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

we should consider the **comparison** to be the algorithm's basic operation. Note that the number of comparisons will be the same for all arrays of size n; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.

Let us denote C(n) the number of times this comparison is executed. 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. Therefore, we get the following sum for C(n):

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

This is an easy sum to compute because it is nothing other than 1 repeated n − 1 times. Thus,

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

Before proceeding with further examples, you may want to review Appendix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

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

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

and two summation formulas

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

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

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
```

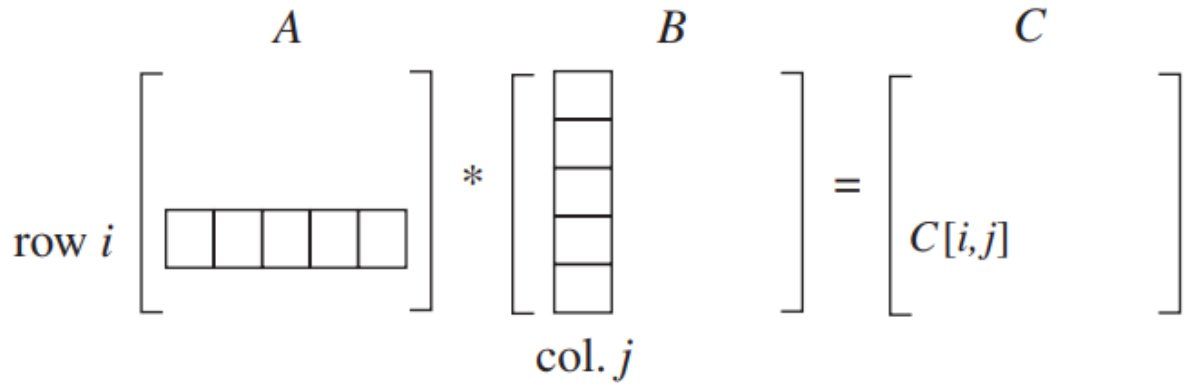
Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.In this example we will limit our investigation to the worst case only

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

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

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

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

There are two arithmetical operations in the innermost loop here—multiplication and addition.Actually, we do not have to choose between them, because on each repetition of the innermost loop each of the two is executed exactly once. Still, following a well-established tradition, we consider multiplication as the basic operation

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

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

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

$$= n^3$$

If we now want to estimate the running time of the algorithm on a particular machine, we can do it by the product

$$T(n) \approx C_m M(n) = C_m n^3,$$

where C_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx C_m M(n) + C_a A(n) = C_m n^3 + C_a n^3 = (C_m + C_a) n^3,$$

where C_a is the time of one addition.

EXAMPLE 4 :

ALGORITHM Sum(n)
//Input: A nonnegative integer n
S ← 0
for i ← 1 to n do
S ← S + i
return S

Solution:

$$C(n) = \sum_{i=1}^{n} 1$$

$$= n \in \theta(n)$$

Mathematical analysis of recursive algorithm

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

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \text{ for } n \geq 1$$

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

The basic operation of the algorithm is multiplication,5 whose number of executions we denote M(n).Since the function F (n) is computed according to the formula

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

the number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n \text{ for } n > 0}$$

Indeed, M(n − 1) multiplications are spent to compute F (n − 1), and one more multiplication is needed to multiply the result by
n.The above equation is a recurrence relation

Initial condition

if n = 0 return 1.

Therefore,

M(0) = 0.

M(n) = M(n − 1) + 1 substitute M(n − 1) = M(n − 2) + 1
= [M(n − 2) + 1] + 1 = M(n − 2) + 2 substitute M(n − 2) = M(n − 3) + 1
= [M(n − 3) + 1] + 2 = M(n − 3) + 3

Therefore general formula is

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

On substituting i = n, we get

$$M(n) = M(n - n) + n$$
$$M(n) = M(0) + n$$
$$M(n) = n \in \theta(n)$$

EXAMPLE 5 : recursive algorithms: the Tower of Hanoi puzzle

Code snippet:

```
Void ToH(int n, char src, char temp, char dest)
{
    if(n==1)
    {
        Printf("move disc %d from tower %c to %c\n", n, src,dest);
        Return;
    }
    /* Move n-1 discs from source to temp */
    ToH(n-1, src, dest, temp);
    Printf("move disc %d from tower %c to %c\n", n,src,dest);
    /*Move n-1 discs from temp to destination */
    ToH(n-1, temp, src, dest);
}
```

Clearly, the number of moves M(n) depends on n only, and we get the following recurrence equation for it

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

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \qquad \textbf{(2.3)}$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

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

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

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

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 (2.3):

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

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of n

EXAMPLE 6 : Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{n-1} + 2^{n-1}$.

Algorithm

POT(n)

if n == 0 return 1

else return POT(n-1) + POT(n-1)

Basic operation : sum

initial condition : S(0) = 0

$S(n) = S(n-1) + 1 + S(n-1)$

$S(n) = 2S(n-1) + 1$ --- ①

from eq. ①

$S(n-1) = 2S(n-2) + 1$ --- ②

$S(n-2) = 2S(n-3) + 1$ --- ③

sub ② in ①

$S(n) = 2[2S(n-2) + 1] + 1$

$= 2^2 S(n-2) + 2 + 1$

sub (3)

$S(n) = 2^2[2S(n-3) + 1] + 2 + 1$

$= 2^3 S(n-3) + 2^2 + 2 + 1$

in general,

$S(n) = 2^i S(n-i) + 2^{i-1} + 2^{i-2} + \dots 2 + 1$

sub $i = n$

$S(n) = 2^n S(0) + 2^{n-1} + 2^{n-2} \dots + 2 + 1$

$= 2^{n-1} + 2^{n-2} + \dots + 2 + 1$

$= 2^n - 1 \in \Theta(2^n)$

Brute force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection sort

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]

The basic operation is the key comparison A[j] < A[min].

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

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

Bubble sort

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]

The basic operation is the key comparison A[j+1] < A[j].

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

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

2 $x(n) = 3x(n-1)$, for all $n > 1$

$x(1) = 4$

from given eq we get,

$x(n-1) = 3x(n-2)$ ... ①

$x(n-2) = 3x(n-3)$ ... ②

∴ we have,

$$x(n) = 3x(n-1)$$
$$= 3\{3(x\,n-2)\}$$
$$= 3^2 x(n-2)$$

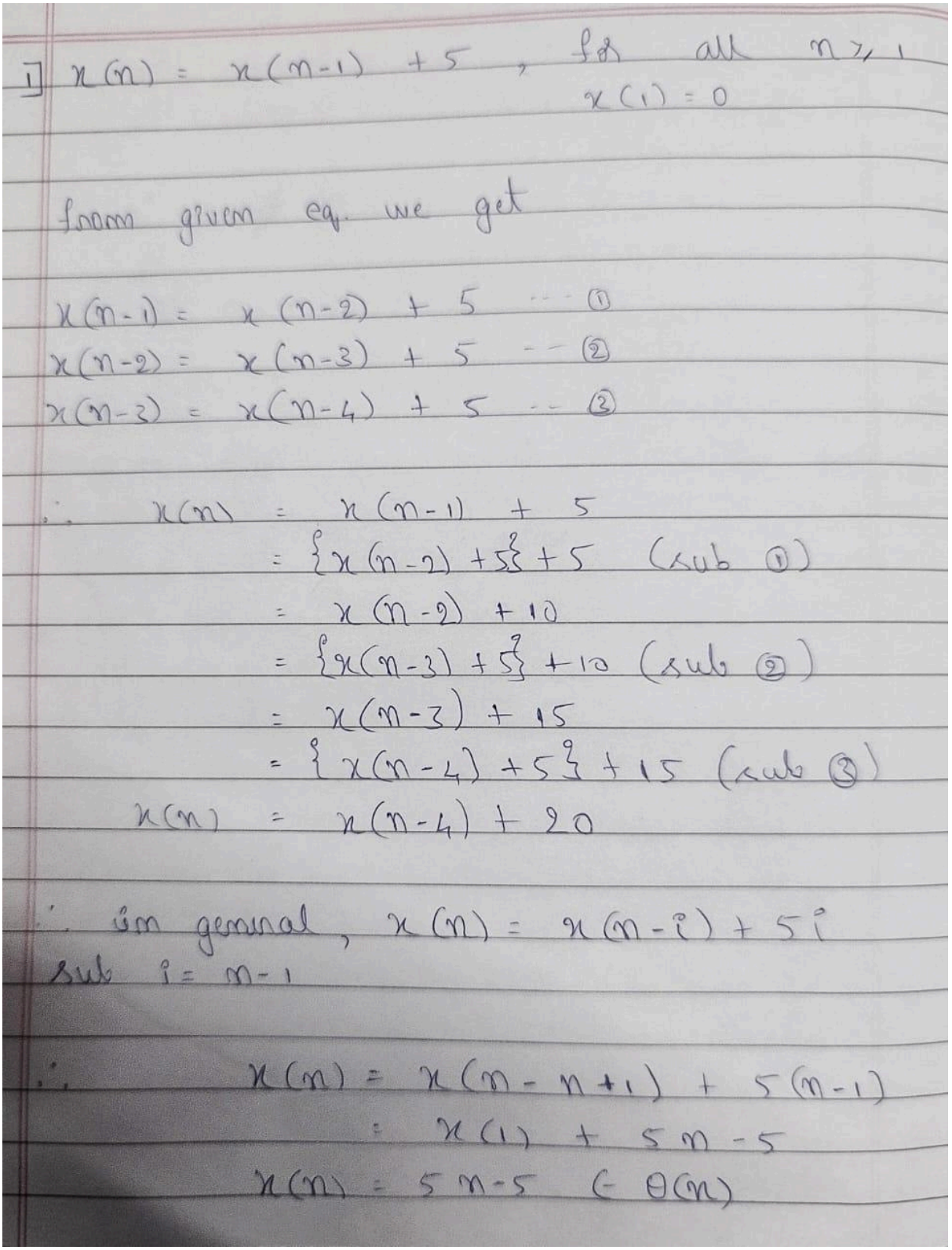
∴ in general,

$$x(n) = 3^i x(n-i)$$

sub $i = n-1$

$$x(n) = 3^{n-1} x(n-n+1)$$
$$= 3^{n-1} x(1)$$

$\therefore x(n) = 3^{n-1}(4) \in \Theta(3^n)$



Brute force string matching

Brute Force String Matching

```
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
```

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
                N O T
```

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

Number of comparisons in this example is : 12

in the worst case, the algorithm makes m(n − m + 1) character comparisons, which puts it in the O(nm) class

Exhaustive search

- Exhaustive Search is a brute-force algorithm that systematically enumerates all possible solutions to a problem and checks each one to see if it is a valid solution

Note : "enumerate" refers to the process of systematically listing or counting all possible elements or items in a set

For example, if you have a set of elements {A, B, C}, enumerating all possible combinations might involve generating sequences like {A, B, C}, {A, C, B}, {B, A, C}, {B, C, A}, {C, A, B}, {C, B, A}. The exhaustive search algorithm would go through each of these combinations to check if it satisfies the given problem constraints.

Divide and Conquer

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

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

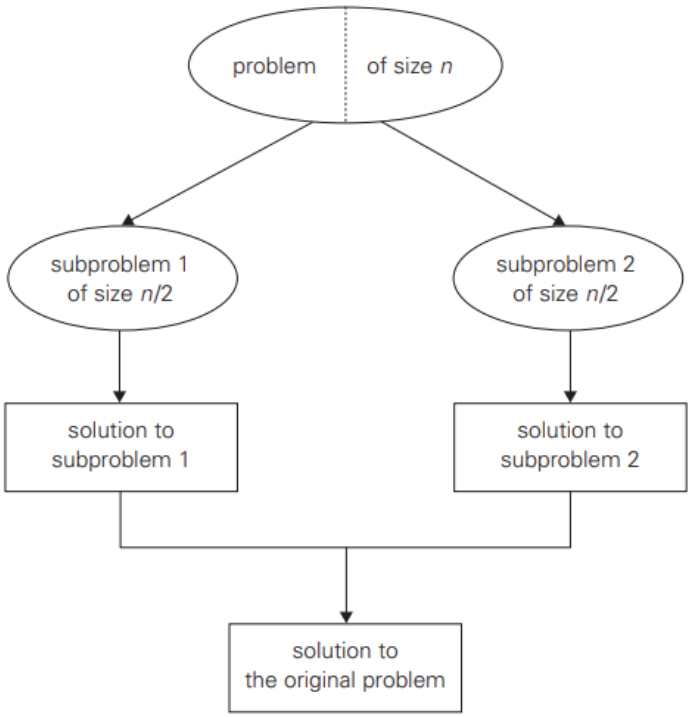


FIGURE 5.1 Divide-and-conquer technique (typical case).

Merge sort

7.7 Merge Sort in Data Structure | Sorting Algorithms| DSA Full Course

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 $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A) //see below

