Unit I

1

DESIGN STRATEGIES
ROLE OF ALGORITHM

Unit Structure

1.0 Objective

After going through this unit, you will be able to:

- What is an algorithm?
- What is the need of algorithm?
- What is the role of algorithm in computing?
- How to analyse algorithm?
- How to design an algorithm?
- What is Growth function?
- List of standard notation and common functions?

1.1 Introduction

In this chapter, you will learn what is an algorithm and its importance in computer technologies. It covers some suitable examples as well. Along with we will get the idea about growth function, standard notations and functions which are additional interesting ingredients of the algorithms.

1.2 The Role of Algorithms in Computing

1.2.1 Algorithms

Algorithm is step by step procedure which takes input value, process it or apply some computational techniques and produce some output value. In another way we can state the algorithm as a list of steps which transform the input to output using some processing function. We can also consider algorithm is a tool for solving some computational problems. Algorithm state the relationship of input and output.

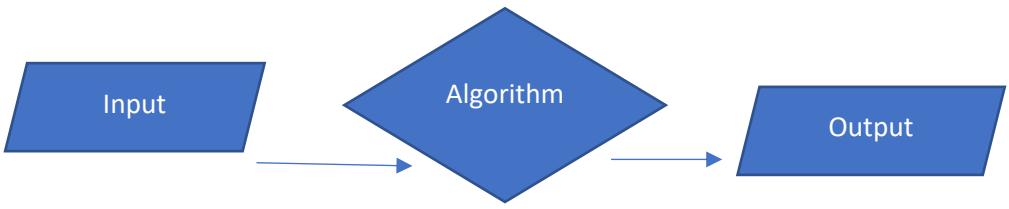


Fig1: Illustration of Algorithm

For example – if we need to sort the numbers in ascending order then it requires the formulate the problem as follows:

Input: A sequence of n numbers (n1,n2,n3,…,nn)

Output: (n1',n2',n3',…,nn') such that n1'<=n2'<=…<=nn')

Now we can take one example to solve this problem with numerical values.

So, the input is (5,2,4,6,1), and after sorting the output should be (1,2,4,5,6). Input sequence is called as **instance** of a problem.

An algorithm is to accurate if for every input it halts with the accurate output. We can use flowchart to represent the sequence of steps in pictorial form.

Characteristics of Algorithm

The Algorithm designed is language independent, i.e. they are just plain instructions set that can be implemented in any language, and the output will be the same, as expected.

Following are the key characteristics of algorithm -

- **Well-Defined Inputs**: If an algorithm says to take inputs instance, it should be clear that which type of inputs in required for processing.
- **Well-Defined Outputs:** The algorithm must clearly define that what type of output will be generated.
- **Finiteness:** The algorithm should not end up in an infinite loop.
- **Feasibility:** The algorithm must be generic and practical, such that it can be executed upon available resources.
- **Language Independent:** It must be just plain instructions set that can be implemented in any language, and yet the output will be same, as expected.

1.3 Algorithms as a technology

Consider the scenario, computers were tremendously fast and memory of computer was free as well. You would still like to prove that your solution method terminates with the correct answer. If computers were tremendously fast, any accurate method for solving a problem would give accurate result.

Every implementation should be in the proper bounded software engineering practice environment. But as a human being, we use to like to work on simple and easiest method to implement the solution for problem. As if we consider that computer is very fast but not infinite fast and memory also the integral part of scenario as memory costing is also very expensive.

So, computing time is a bounded resource, and so is space in memory. You should use these resources sensibly or as per your project need, and algorithms that are efficient in terms of time or space will help you do so.

Algorithms and other technologies

As algorithm should be effective and correct algorithm selection is the art and for implementation purpose, selection of hardware is also important part. As technological changes are happened at every single day and which help us to do our work more effectively. Algorithm is also get the similar importance as it save your reverse engineering or reframing the implementation of problem. Ultimately it helps for different technologies to take proper decision such as web technology, networking, wired and wireless network; etc.