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]
```

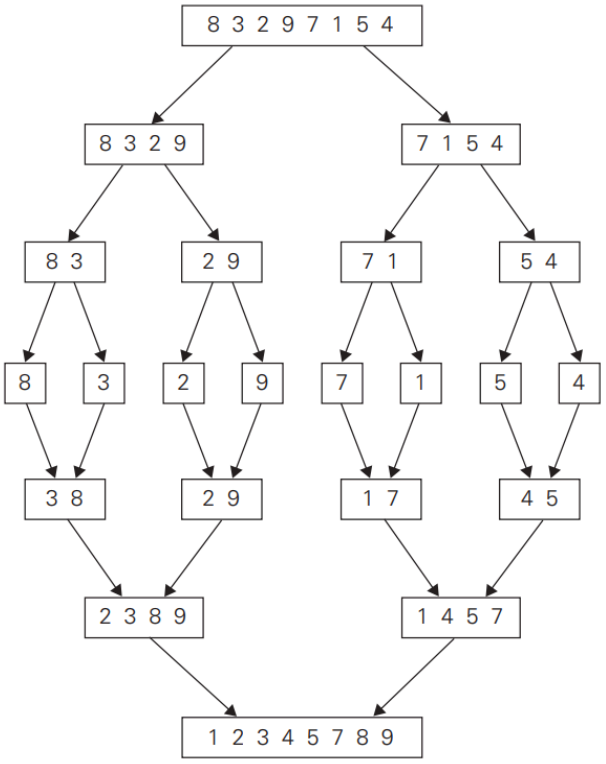


FIGURE 5.2 Example of mergesort operation.

Masters theorem

The Master Theorem provides a systematic way of solving recurrence relations of the form:

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

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.

The recurrence relation for the **number of key comparisons** C(n) is

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

Let us analyze Cmerge(n), the number of key comparisons performed during the merging stage

In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., *smaller elements may come from the alternating arrays*). Therefore, for the worst case, Cmerge(n) = n − 1, and we have the recurrence

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

$$Cworst(n) = n \log_2 n - n + 1 \text{ (From master theorem)}$$

Hence, according to the Master Theorem, **Cworst(n) ∈ θ(n log n)**

Quick sort

Logic : 7.6 Quick Sort in Data Structure | Sorting Algorithm | DSA Full Course

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])

ALGORITHM Partition(A[l..r])
//Partitions a subarray by 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

Best case efficiency

If all the splits happen in the middle of corresponding subarrays, we will have the best case. For partition algorithm 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

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

According to the Master Theorem, **Cbest(n) ∈ θ(n log n)**

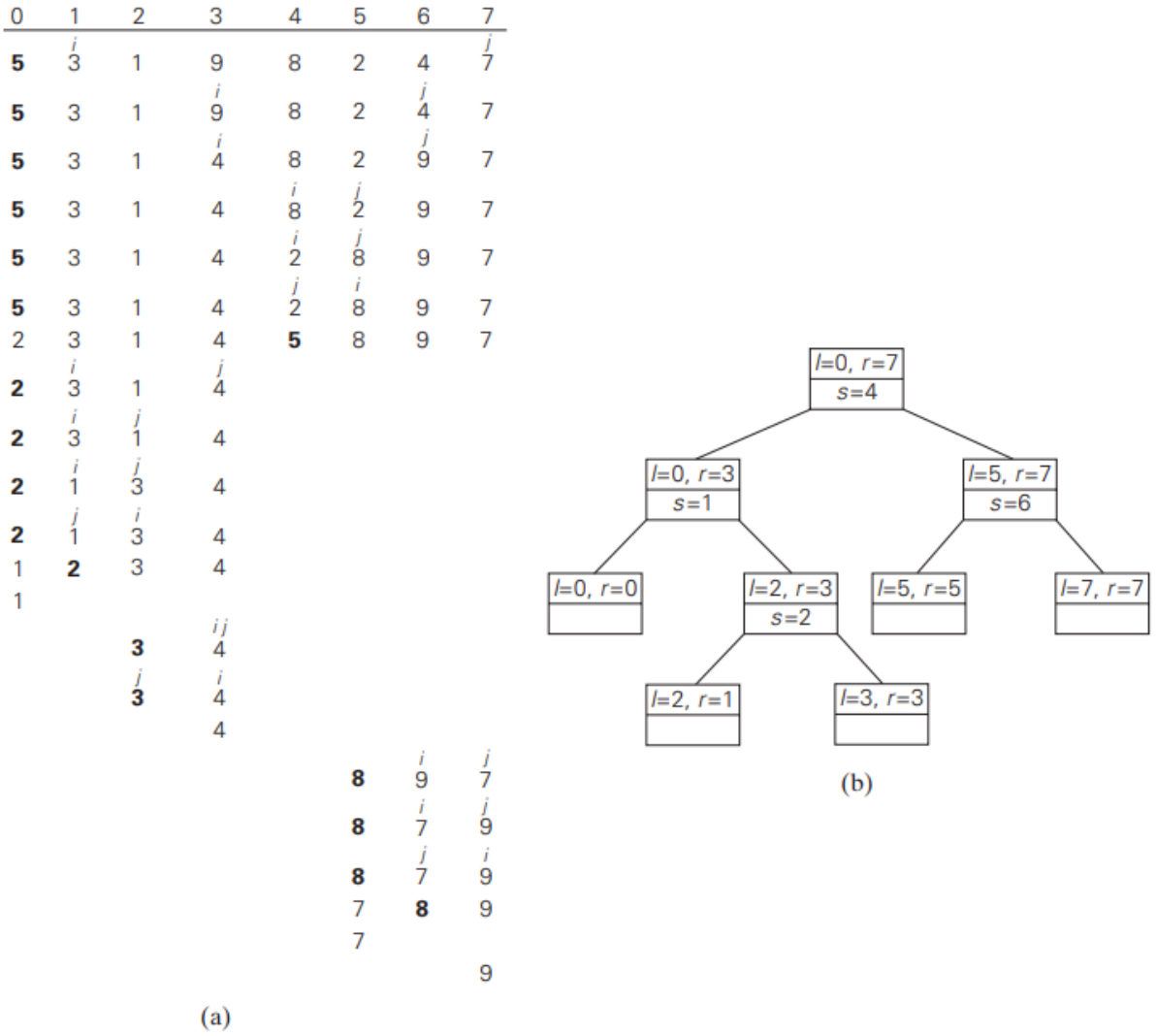


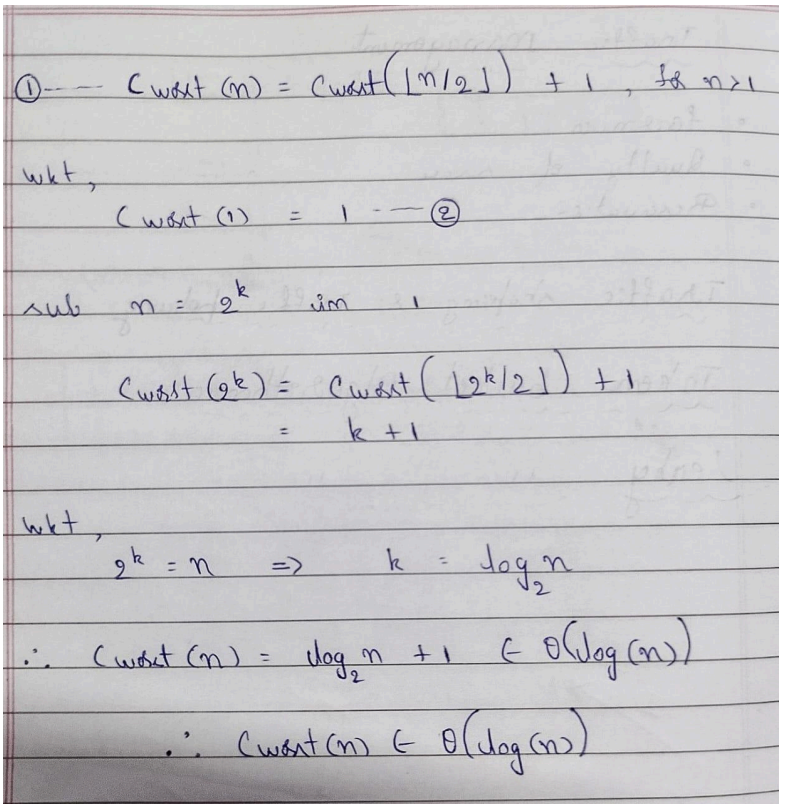
FIGURE 5.3 Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to *Quicksort* with input values and r of subarray bounds and split position s of a partition obtained.

Binary search-

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Basic operation : key comparison



Unit 2

UNIT-II	
	15 Hours
DECREASE & CONQUER: General method, Insertion Sort algorithm, **Graph algorithms:** Depth First Search, Breadth First Search, Topological Sorting with complexity analysis **TRANSFORM AND CONQUER:** General method, **Balanced Search Trees:** AVL trees, 2-3 trees, Heaps and Heap sort algorithms with complexity analysis **TIME AND SPACE TRADEOFFS:** Sorting by counting,Input Enhancement in String Matching: Horspool's algorithm and analysis **(Text Book-1: Chapter 5: 5.1 to 5.3, Chapter 6: 6.3 to 6.4, Chapter 7:7.1, 7.2)** **DYNAMIC PROGRAMMING:** General method, The Floyd-Warshall Algorithm, The Knapsack problem and memory function with complexity study **(Text Book-1: Chapter 8: 8.2 and 8.4).**	

Decrease & Conquer

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

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

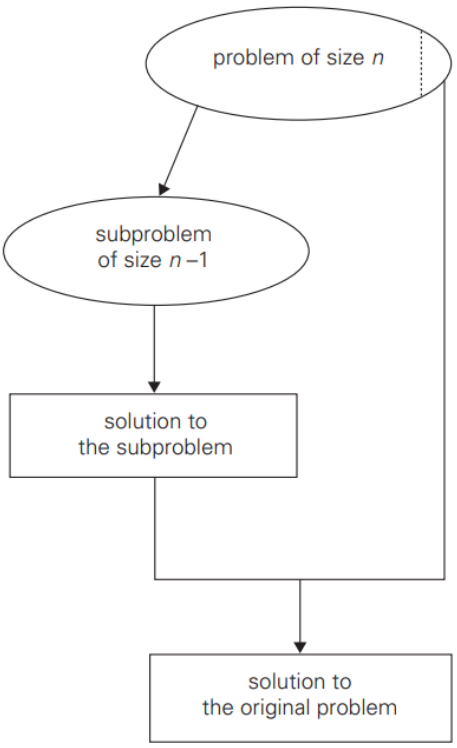


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

- In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.
- Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

Example:

Consider, as an example, the exponentiation problem of computing a^n where a = 0 and n is a nonnegative integer. The relationship between a solution to an instance of size n and an instance of size n − 1 is obtained by the obvious formula $a^n = a^{n-1}$. a. So the function f (n) = an 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. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.)

Decrease-by-a-constant-factor

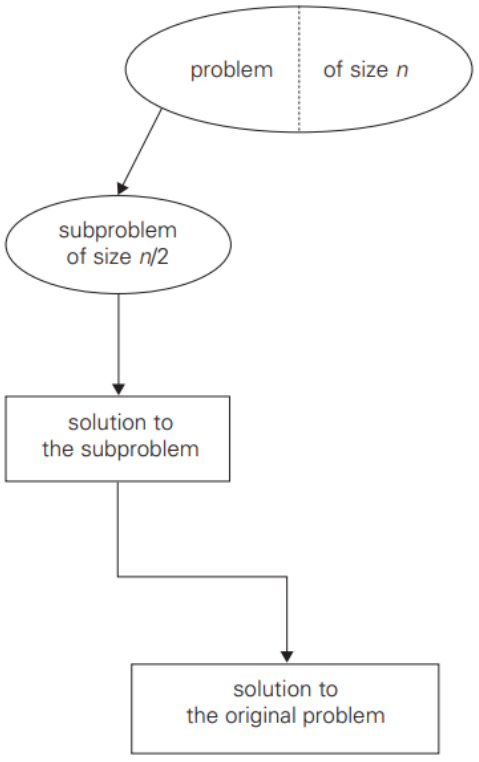


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

- The decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm.

- In most applications, this constant factor is equal to two.The decrease-by-half idea is illustrated in Figure 4.2.

Example:

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = a^{(n/2)^2}$. But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Variable-size-decrease

Finally, in the variable-size-decrease variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another.

Example :

Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula gcd(m, n) = gcd(n, m mod n). Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

Insertion sort

For logic/tracing watch:

7.4 Insertion Sort Algorithm |Explanation with C Program| Data Structure Tutorials

Animation/Visualization:

INSERTION SORT | Sorting Algorithms | DSA | GeeksforGeeks

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
```

Example :

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.

Worst case efficiency

- The basic operation of the algorithm is the key comparison A[j]> v.
- In the worst case, A[j] > v is executed the largest number of times, i.e., for every j = i − 1,..., 0.
- Since v = A[i], it happens if and only if A[j] > A[i] for j = i − 1,..., 0.
- Thus, for the worst-case input, we get A[0]> A[1] (for i = 1), A[1] > A[2] (for i = 2),...,A[n − 2] > A[n − 1] (for i = n − 1).

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

Best case efficiency

In the best case, 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] ≤ A[i] for every i = 1,...,n − 1, i.e., if the input array is already sorted in ascending order.

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

Average case efficiency

A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

Note:

- Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively.

- **Almost-sorted arrays** do arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs.

Depth first search

Working : L-4.15: BFS & DFS | Breadth First Search | Depth First Search | Graph Trav…

Or

6.2 BFS and DFS Graph Traversals| Breadth First Search and Depth First Search | Da…

ALGORITHM DFS(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph G = {V,E}
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are first encountered by the DFS traversal mark each vertex in V with 0 as a mark of being "unvisited"

count ← 0
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count

count ← count + 1; mark v with count
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)

Example:

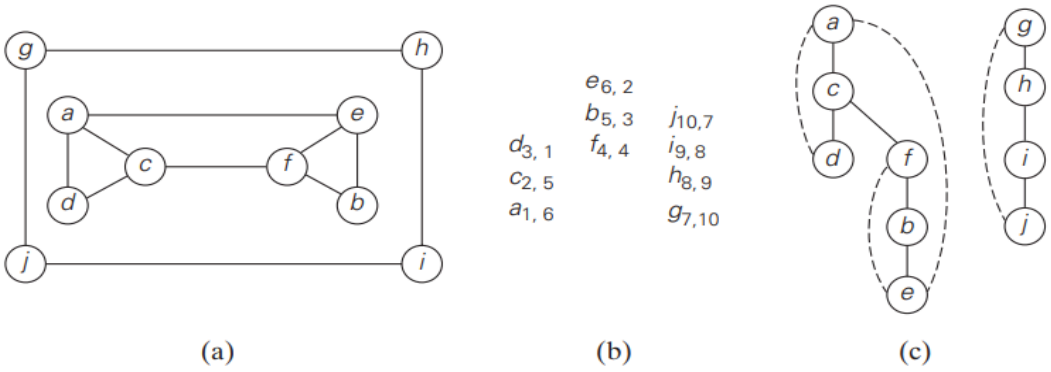


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

Description:

- Depth-first search starts a graph's traversal at an arbitrary vertex by marking it as visited.
- On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices, a tie can be resolved arbitrarily. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph. In our examples, we always break ties by the alphabetical order of the vertices.)
- This process continues until a dead end—a vertex with no adjacent unvisited vertices— is encountered.
- At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.
- The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.
- By then, all the vertices in the same connected component as the starting vertex have been visited.
- If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

Depth-first search forest.

- It is also very useful to accompany a depth-first search traversal by constructing the so-called **depth-first search forest**.
- The starting vertex of the traversal serves as the root of the first tree in such a forest.
- Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a **tree edge** because the set of all such edges forms a forest.
- The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a **back edge** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest.

Applications

1. Detecting cycle in a graph: A graph has a cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

2. Path Finding: We can specialise the DFS algorithm to find a path between two given vertices u and z.

Advantages of Depth First Search:

- **Memory Efficiency**: DFS has a low memory requirement compared to breadth-first search, as it only needs to store a stack of nodes on the current path.
- **Space Complexity:** The memory requirement for DFS is linear with respect to the search graph, making it more space-efficient than algorithms like breadth-first search.

- **Simplicity of Implementation:** DFS is relatively simple to implement compared to some other search algorithms, making it a practical choice for certain applications.

Disadvantages of Depth First Search:

- **Completeness:** DFS is not guaranteed to find a solution, and in some cases, it might get stuck in an infinite loop if the search space is infinite or if the solution is located along an infinite path.
- **Non-Optimality:** If there are multiple solutions to a problem, DFS does not guarantee finding the optimal solution..
- **Path Dependence:** The performance of DFS can heavily depend on the order in which nodes are explored. If the algorithm happens to choose a path that leads away from the solution, it may take a long time to backtrack and explore other paths.

Breadth first search

ALGORITHM *BFS*(G)

//Implements a breadth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
  if v is marked with 0
    bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count* and initialize a queue with v
while the queue is not empty **do**
  for each vertex w in V adjacent to the front vertex **do**
    if w is marked with 0
      $count \leftarrow count + 1$; mark w with *count*
      add w to the queue
  remove the front vertex from the queue

Example:

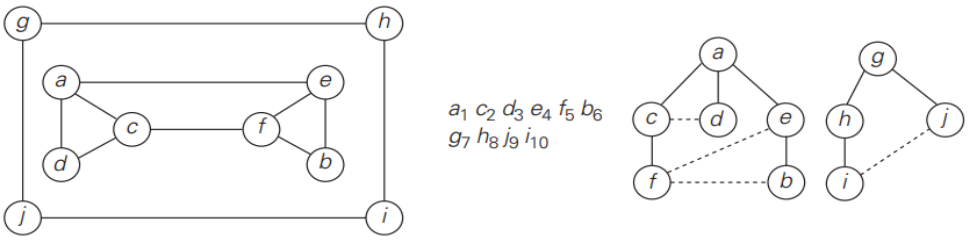


FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

le :

Description:

- Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called **breadth-first search forest**.
- The traversal's starting vertex serves as the root of the first tree in such a forest.
- Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a **tree edge**.
- If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a **cross edge**.

Differentiate between DFS and BFS

	DFS	**BFS**
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Topological sorting

Some definitions

- A **directed graph**, or digraph for short, is a graph with directions specified for all its edges

- A **directed cycle** in a graph is a sequence of vertices where there is a directed edge from each vertex to the next, and the last vertex in the sequence has a directed edge back to the first vertex, forming a closed loop.

Why topological sorting is needed?

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 following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

Topological sorting is used to overcome this problem

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

The First algorithm for topological sorting

- Perform a DFS traversal on the directed graph.
- While performing the DFS traversal, keep track of the order in which vertices become dead-ends or pop the vertices ones the become a dead end (i.e., vertices from which there are no more unexplored outgoing edges).
- Once the DFS traversal is complete, reverse the order in which the vertices became dead-ends or reverse the order of poped vertices.
- The reversed order provides a potential solution to the topological sorting problem.