1.4 Getting Started

1.3.1 Insertion sort

First algorithm is insertion sort. It is just like playing cards game as you can insert the card in between two cards and make the sequence. The logic is to arrange list in ascending order using insertion sort.

Input: A sequence of n numbers (n1,n2,n3,…,nn)

Output: (n1’,n2’,n3’,…,nn’) such that n1’<=n2’<=…<=nn’)

So, firstly we have our number in the form of array with size n and the numbers we need to sort can be called as **keys**. We can implement it in any language but in this chapter, we will illustrate it using C programming language. In insertion sort, procedure should be follow in following steps –

Pseudo code

```
Insertion_Sort(Arr)
    For j = 2 to Arr.length
    key =Arr[j]
    i = j-1
    while( i > 0 and Arr[i] > key)
        Arr[i+1] = Arr[i]
        i =i – 1
    Arr[i+1] = key
```

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

- Initialization: It is first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

For example –

22, 21, 23, 15, 16

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 21 is smaller than 22, move 22 and insert 21 before 22
21, 22, 23, 15, 16

i = 2. 23 will remain at its position as all elements in A[0..i-1] are smaller than 23
21, 22, 23, 15, 16

i = 3. 15 will move to the beginning and all other elements from 21 to 23 will move one position ahead of their current position.
15, 21, 22, 23, 16

i = 4. 16 will move to position after 15, and elements from 21 to 23 will move one position ahead of their current position. **15, 16, 21, 22, 23**

Program –

```
void insertionSort(int a1[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = a1[i];
      j = i - 1;
  /* Move elements of a1[0..i-1], that are greater than key, to one position ahead
        of their current position */
      while (j >= 0 && a1[j] > key) {
         a1[j + 1] = a1[j];
         j = j - 1;
      }
      a1[j + 1] = key;
   }
}
```

Pseudocode conventions

We use the following conventions in our pseudocode. Indentation indicates block structure. For example,

1. the body of the for loop that begins on line 1 consists of lines 2–8
2. body of the while loop that begins on line 5 contains lines 6–7 but not line 8.

Our indentation style applies to if-else statements2 as well. Using indentation instead of conventional indicators of block structure, such as begin and end

statements, greatly reduces clutter while preserving, or even enhancing, clarity. The looping constructs while, for, and repeat-until and the if-else conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.

1.3.2 Analyzing algorithms

Analysing an algorithm means predicting the required resources for the solving problem statement. Resources means computer hardware such as memory, bandwidth and communication channel. But the majorly computer hardware used for solving problem statement is primary concern, but most often it is computational time that we want to measure.

Generally, we used to do analysis of several algorithm and then pick up the efficient one. It is our practice to check whether we are using right algorithm or not. Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs.

Most of times, we shall assume a generic one processor, random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

So, following example will gives you the detail idea about how to analyse the problem and how the algorithm plays important role to solve the problem.

Analysis of insertion sort

Following program will gives you the idea about insertion sort.

```
void insertionSort(int a1[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = a1[i];
      j = i - 1;
  /* Move elements of a1[0..i-1], that are greater than key, to one position ahead
```

```
            of their current position */
          while (j >= 0 && a1[j] > key) {
            a1[j + 1] = a1[j];
            j = j - 1;
          }
          a1[j + 1] = key;
        }
      }
```

Observations are –

1. Its total time requires is based on number of inputs. (sorting 5 numbers will take less time as compare to sorting 50 numbers)
2. If the size of array or list is same but some number of the first list is sorted and another list is fully unsorted then first list will take less time as compare to whole unsorted list.
3. The time taken by an algorithm raises with the size of the input, so it is traditional to define the running time of a program as a function of the size of its input.
4. So, most important part of analysis is “running time” and “size of input”.
5. The running time of an algorithm on a particular input is the number of primitive processes or “steps” executed.
6. It is suitable to define the notion of step so that it is as machine-independent as possible.

Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```
for (i = n-1; i<=i; j++)
        if (a[j] > a[j+1])
swap a[j] and a[j+1];
```

Observations are -

1. This looks like the double. The innermost statement, the if, takes O(1) time. It doesn‟t necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here.

2. The outer loop executes n times, but the inner loop executes a number of times that depends on i. The first time the inner for executes, it runs i = n-1 times. The second time it runs n-2 times, etc. The total number of times the inner if statement executes is therefore:

(n-1) + (n-2) + ... + 3 + 2 + 1

This is the sum of an arithmetic series.

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

The value of the sum is n(n-1)/2. So the running time of bubble sort is O(n(n-1)/2), which is O((n2 -n)/2). Using the rules for big-O given earlier, this bound simplifies to O((n2)/2) by ignoring a smaller term, and to O(n2), by ignoring a constant factor. Thus, bubble sort is an O(n2) algorithm.

1.3.3 Designing algorithms

We can choose from a wide range of algorithm design techniques.

Design techniques are –

1. Brute force/ Exhaustive search
2. Divide and Conquer
3. Transformation
4. Dynamic programming
5. Greedy programming
6. Iterative improvement/ incremental approach
7. Randomization

For insertion sort, we used an incremental approach: having sorted the subarray A[1 .. j – 1], we inserted the single element A[j] into its proper place, yielding the sorted subarray A[1.. j].

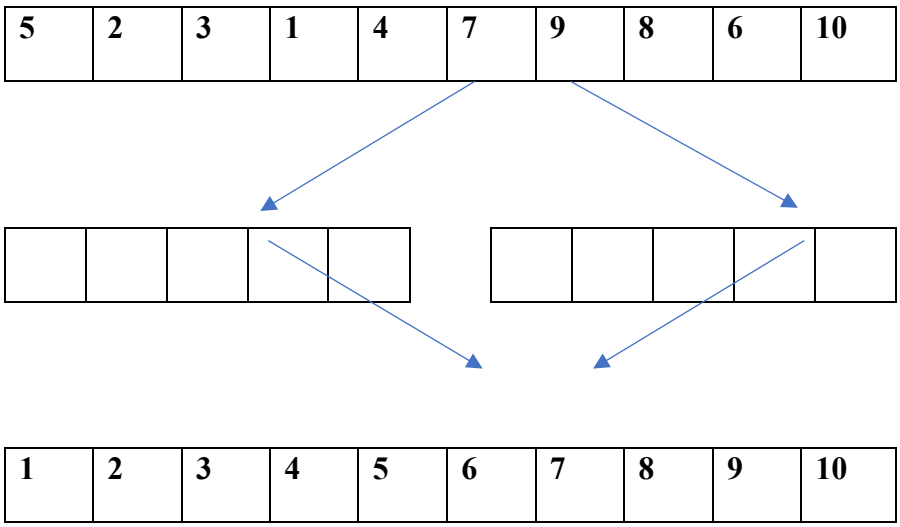
The divide and conquer approach

Many algorithm has different designing techniques like merge sort or quick sort has recursive structure. So, the observation is –

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

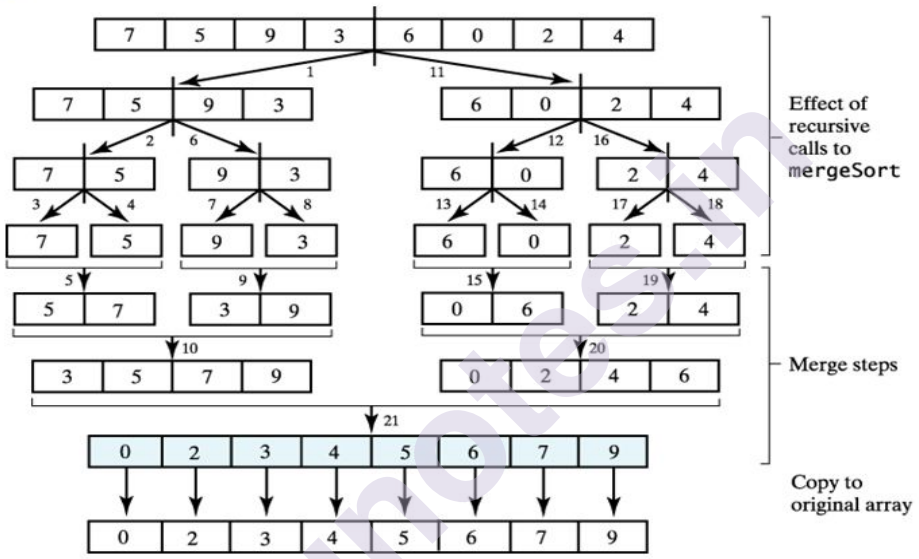
The divide-and-conquer paradigm involves three steps at each level of the recursion:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. **Combine** the solutions to the subproblems into the solution for the original problem.



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

- Divide: Divide the n-element sequence to be sorted into two sub-sequences of n=2 elements each.
- Conquer: Sort the two sub-sequences recursively using merge sort.
- Combine: Merge the two sorted sub-sequences to produce the sorted answer



1.5 Growth of Functions

Growth function gives a simple description of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Following steps need to consider –

1. Once the input size n becomes large enough, merge sort, with its ,θ(nlg n) worst-case running time, beats insertion sort, whose worst-case running time is $\theta(n^2)$.
2. The extra precision is not usually worth the effort of computing it.
3. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

4. When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms.

That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space necessity of the algorithm in terms of the size n of the input data. The storage space essential by an algorithm is simply a multiple of the data size n.