Example :

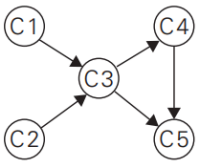


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

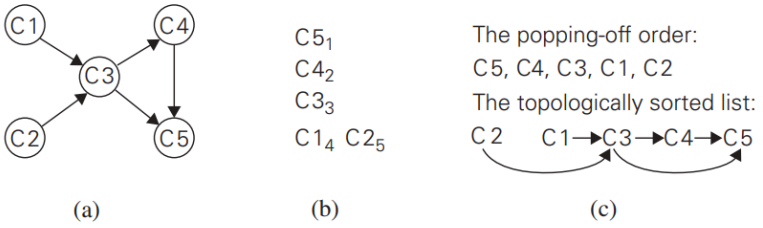


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

The Second algorithm for topological sorting

- A source vertex is defined as a vertex with no incoming edges.
- Once a source is identified, delete the source along with all the edges outgoing from it.
- Repeat the process until no vertices remain in the graph.
- The order in which the vertices are deleted during the process provides a solution to the topological sorting problem.

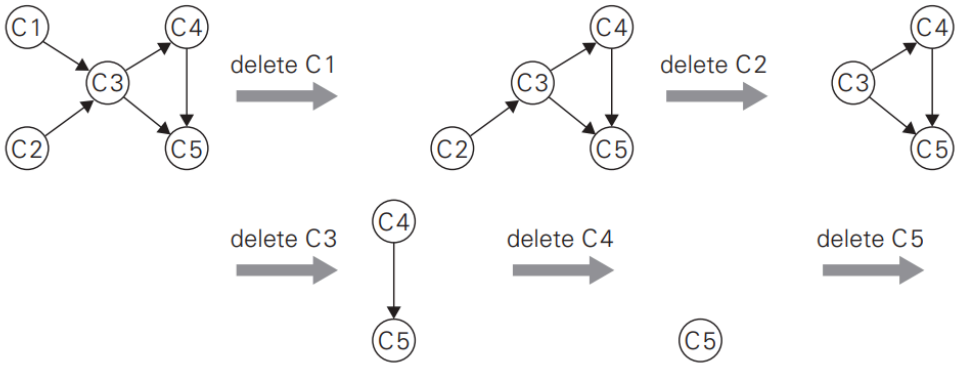


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Transform and conquer

- The "transform-and-conquer" technique is a problem-solving approach that involves two stages: transformation and conquering.
- In the transformation stage, the original problem instance is modified to make it more amenable(easily persuaded or controlled) to solution.
- In the conquering stage, the transformed problem instance is solved using appropriate methods or algorithms.

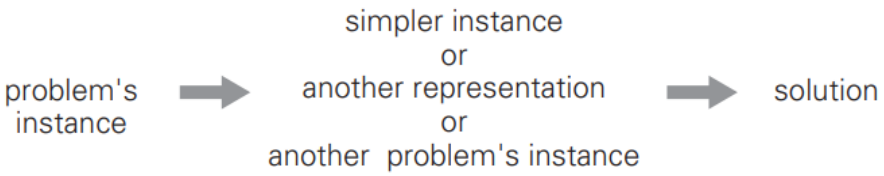


FIGURE 6.1 Transform-and-conquer strategy.

Three major variations of the transformation stage are often used:

- **Instance Simplification:** The problem instance is transformed into a simpler instance of the same problem, making it easier to solve.

- **Representation Change:** The problem instance is transformed into a different representation of the same instance. This change may reveal patterns or structures that simplify the problem.
- **Problem Reduction:** The problem instance is transformed into an instance of a different problem for which a solution algorithm is already available. This involves reducing the original problem to a known problem with a known solution.

Balanced trees

Balanced trees are data structures designed to maintain balance between elements in a tree and ensure efficient search, insertion, and deletion operations.

There are two main approaches to maintaining balance in trees:

Self-Balancing Trees (Instance-Simplification Variety):

- This approach transforms unbalanced binary search trees into balanced ones, making them self-adjusting.
- Specific types of self-balancing trees include AVL trees and red-black trees.

B-Trees (Representation-Change Variety):

- B-trees maintain balance by allowing nodes to have multiple elements, and they are particularly useful in scenarios where data may be frequently inserted or deleted.
- Different types of balanced trees falling under this category include 2-3 trees, 2-3-4 trees, and more general B-trees.

AVL trees

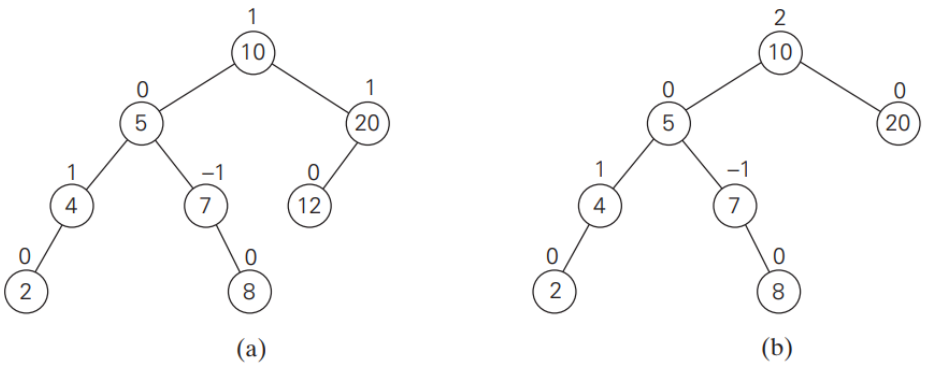


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

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.)

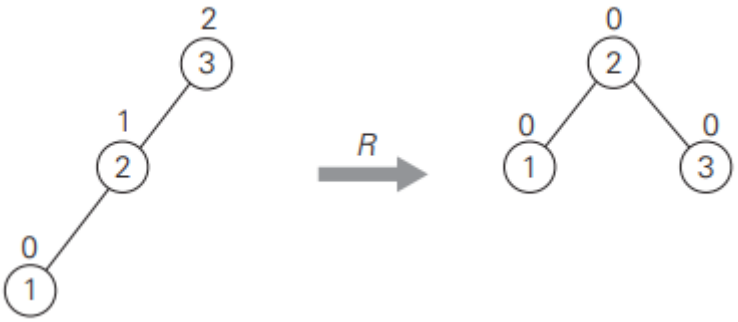
Rotations in AVL

How to Create AVL tree | LL, RR, LR, RL Rotation in AVL | Data Structure

- 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.
- There are only four types of rotations.

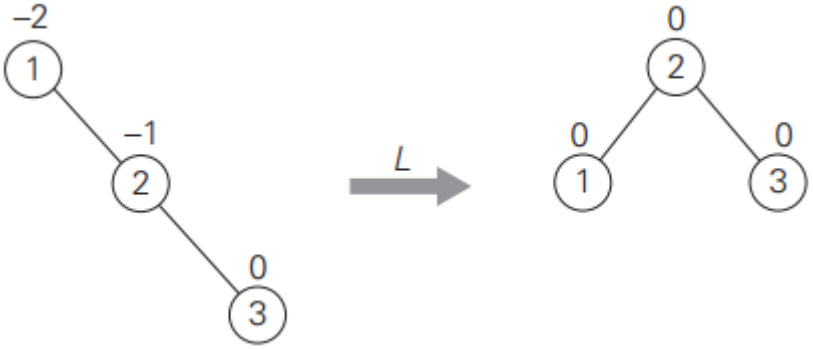
Single right rotation or R-rotation:

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.



Single left rotation or L-rotation:

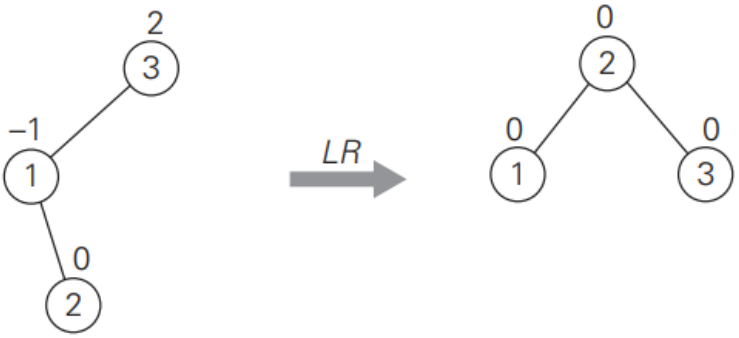
It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of −1 before the insertion.



Double left-right rotation (LR rotation):

The second rotation type is called the double left-right rotation (LRrotation). It is, in fact, a combination of two rotations:

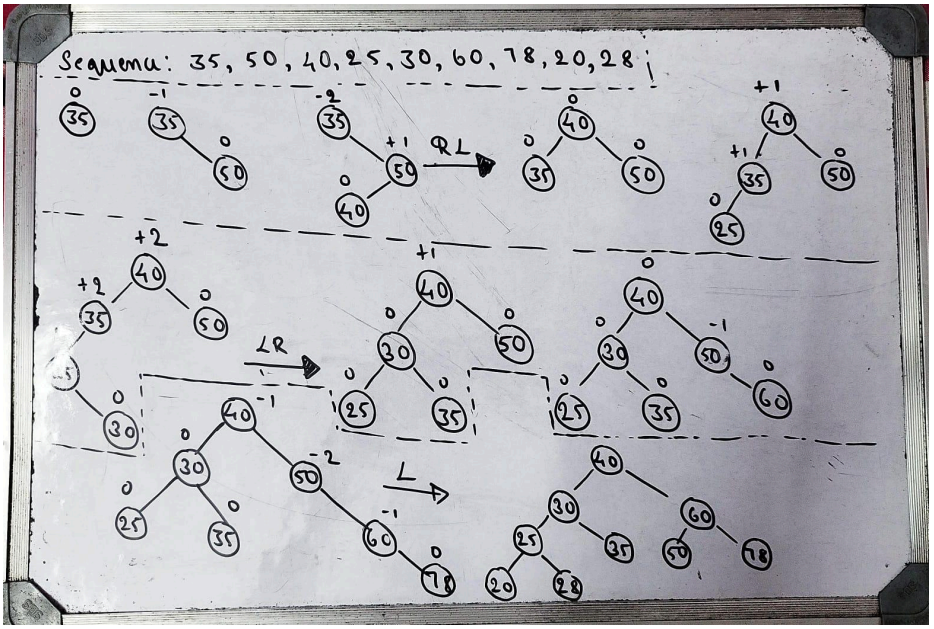
- We perform the L-rotation of the left subtree of root r followed by the R-rotation of the new tree rooted at r (Figure 6.5).
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



Double right-left rotation (RL-rotation)

The double right-left rotation (RL-rotation) is the mirror image of the double LR-rotation and is left for the exercises

Example : **AVL Tree Creation in Data Structure | All Imp Points**



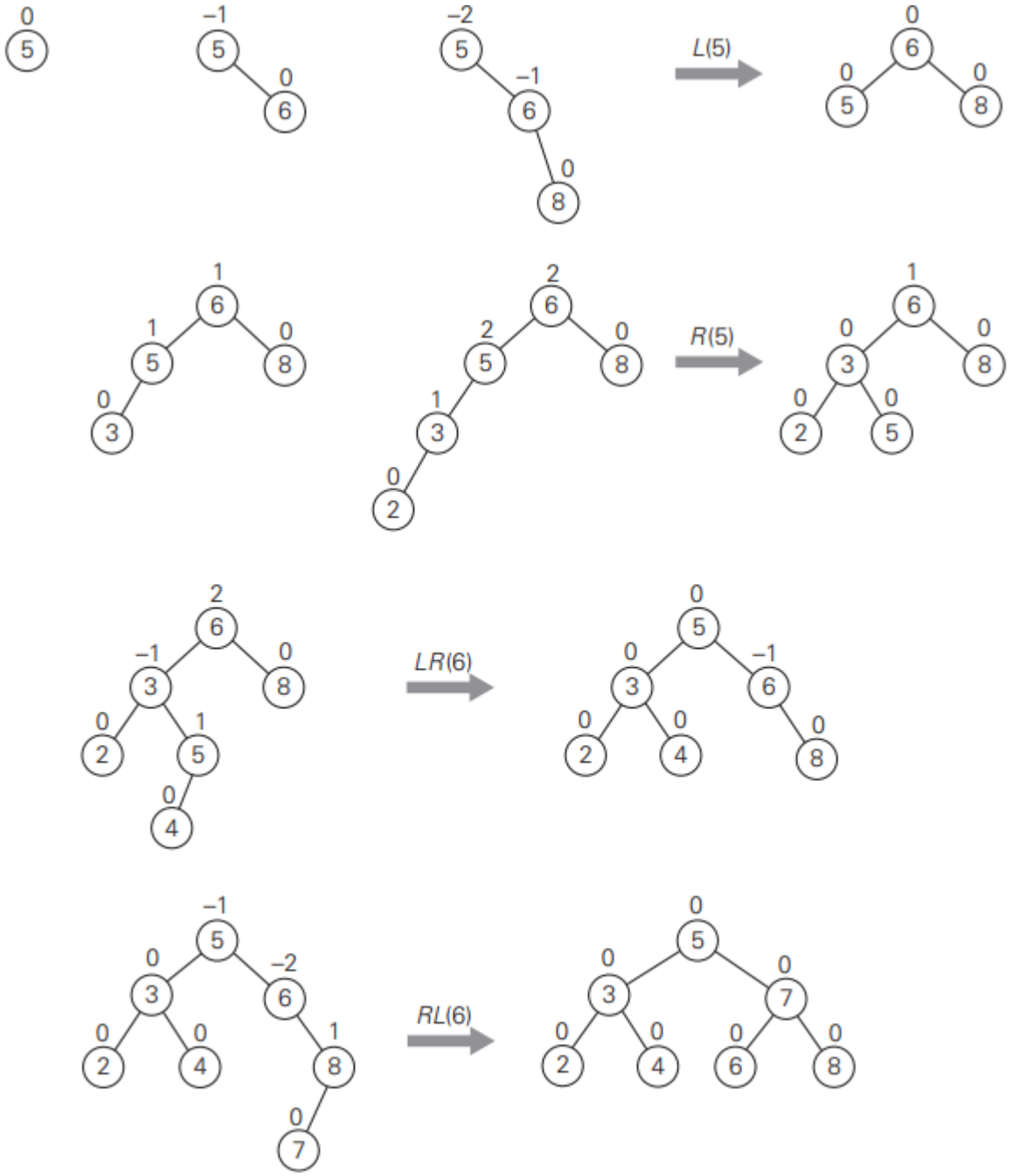
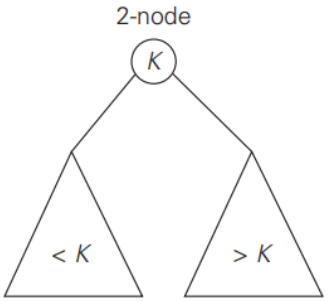


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

2 - 3 trees

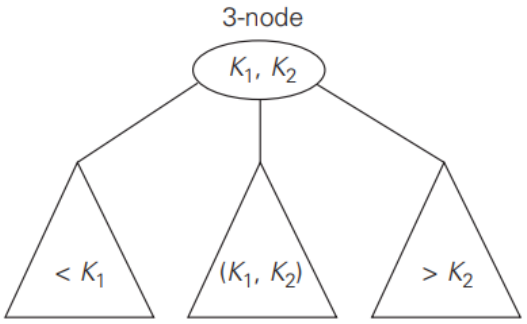
- A 2-3 tree is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

2 - node :



- 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.
- In other words, a 2-node is the same kind of node we have in the classical binary search tree.

3 - node:



- 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.