Complexity shall refer to the running time of the algorithm. The function f(n), gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data.

The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.

2. Average Case : The expected value of f(n).

3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms. Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

Asymptotic notation

The notations we use to define the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N=\{0,1,2,\ldots\}$.

Such notations are suitable for describing the worst-case running-time function T(n), which frequently is defined only on integer input sizes. We sometimes find it convenient, however, to abuse asymptotic notation in a variation of ways.

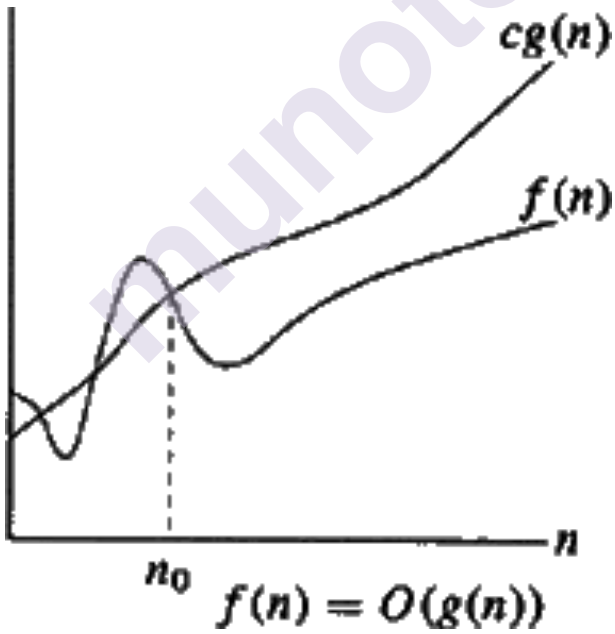
For example, we might extend the notation to the domain of real numbers or, alternatively, restrict it to a subset of the natural numbers. We should make sure, however, to understand the precise meaning of the notation so that when we abuse, we do not misuse it.

This section defines the basic asymptotic notations and also presents some common abuses. The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O)[1],
2. Big–OMEGA (Ω),
3. Big–THETA (θ) and
4. Little–OH (o)

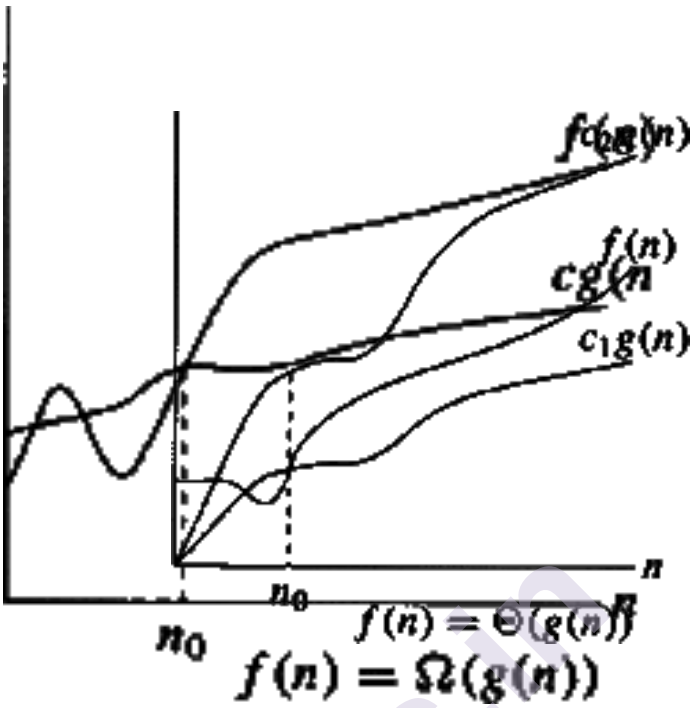
1. Big-OH (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (≤) that of g(n).



2. Big OMEGA Ω (Lower Bound)

f(n) = Ω(g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (≥) that of g(n).



In 1892, P. Bachmann invented a notation for characterizing the asymptotic behaviour of functions. His invention has come to be known as *big oh notation*.

Big-THETA *θ(Same order)*

f(n) = θ(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = Ω (g(n)].

4. Little-OH (o)

T(n) = o(p(n)) (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if T(n) = O(p(n)) and T(n) ≠ θ(p(n))].

1.4.2 Standard notations and common functions.

Monotonicity

1. A function $f(n)$ is ***monotonically increasing*** if $m \leq n$ implies $f(m) \leq f(n)$.
2. Similarly, it is ***monotonically decreasing*** if $m \leq n$ implies $f(m) \geq f(n)$.
3. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x").

For all real x,

(3.3) $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$.

For any integer n,

$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$,

and for any real number $n \geq 0$ and integers $a, b > 0$,

(3.4) $\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil$,

(3.5) $\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$,

(3.6) $\lceil a/b \rceil \leq (a + (b - 1))/b$,

(3.7) $\lfloor a/b \rfloor \geq ((a - (b - 1))/b$.

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

Modular arithmetic

For any integer a and any positive integer n, the value $a \bmod n$ is the ***remainder*** (or ***residue***) of the quotient a/n:

(3.8) $a \bmod n = a - \lfloor a/n \rfloor n$.

Given a precise notion of the remainder of one integer when divided by another, it is convenient to provide special notation to specify equality of remainders.