Example :

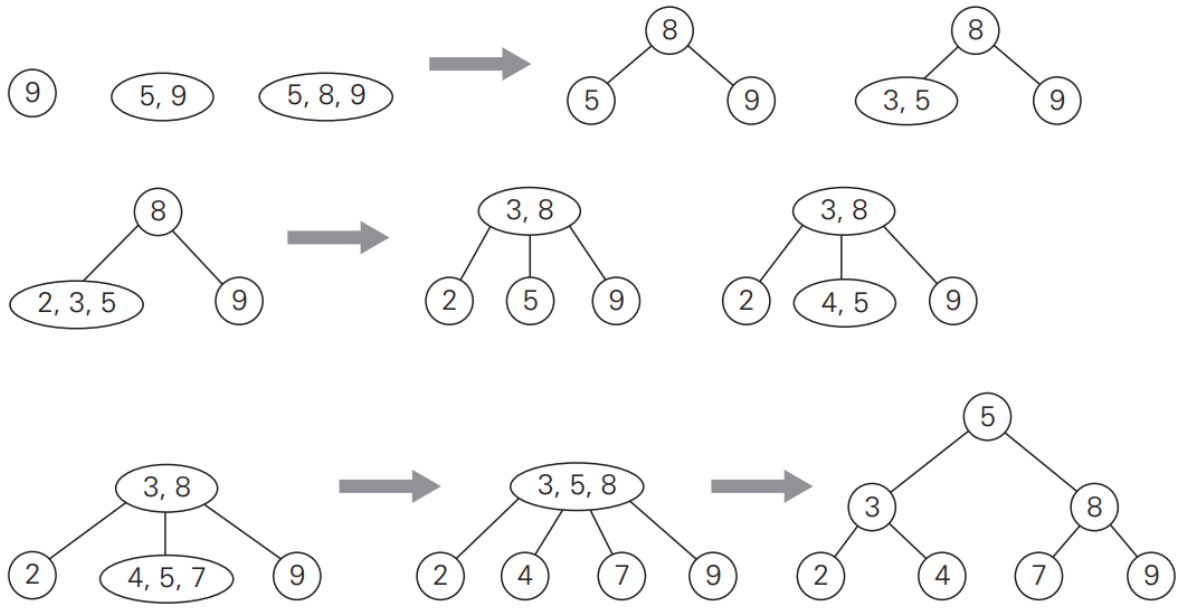


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

4. For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree.
    a. 1, 2, 3, 4, 5, 6
    b. 6, 5, 4, 3, 2, 1
    c. 3, 6, 5, 1, 2, 4

Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G. Use the alphabetical order of the letters and insert them successively starting with the empty tree.

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:

1. 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.
2. The **parental dominance** or **heap property**—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

Example :

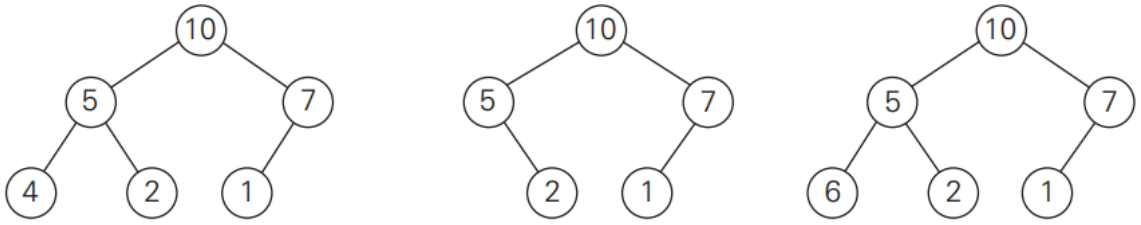


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

Consider the trees of Figure 6.9. 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.

Important properties of heap

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 $\lceil n/2 \rceil$ 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 $\lfloor i/2 \rfloor$.

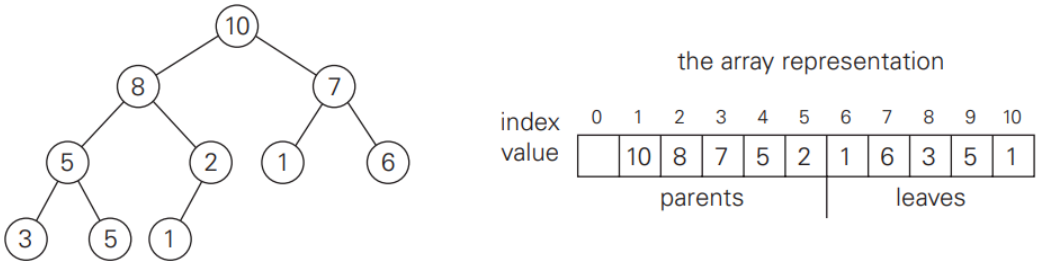


FIGURE 6.10 Heap and its array representation.

Construction of heap : Bottom-up heap construction

Initialises the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows :

- Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this 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. (Eventually, it has to because it holds automatically for any key in a leaf.)
- After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the root of the tree.

Example:

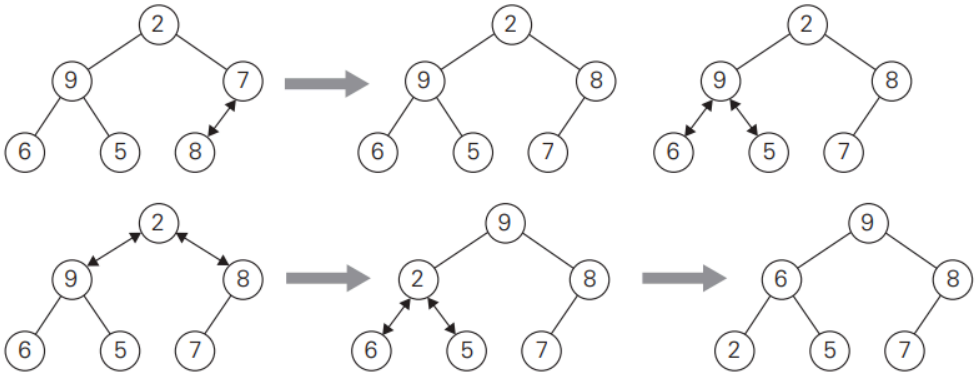


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance. I,

Bottom up heap construction algorithm

Logic : 7.9 Heap Sort | Heapify Method | Build Max Heap Algorithm | Sorting Algorith…

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
```

How efficient is this algorithm in the worst case? Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of

Time complexity

Since moving 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) where h be the height of the tree.

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

Top down heap construction

- The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap; some people call it the top-down heap construction algorithm.
- So how can we insert a new key K into a heap? First, attach a new node with key K in it after the last leaf of the existing heap.
- Then sift 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 (illustrated in Figure 6.12).

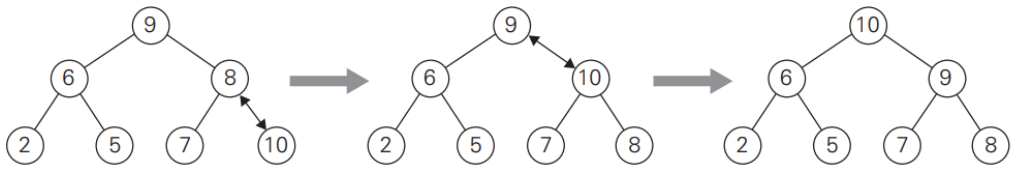


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

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.

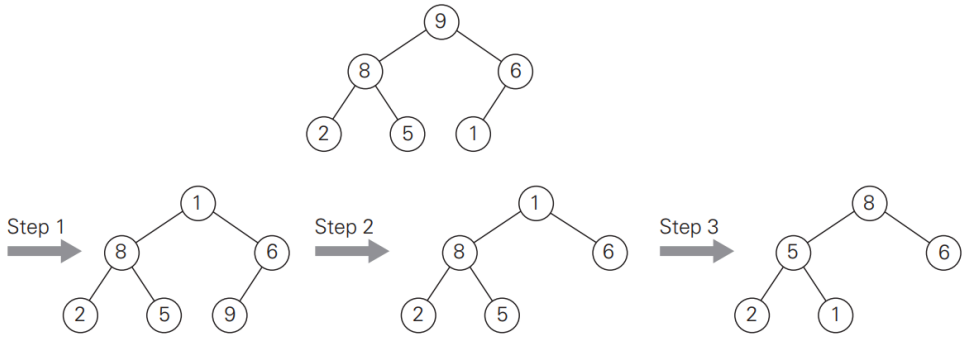


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Heapsort :

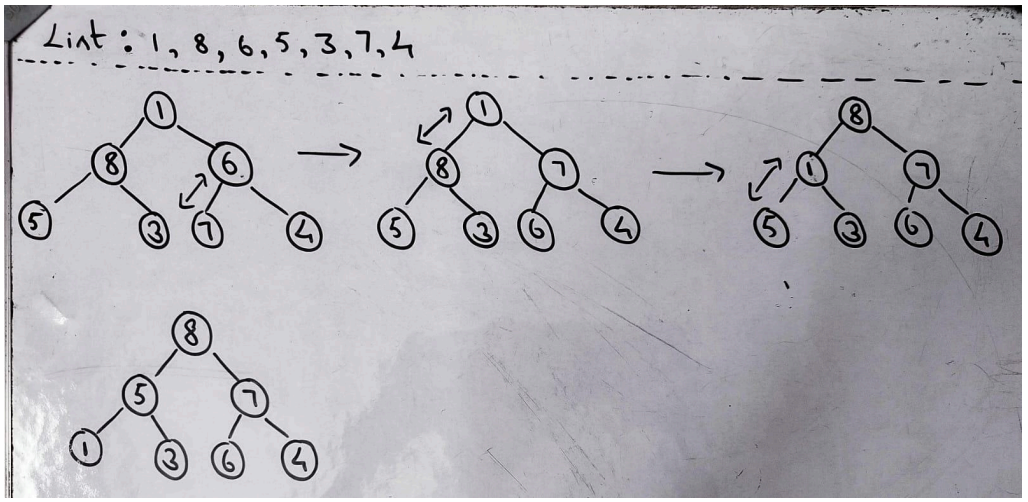
Visualise : HEAP SORT | Sorting Algorithms | DSA | GeeksforGeeks

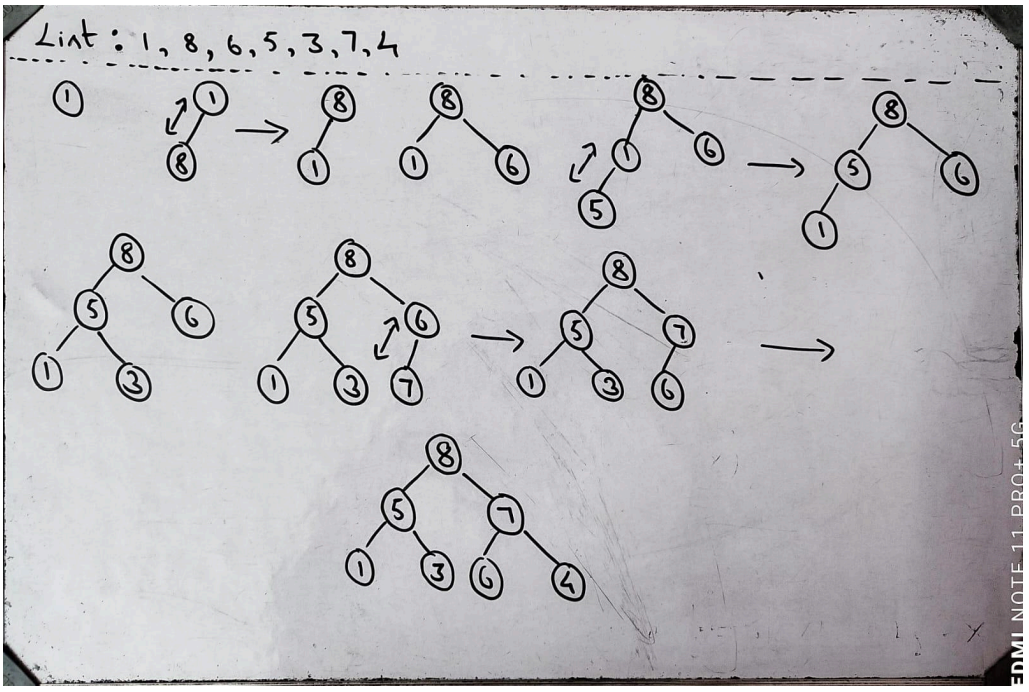
This is a two-stage sorting algorithm that works as follows.
Stage 1 (heap construction): Construct a heap for a given array.
Stage 2 (maximum deletions): Apply the root-deletion operation n − 1 times to the remaining heap.

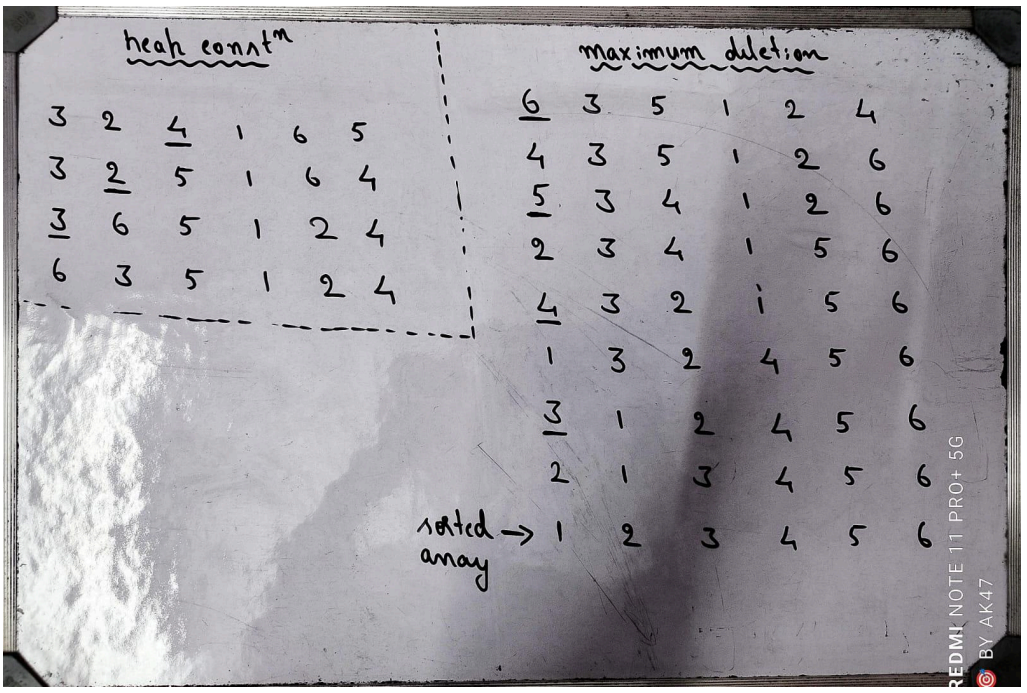
Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

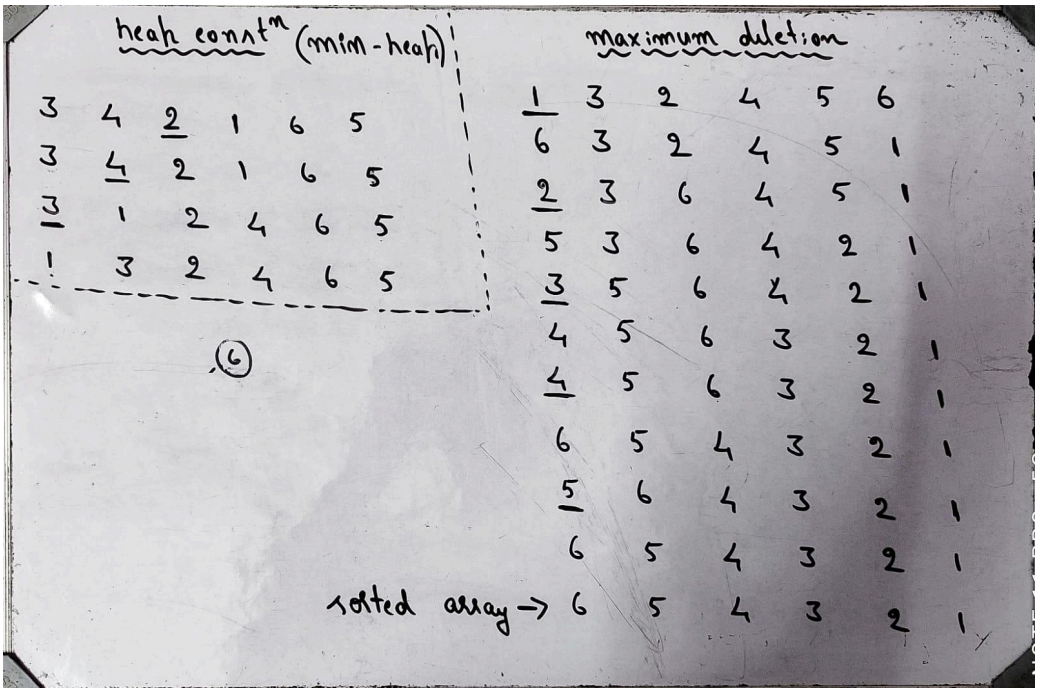
FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

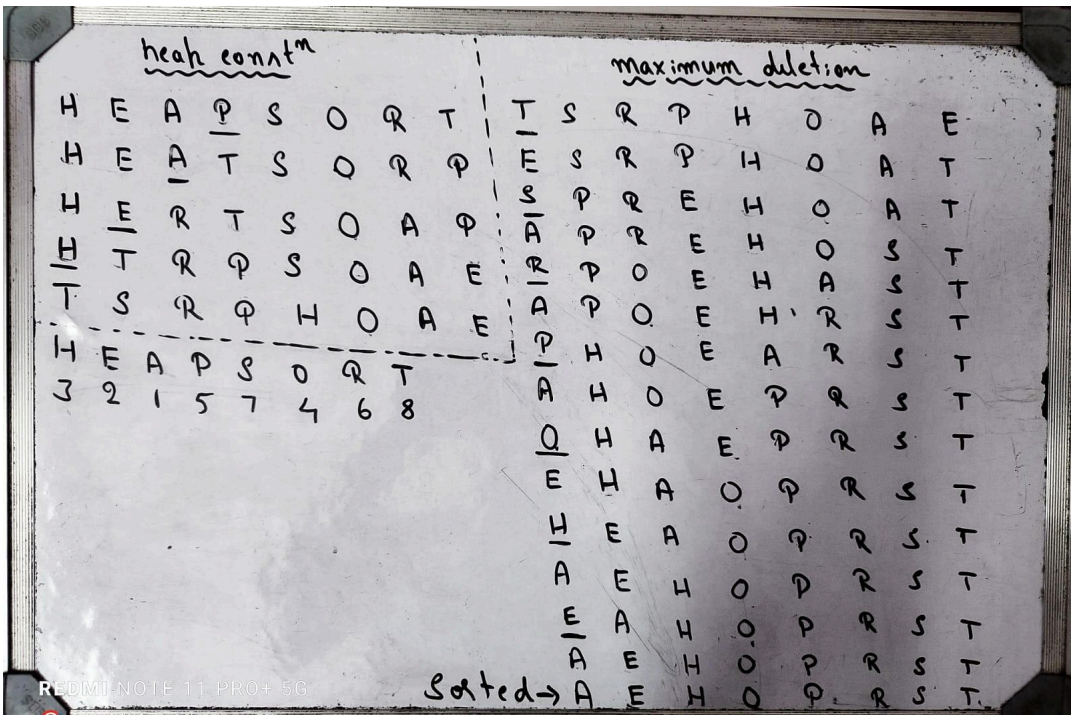
1. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.
2. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down approach).
3. Sort the below given lists by heap sort using the array representation of heaps:
    1. 3, 2, 4, 1, 6, 5 (Increasing order)
    2. 3, 4, 2, 1, 5, 6 (Decreasing order)
    3. H, E, A, P, S, O, R, T (alphabetical order)






heap constⁿ
3 2 4 1 6 5
3 2 5 1 6 4
3 6 5 1 2 4
6 3 5 1 2 4
maximum deletion
6 3 5 1 2 4
4 3 5 1 2 6
5 3 4 1 2 6
2 3 4 1 5 6
4 3 2 1 5 6
1 3 2 4 5 6
3 1 2 4 5 6
2 1 3 4 5 6
sorted array → 1 2 3 4 5 6
REDMI NOTE 11 PRO+ 5G
BY AK47


heap constⁿ (min-heap)
3 4 2 1 6 5
3 4 2 1 6 5
3 1 2 4 6 5
1 3 2 4 6 5
⑥
maximum deletion
1 3 2 4 5 6
6 3 2 4 5 1
2 3 6 4 5 1
5 3 6 4 2 1
3 5 6 4 2 1
4 5 6 3 2 1
4 5 6 3 2 1
6 5 4 3 2 1
5 6 4 3 2 1
6 5 4 3 2 1
sorted array → 6 5 4 3 2 1


heap constn
H E A P S O R T
H E A T S O R P
H E R T S O A P
H T R P S O A E
T S R P H O A E
H E A P S O R T
3 2 1 5 7 4 6 8
maximum deletion
T S R P H O A E
E S R P H O A T
S P R E H O A T
R P R E H O S T
A P O E H A S T
P H O E A R S T
A H O E P R S T
Q H A E P R S T
E H A O P R S T
H E A O P R S T
A E H O P R S T
E A H O P R S T
A E H O P R S T
Sorted→ A E H O P R S T

Space and time tradeoffs

Sort by counting

Input enhancement : Input enhancement is an approach where additional information is derived from preprocessing the problem's input and is used for solving the problem.

Comparison counting sort : Comparison Count Sorting

Example :

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.

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
```

More formally, the number of times its basic operation, the comparison **A[i] < A[j]**, is executed is equal to the sum we have encountered several times already:

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

Distribution counting : 34 Comparison Count Sort & Distribution Count Sort

Distribution counting sort is used in a situation where elements to be sorted belong to a **known small set of values.**

Example :

EXAMPLE Consider sorting the array

13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

	$D[0..2]$			$S[0..5]$					
$A[5] = 12$	1	**4**	6				12		
$A[4] = 12$	1	**3**	6			12			
$A[3] = 13$	1	2	**6**						13
$A[2] = 12$	1	**2**	5		12				
$A[1] = 11$	**1**	1	5	11					
$A[0] = 13$	0	1	**5**					13	

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

Algorithm :

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

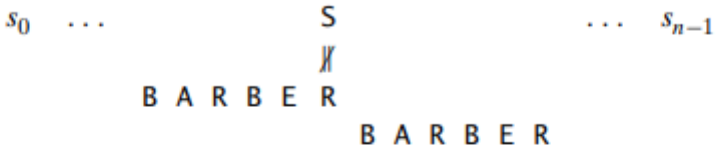
```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

- Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array A.
- This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered

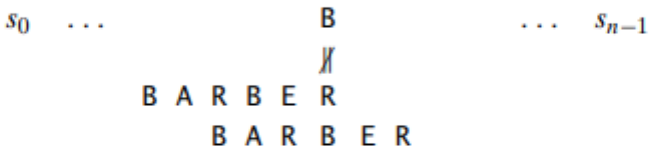
Horspool's Algorithm : 35 Horspool's String Matching

In horspool's algorithm while matching the given pattern with the text, the following four possibilities can occur.

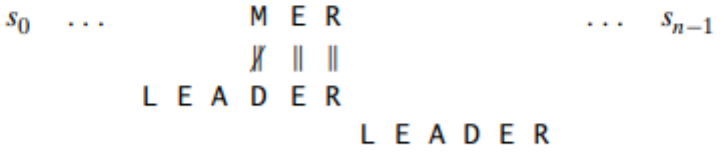
Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



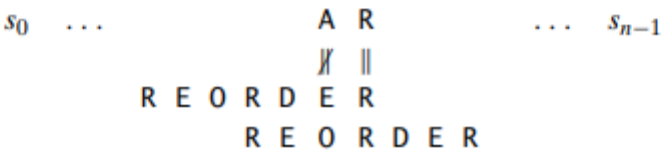
Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:



Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:



Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:



Formula to construct the table

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

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
```

■

Algorithm to generate shift table and string matching

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 $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

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
```

Dynamic programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems and solving each subproblem only once, storing the results in a table or array.

Warshall's algorithm : 41 Warshall's Algorithm

Transitive closure : The transitive closure of a directed graph with n vertices can be defined as the n × n boolean matrix T = {tij}, in which the element in the ith row and the j th column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the j th vertex; otherwise, tij is 0.

An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 8.11.

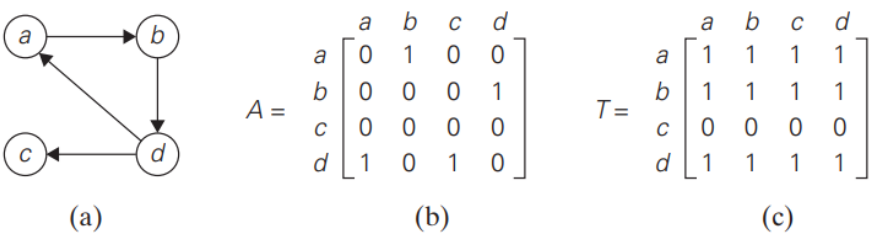


FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Rule for changing zeros in Warshall's algorithm.

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$. This rule is illustrated in Figure 8.12.

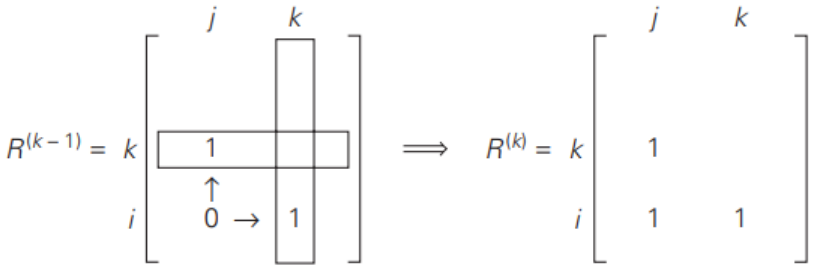
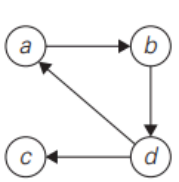


FIGURE 8.12 Rule for changing zeros in Warshall's algorithm.

Example:



$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)} =$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

Warshall's algorithm

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency of warshall's algorithm is : $\theta(n^3)$.

Floyd's algorithm : 39 Floyd's Algorithm

Underlying idea of Floyd's algorithm

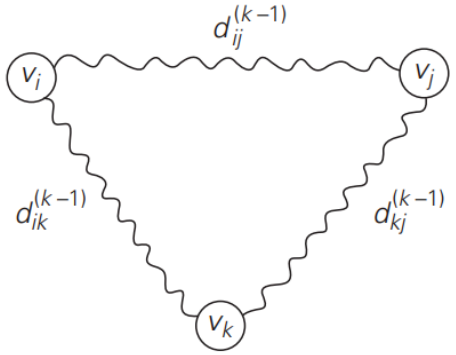


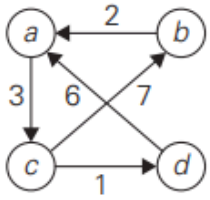
FIGURE 8.15 Underlying idea of Floyd's algorithm.

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}. \qquad \textbf{(8.14)}$$

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W  //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Example:



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

The knapsack problem :

0/1 knapsack problem-Dynamic Programming | Data structures and algorithms

Or

4.5 0/1 Knapsack - Two Methods - Dynamic Programming

Recurrence relation:

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

FIGURE 8.4 Table for solving the knapsack problem by dynamic programming.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases} \quad \textbf{(8.6)}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0. \quad \textbf{(8.7)}$$

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

The dynamic programming table, filled by applying formulas (8.6) and (8.7), is shown in Figure 8.5.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

FIGURE 8.5 Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

- Thus, the maximal value is F (4, 5) = $37.
- We can find the composition of an optimal subset by backtracing the computations of this entry in the table.
- Since F (4, 5)>F(3, 5), item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity.
- The value of the latter is F (3, 3). Since F (3, 3) = F (2, 3), item 3 need not be in an optimal subset.
- Since F (2, 3)>F(1, 3), item 2 is a part of an optimal selection, which leaves element F (1, 3 − 1) to specify its remaining composition.
- Similarly, since F (1, 2)>F(0, 2), item 1 is the final part of the optimal solution **{item 1, item 2, item 4}.**

Memory functions : 42 1 Memory Function Knapsack Problem

The following algorithm implements this idea for the knapsack problem. After initializing the table with special null values(ie. -1), the recursive function needs to be called with i = n (the number of items) and j = W (the knapsack capacity).

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

Solution:

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.

UNIT 3

UNIT-III

GREEDY TECHNIQUE:
General method of Greedy technique, **Minimum Spanning Trees:** Prim's Algorithm, Kruskal's Algorithm, Single-Source Shortest Paths using Dijkstra's Algorithm, Huffman Trees
(Text Book-1: Chapter 9: 9.1 to 9.4)
The Bellman-Ford algorithm, **(Text Book-2: Chapter 24: 24.1).**
BACKTRACKING: General method, State space trees and algorithms for N-Queens problem, Subset-sum problem
(Text Book-1: Chapter 12: 12.1 selected topics)
BRANCH AND BOUND: General method, Solving job Assignment Problem ,Travelling Salesman problem, Knapsack Problem using branch and bound method
(Text Book-1: Chapter 12: 12.2)
P,NP and NP Complete Problems (**Text Book-1: Chapter 11: 11.3)**

Greedy Technique

- The greedy technique is a problem-solving approach commonly used in algorithm design.
- It involves making a sequence of choices, each time selecting the best local option available at that moment.

The key characteristics of the greedy technique can be summarised as follows:

- **Feasibility:** At each step, the choice made must satisfy the problem's constraints.
- **Local optimality:** The choice made at each step should be the best among all feasible options available at that step.
- **Irrevocability:** Once a choice is made, it cannot be changed in subsequent steps of the algorithm.

To apply the greedy technique effectively, one typically follows these steps:

- **Identify a strategy:** Understand the problem and determine a strategy for making choices at each step. This often involves selecting the option that seems best at the moment without considering future steps.
- **Iterative process:** Apply the chosen strategy iteratively, making locally optimal choices at each step until a complete solution is reached.
- **Prove optimality:** To demonstrate that the greedy algorithm yields an optimal solution, it's essential to prove that the locally optimal choices made at each step collectively lead to the best possible outcome.

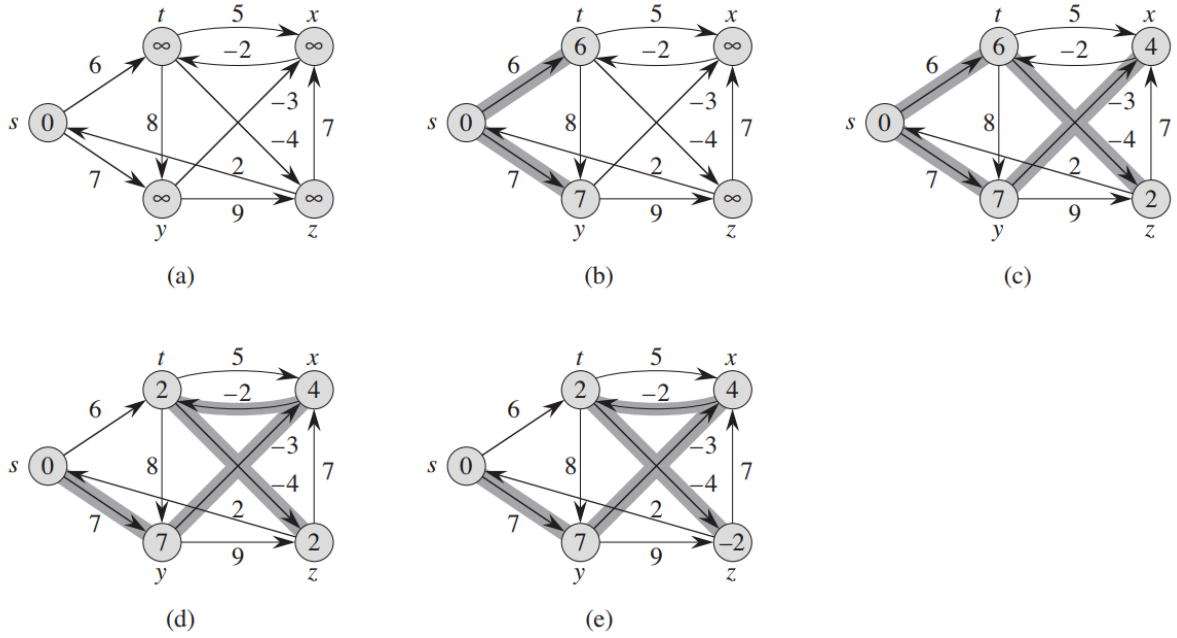
Bellman-ford Algorithm : 9 Bellman Ford 1

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source

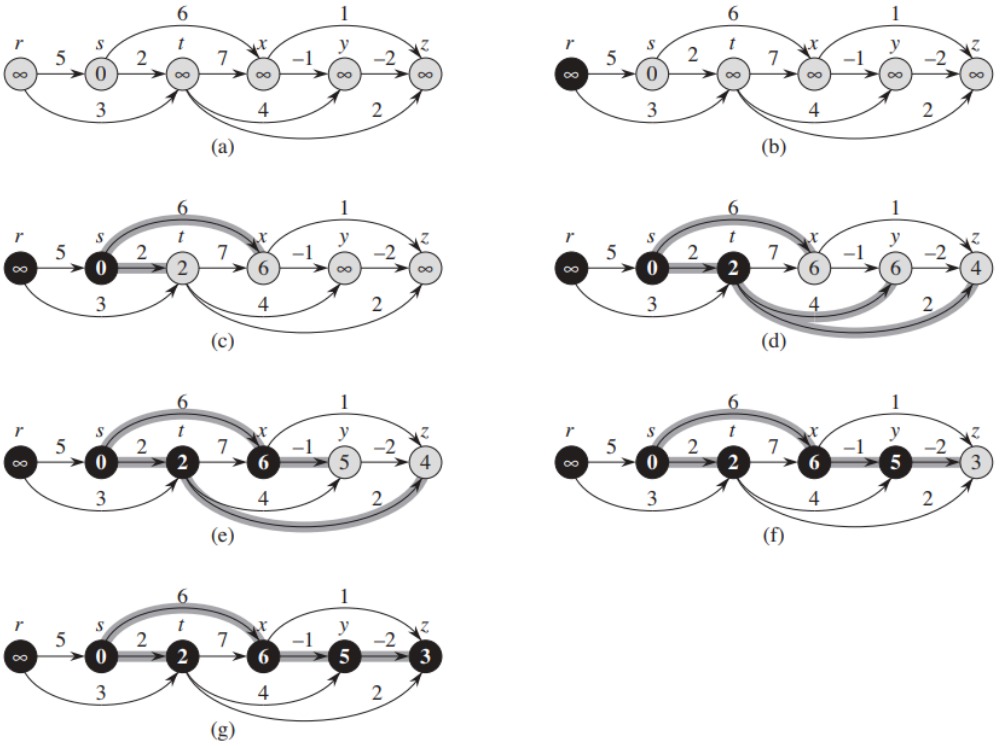
Example:



Single-source shortest paths in directed acyclic graphs

11 Single Source Shortest Path in DAG

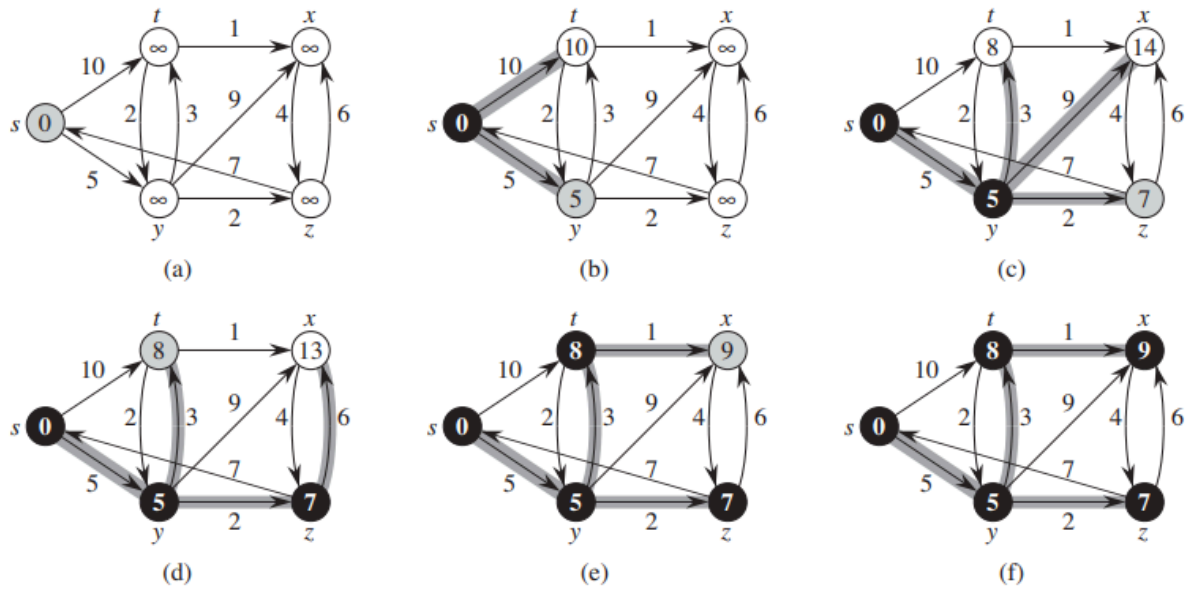
```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```