If $(a \bmod n) = (b \bmod n)$, we write $a \equiv b \pmod n$ and say that a is ***equivalent*** to b, modulo n. In other words, $a \equiv b \pmod n$ if a and b have the same remainder when divided by n. Equivalently, $a \equiv b \pmod n$ if and only if n is a divisor of $b - a$.

We write $a \not\equiv b \pmod n$ if a is not equivalent to b, modulo n.

Polynomials

Given a nonnegative integer d, a ***polynomial in n of degree d*** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

- where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$.
- For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$.
- For any real constant $a \geq 0$, the function n^a is monotonically increasing, and for any real constant $a \leq 0$, the function n^a is monotonically decreasing.

We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

Exponentials

For all real $a > 0$, m, and n, we have the following identities:

$a^0 = 1,$

$a^1 = a,$

$a^{-1} = 1/a,$

$(a^m)^n = a^{mn},$

$(a^m)^n = (a^n)^m,$

$a^m a^n = a^{m+n}.$

For all n and $a \geq 1$, the function a^n is monotonically increasing in n. When convenient, we shall assume $0^0 = 1$.

The rates of growth of polynomials and exponentials can be related by the following fact. For all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 , \tag{3.9}$$

from which we can conclude that
$n^b = o(a^n)$.

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

Using e to denote 2.71828..., the base of the natural logarithm function, we have for all real x,

(3.10) $$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} ,$$

where "!" denotes the factorial function defined later in this section. For all real x, we have the inequality

(3.11) $$e^x \geq 1 + x ,$$

where equality holds only when $x = 0$. When $|x| \leq 1$, we have the approximation

(3.12) $$1 + x \leq e^x \leq 1 + x + x^2 .$$

When $x \to 0$, the approximation of e^x by $1 + x$ is quite good:

$e^x = 1 + x + \Theta(x^2)$.

(In this equation, the asymptotic notation is used to describe the limiting behavior as $x \to 0$ rather than as $x \to \infty$.) We have for all x,

(3.13) $$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x .$$

Logarithms

We shall use the following notations:

$\lg n = \log_2 n$ (binary logarithm) ,

$\ln n = \log_e n$ (natural logarithm) ,

$\lg^k n = (\lg n)^k$ (exponentiation) ,

$\lg \lg n = \lg(\lg n)$ (composition) .

An important notational convention we shall adopt is that *logarithm functions will apply only to the next term in the formula*, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and n,

$$
\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b},
\end{aligned}
\tag{3.14}
$$

$$
\begin{aligned}
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a},
\end{aligned}
\tag{3.15}
$$

where, in each equation above, logarithm bases are not 1.

By equation (3.14), changing the base of a logarithm from one constant to another only changes the value of the logarithm by a constant factor, and so we shall often use the notation "lg n" when we don't care about constant factors, such as in O-notation. Computer scientists find 2 to be the most natural base for logarithms because so many algorithms and data structures involve splitting a problem into two parts.