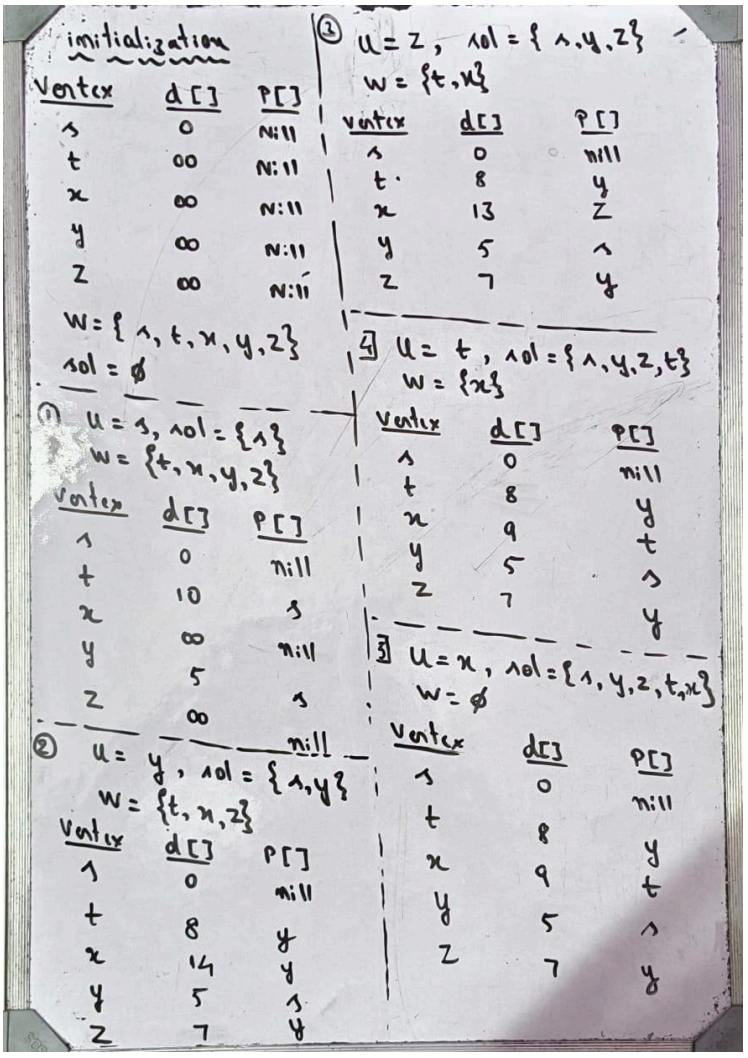
Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative

Example:


(a)
(b)
(c)
(d)
(e)
(f)

Tracing:


initialization
Vertex d[] P[]
s 0 NIL
t ∞ NIL
x ∞ NIL
y ∞ NIL
z ∞ NIL
W = {s, t, x, y, z}
sol = ∅
① u = s, sol = {s}
W = {t, x, y, z}
Vertex d[] P[]
s 0 nill
t 10 s
x ∞ nill
y 5 s
z ∞ nill
② u = y, sol = {s, y}
W = {t, x, z}
Vertex d[] P[]
s 0 nill
t 8 y
x 14 y
y 5 s
z 7 y
③ u = z, sol = {s, y, z}
W = {t, x}
Vertex d[] P[]
s 0 nill
t 8 y
x 13 z
y 5 s
z 7 y
④ u = t, sol = {s, y, z, t}
W = {x}
Vertex d[] P[]
s 0 nill
t 8 y
x 9 t
y 5 s
z 7 y
⑤ u = x, sol = {s, y, z, t, x}
W = ∅
Vertex d[] P[]
s 0 nill
t 8 y
x 9 t
y 5 s
z 7 y

48 Dijkstra's Algorithm Single Source Shortest Path

ALGORITHM Dijkstras(G,s)

//Output : The shortest path from s to all vertices in V

```
for each vertex v in V do
        d[v] ← ∞
        p[v] ← nil
d[s] ← 0
soln  ← ϕ
w ← V

while w ≠ ϕ
        u ← EXTRACT_MIN(w)
        soln ← soln ∪ {u}
        w ← w - {u}
        for each vertex v in V adjacent to u
                if d[v] > d[u] + w(u,v) //w(u,v) is weight from vertex u to v
                        d[v] ← d[u] + w(u,v)
                        p[v] ← u
return d
```

OR

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u 2 V S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

ALGORITHM

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

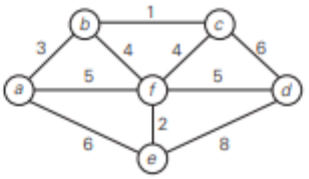
Minimum spanning trees

Prim’s Algorithm

- **DEFINITION** A spanning tree of an undirected connected graph is its **connected acyclic** subgraph (i.e., a tree) that **contains all the vertices** of the graph.
- If such a graph has weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.
- Prim’s Algorithm is used to find the minimum cost spanning tree

Examples :

L-4.9: Prim's Algorithm for Minimum Cost Spanning Tree | Prims vs Kruskal



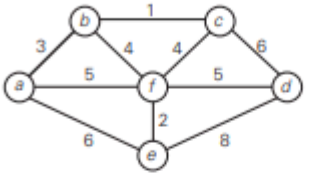
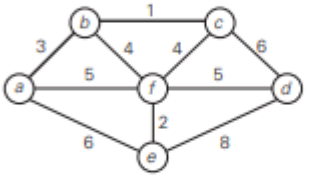
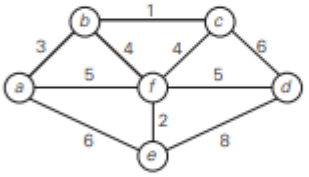
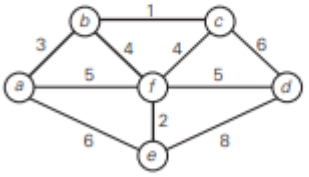
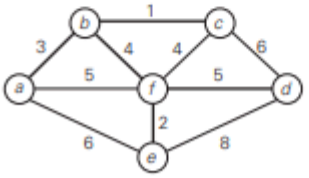
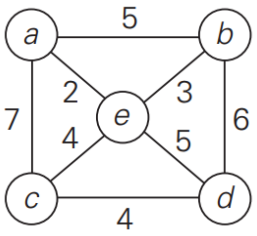
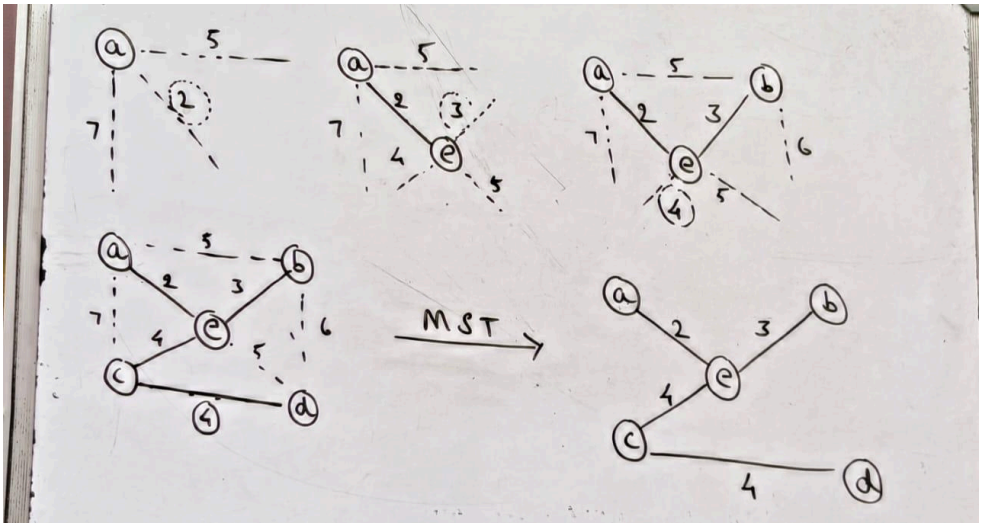
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b**(**a**, **3**) c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c**(**b**, **1**) d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f**(**b**, **4**)	
f(b, 4)	d(f, 5) **e**(**f**, **2**)	
e(f, 2)	**d**(**f**, **5**)	
d(f, 5)		

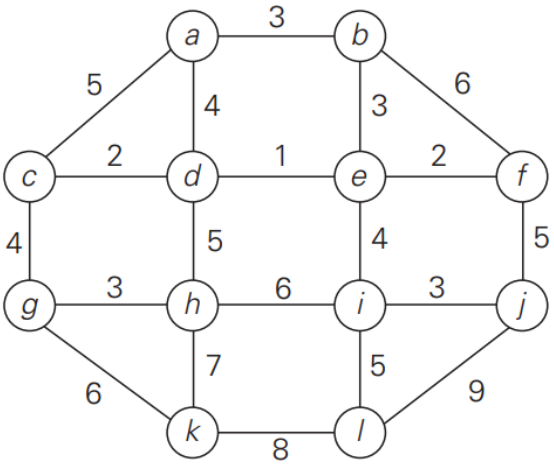
FIGURE 9.3 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

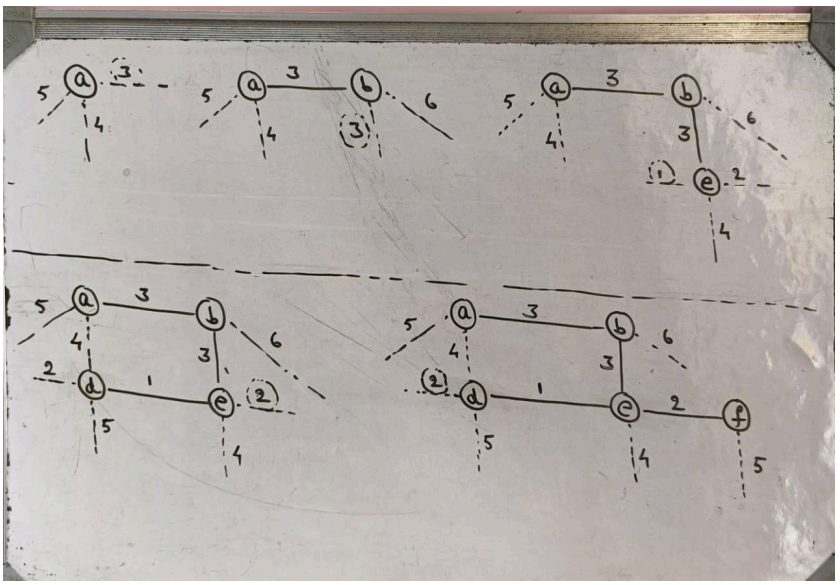
Example 2 :

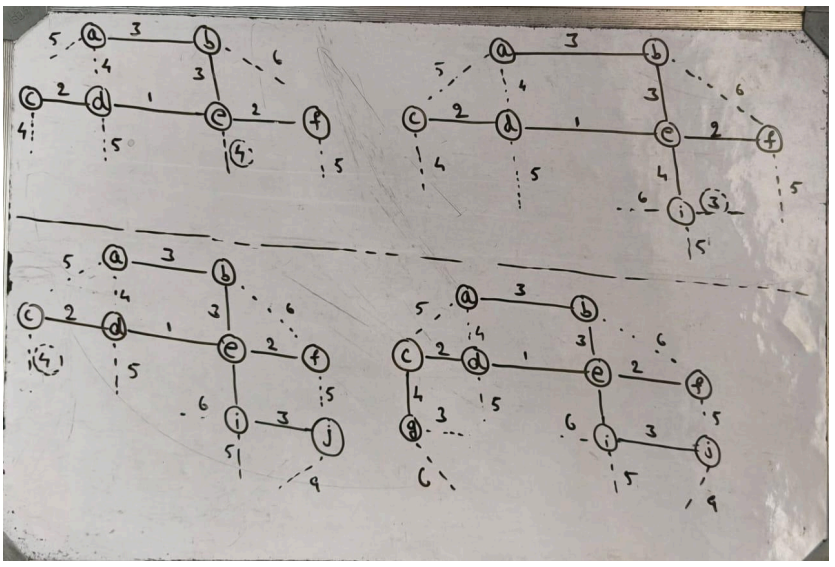


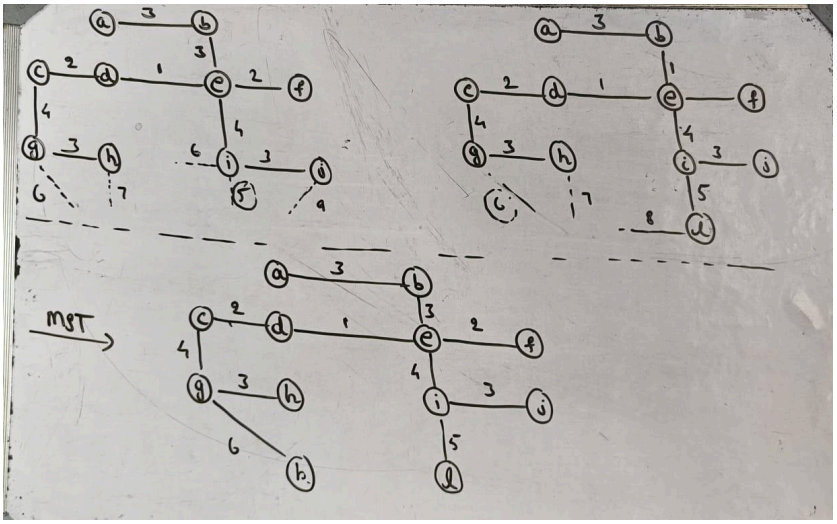


Example 3 :








MST

45 Introduction to Greedy Technique & Prims Algorithm to find MST

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

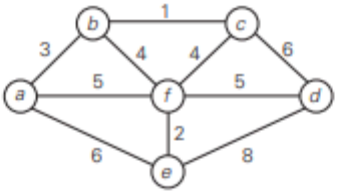
Hence, the running time of this implementation of Prim's algorithm is in

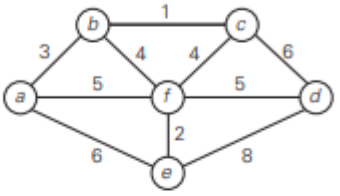
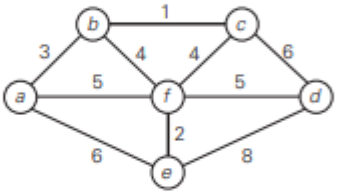
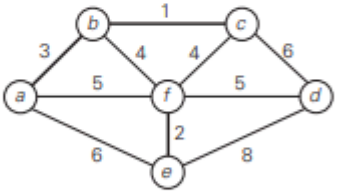
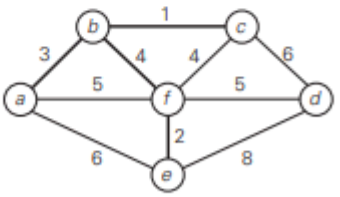
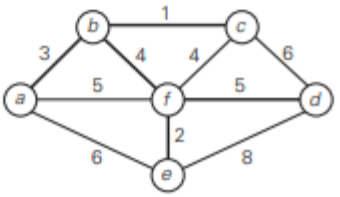
$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

Kruskal's Algorithm

- Kruskal's Algorithm is used to find the minimum cost spanning tree

Example : L-4.8: Kruskal Algorithm for Minimum Spanning Tree in Hindi | Algorithm



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de **1** 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 **2** 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 **3** 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 **4** 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 **5** 6 6 8	
df 5		

47 Kruskal's Algorithm to find Minimum Spanning Tree

ALGORITHM *Kruskal*(G)

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Huffman Trees and Codes

- Huffman coding is a technique used to construct a prefix-free binary tree, known as a Huffman tree or Huffman code tree, based on the frequency of symbols in the input text.
- The algorithm assigns shorter codewords to more frequent symbols and longer codewords to less frequent symbols, resulting in a more efficient encoding scheme.

Huffman's algorithm :

3.4 Huffman Coding - Greedy Method

OR

9.1 Huffman Coding -Greedy Method |Data Structures Tutorials

Step 1: Initialise n one-node trees

- Create a separate tree for each symbol in the alphabet, with each tree consisting of a single node.
- Record the frequency of each symbol in its respective tree's root.

Step 2: Repeat the following operation until a single tree is obtained

- Continuously merge the two trees with the smallest weights (frequencies) into a new tree.
- The new tree's weight is the sum of the weights of the two merged trees.
- Continue this process until only one tree remains, which will be the Huffman tree.

OR

1. Calculate the number of times each character appears in the string.
2. Sort the characters in increasing order of how often they appear.
3. Mark each unique character as a leaf node.
4. Create an empty node.
5. Add the character with the **lowest count of occurrences as the left child** of the empty node.
6. Add the character with the **second lowest count of occurrences as the right child** of the empty node.
7. Assign the sum of the two minimum frequencies to the empty node.
8. Repeat steps 2–6.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 9.12.

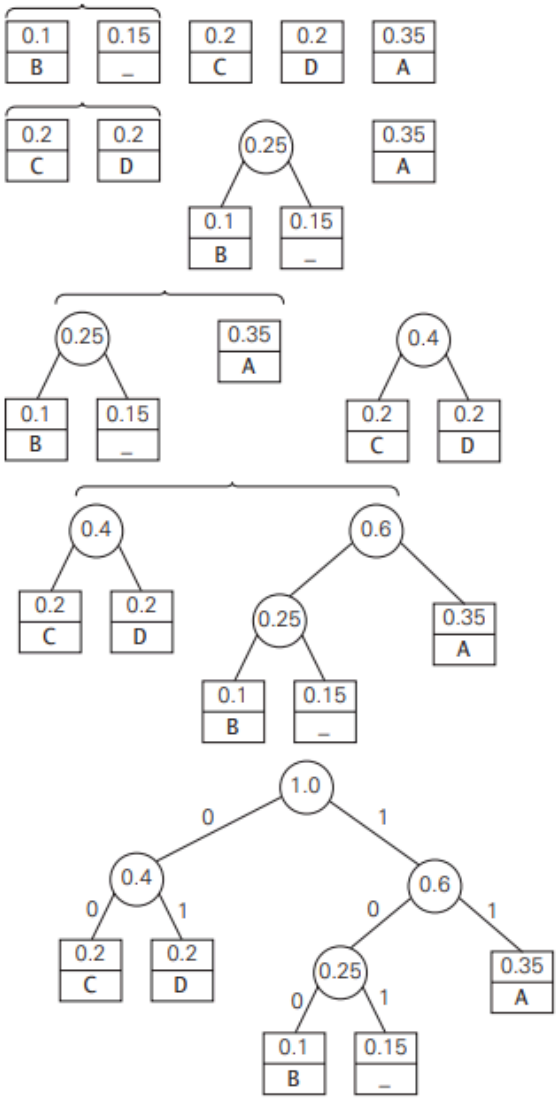


FIGURE 9.12 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

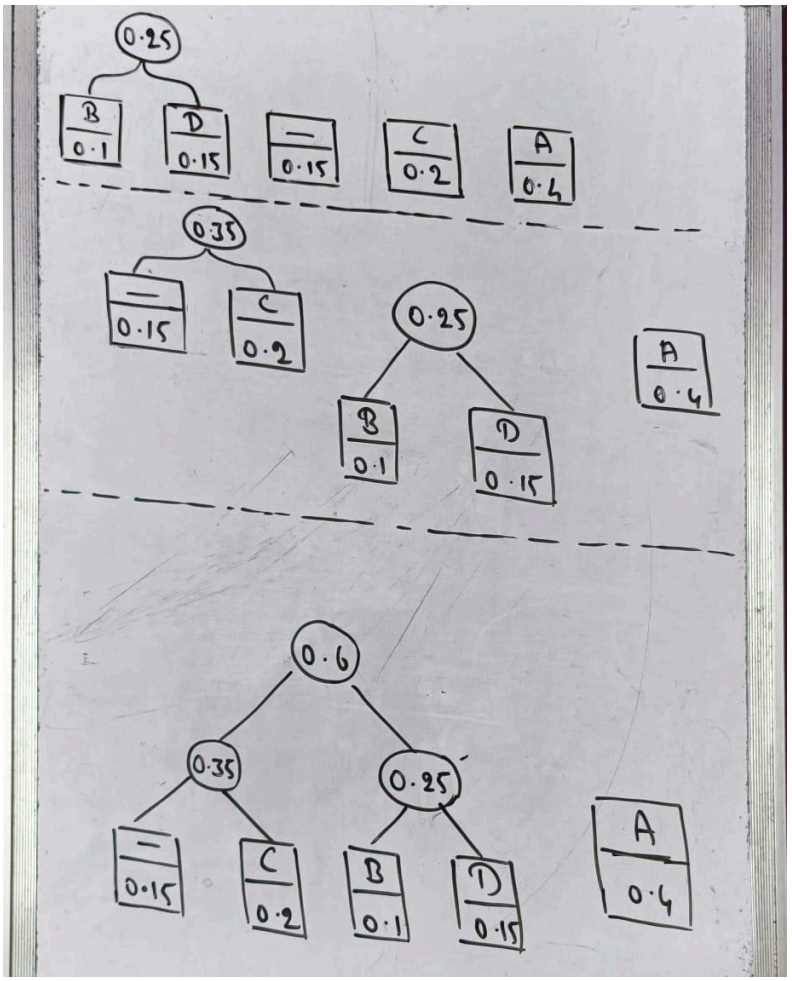
- Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.
- With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is 2 . 0.35 + 3 . 0.1 + 2 . 0.2 + 2 . 0.2 + 3 . 0.15 = 2.25.
- Had we used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per symbol. Thus, for this toy example, Huffman's code achieves the compression ratio—a standard measure of a compression algorithm's effectiveness—of(3 − 2.25)/3 . 100%= 25%.

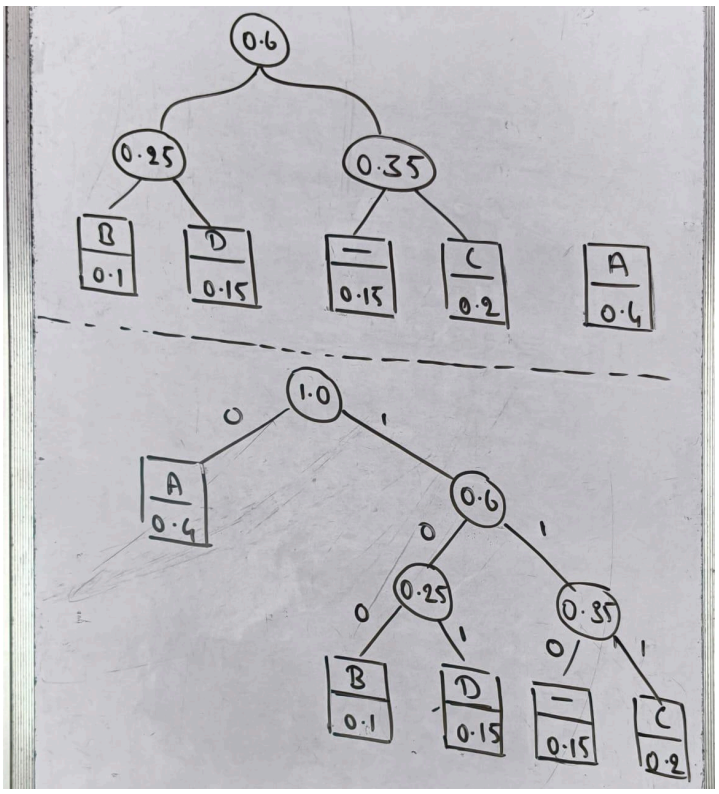
1. a. Construct a Huffman code for the following data:

symbol	A	B	C	D	_
frequency	0.4	0.1	0.2	0.15	0.15

b. Encode ABACABAD using the code of question (a).

c. Decode 100010111001010 using the code of question (a).





symbol	A	B	C	D	—
frequency	0.4	0.1	0.2	0.15	0.15
codeword	0	100	111	101	110

a) ABACABAD: 0 100 0 111 0 100 0 101

b) 100101111001010: BAD_ADA

Backtracking

General method

Backtracking is a systematic approach to solving problems by exploring all possible solutions incrementally. This is how it works:

1. **Construct Tree:** Represent the problem-solving process as a tree structure, where each node represents a partial solution or decision point.
2. **Explore Choices:** At each decision point, explore all possible choices or candidates for the next step.
3. **Evaluate Candidates:** Evaluate each candidate to determine if it satisfies problem constraints.
4. **Backtrack:** If a candidate leads to a dead end or violates constraints, backtrack to the previous decision point and try another option.
5. **Repeat:** Continue exploring and backtracking until a valid solution is found or all possibilities are exhausted.

n-Queens Problem : 53 N-Queen's Problem

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the **same row** or in the **same column** or on the **same diagonal**.

- For n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2 and n = 3.
- So let us consider the four-queens problem and solve it by the backtracking technique.
- Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in Figure

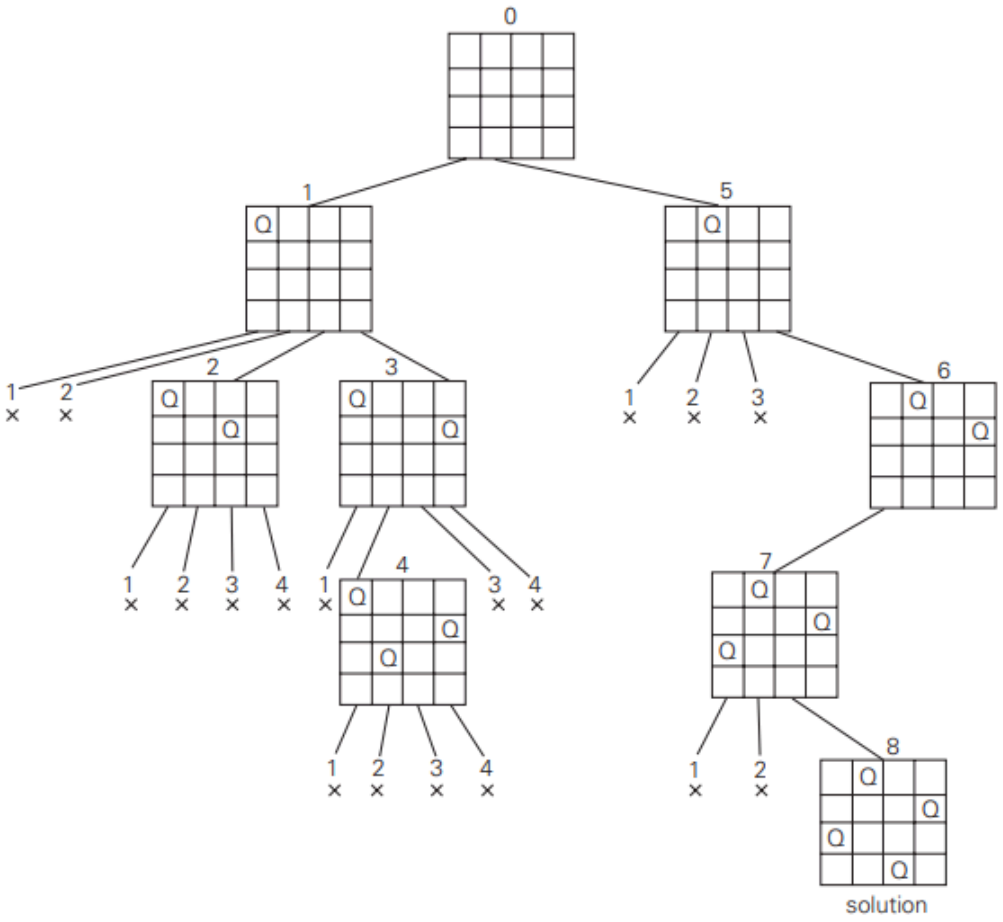


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

- We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.
- Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3.
- This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4).
- Then queen 3 is placed at (3, 2), which proves to be another dead end.
- The algorithm then backtracks all the way to queen 1 and moves it to (1, 2).
- Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in Figure 12.2.

Algorithm for n-Queens problem

Back Tracking Algorithm N Queen's Algorithm

```
ALGORITHM NQueens(k,n)
{
        for i = 1 to n do
        {
                if Place(k,i) = true do
                {
                        x[k] = i;
                        if k = n do
                        {
                                print(x)
                        }
                        else
                        {
                                NQueens(k+1,n)
                        }
                }
        }
}

ALGORITHM Place(k,i)
{
        for j = 1 to k-1 do
        {
                if i = x[j] or abs(x[j]-i) = abs(j-k)
                {
                        return false
                }
        }
        return true
}
```

Subset-Sum Problem

The subset-sum problem involves finding a subset of a given set of positive integers where the sum of the elements in the subset equals a given target sum. Here's a description:

Problem: Given a set A = {a1, a2, a3, … , an} of n positive integers and a target sum d, find a subset of A whose elements add up to d.

Example : For instance, consider A = {1,2,5,6,8} and d = 9. Two solutions exist {1, 2, 6} and {1, 8}.

Note : Set A must be in ascending order.

Example problem:

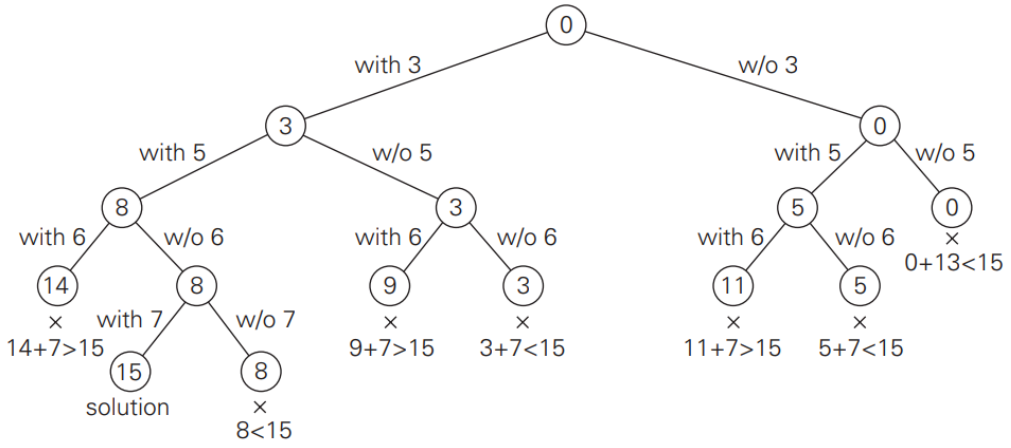


FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

Branch and bound

The branch-and-bound technique is a method used to systematically search for optimal solutions to optimization problems while pruning unpromising branches of the search space.

Here's a general explanation of how branch and bound works:

1. **Problem Definition:** Clearly define the optimization problem with an objective function to optimise and any constraints to satisfy.
2. **Initialization:** Set up initial variables, including the best solution found so far.
3. **State-Space Tree Construction:** Create a tree structure where each node represents a partial solution.
4. **Bound Calculation and Pruning:** Calculate bounds on the objective function for each node and prune branches that cannot lead to a better solution.
5. **Optimization and Termination:** Update the best solution if a node represents an optimal solution and continue exploring promising branches until termination conditions are met.b

Assignment Problem : 54 Assignment Problem using Branch & Bound

- The objective of the assignment problem is to assign ‘n’ jobs to ‘n’ persons such that the total cost of the assignment is as small as possible.

- Assignment problem is a **lower bound problem.**
- Lower bound is calculated as the **summation of the lesser/least value in each row.**

Example:

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

lb = 2 + 3 + 1 + 4

lb = 10

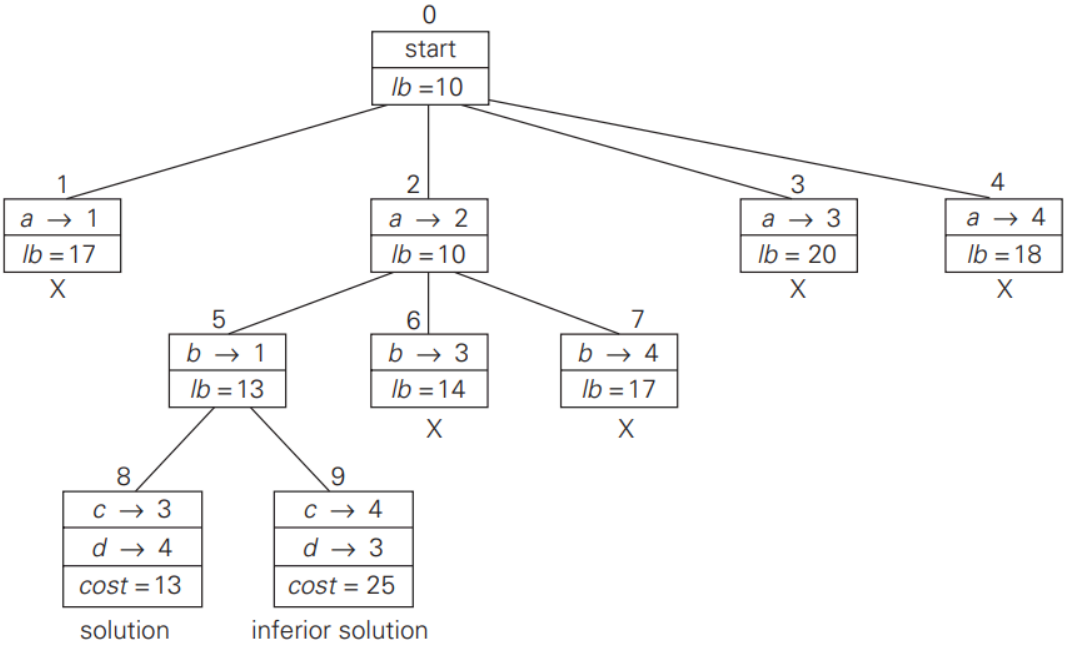
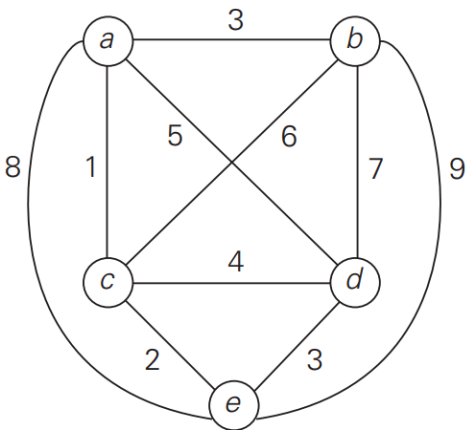


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Travelling Salesman Problem : 56 Travellig Salesman Problem using Branch & Bound

- The objective of the travelling salesman problem is to find a route such that a salesperson visits all the cities and returns back to the same city from where he started the tour with the **minimum cost**.
- TSP is a minimisation problem, hence we are supposed to find the lower bound.
- The lower bound(lb) is denoted as ceil(s/2)
    - s → is the sum of 2 closest cities of a given city, for all cities

Example:



$$lb = \frac{(1+3) + (3+6) + (1+2) + (3+4) + (2+3)}{2}$$

lb = 14 (initial)

Assumptions:

1. ‘a’ is the starting city
2. City ‘b’ is visited before ‘c’.

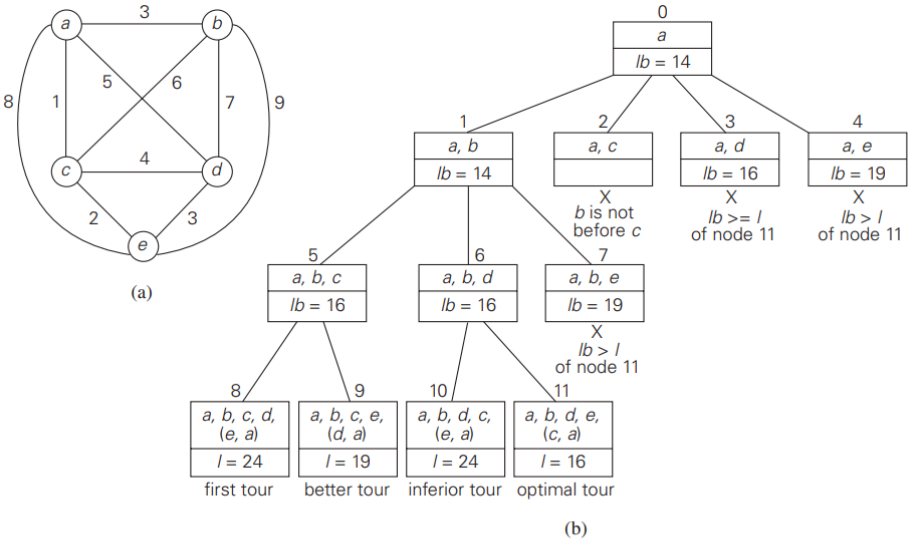


FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

Knapsack Problem : 55 Knapsack Problem using Branch & Bound

- Objective of the knapsack is to maximise the profit, hence it is an upper bound function.
- Here we should always order the items in the descending order of their profit to weight ratio.

I.e : $v1/w1 \geq v2/w2 \geq v3/w3 \geq \ldots vn/wn$

The upper bound is calculated as

$$Ub = v + (m-w)\frac{v_{i+1}}{w_{i+1}}$$

Here m is the maximum capacity of knapsack

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

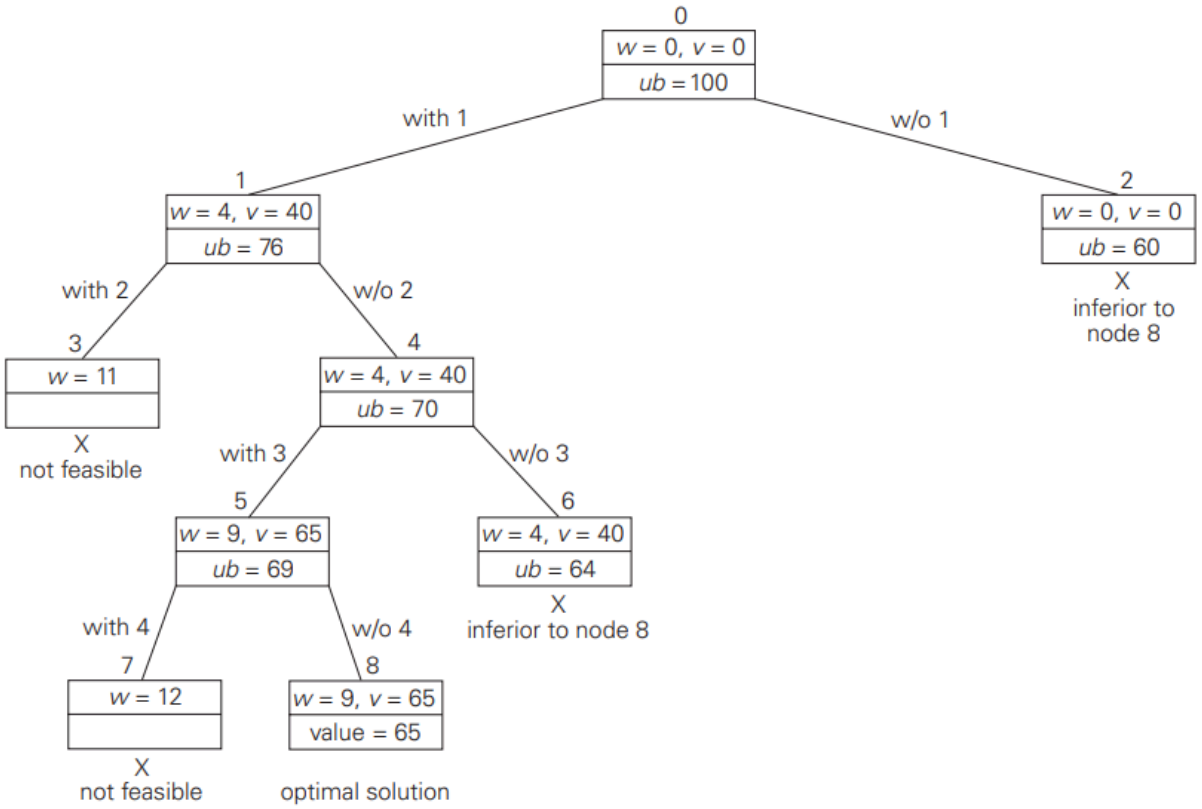


FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

ALGORITHMS

BRUTE FORCE

1. Selection sort . *Time complexity :* $\theta(n^2)$
2. Bubble sort
3. Sequential search
4. Brute-force String matching. *Time complexity : O(nm)*

DIVIDE AND CONQUER

1. Merge sort. *Time complexity :* θ*(n log n)*
2. Quick sort. *Time complexity :* θ*(n log n)*
3. Binary search. *Time complexity :* θ*(log n)*

DECREASE AND CONQUER

1. Insertion sort
2. Depth first Search
3. Breadth first Search
4. Topological sorting

TRANSFORM AND CONQUER

1. Heap (bottom up construction)

TIME AND SPACE TRADEOFFS

1. Comparison count sort
2. Distribution count sort
3. Horspool's algorithm

DYNAMIC PROGRAMMING

1. Floyd algorithm
2. Warshall Algorithm
3. The Knapsack problem

GREEDY TECHNIQUE

1. Bellman-ford algorithm
2. Single-source shortest path in DAG
3. Dijikstra's algorithm

MINIMUM SPANNING TREES

1. Prim’s algorithm
2. Kruskal’s algorithm
3. Optimal Tree problem : Huffman trees

BACK TRACKING

1. N-Queens problem.
2. Subset-sum problem

BRANCH AND BOUND

1. Assignment problem
2. Travelling salesman problem
3. Knapsack problem