There is a simple series expansion for $\ln(1 + x)$ when $|x| < 1$:

$$
\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .
$$

We also have the following inequalities for $x > -1$:

$$
\frac{x}{1+x} \le \ln(1+x) \le x ,
\tag{3.16}
$$

where equality holds only for $x = 0$.

We say that a function $f(n)$ is ***polylogarithmically bounded*** if $f(n) = O(\lg^k n)$ for some constant k. We can relate the growth of polynomials and polylogarithms by substituting $\lg n$ for n and 2^a for a in equation (3.9), yielding

$$
\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 .
$$

From this limit, we can conclude that

$\lg^b n = o(n^a)$

for any constant $a > 0$. Thus, any positive polynomial function grows faster than any polylogarithmic function.

Factorials

The notation $n!$ (read "n factorial") is defined for integers $n \geq 0$ as

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

Thus, $n! = 1 \cdot 2 \cdot 3\ n$.

A weak upper bound on the factorial function is $n! \leq n^n$, since each of the n terms in the factorial product is at most n. ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.17}$$

where e is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well.

$$\begin{aligned} n! &= o(n^n) , \\ n! &= \omega(2^n) , \\ \lg(n!) &= \Theta(n \lg n) , \end{aligned} \tag{3.18}$$

where Stirling's approximation is helpful in proving equation (3.18). The following equation also holds for all $n \geq 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.19}$$

where

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} . \tag{3.20}$$

Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value of n. Formally, let $f(n)$ be a function over the reals. For nonnegative integers i, we recursively define

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

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

The iterated logarithm function

We use the notation lg* n (read "log star of n") to denote the iterated logarithm, which is defined as follows. Let $\lg^{(i)} n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$. Be sure to distinguish $\lg^{(i)} n$ (the logarithm function applied i times in succession, starting with argument n) from $\lg^i n$ (the logarithm of n raised to the ith power). The iterated logarithm function is defined as

$$\lg^* n = \min \{i = 0: \lg^{(i)} n \leq 1\}.$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

Since the number of atoms in the observable universe is estimated to be about 10^{80}, which is much less than 2^{65536}, we rarely encounter an input size n such that $\lg^* n > 5$.

Fibonacci numbers

The ***Fibonacci numbers*** are defined by the following recurrence:

$$(3.21)\quad \begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2. \end{aligned}$$

Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Fibonacci numbers are related to the ***golden ratio*** φ and to its conjugate $\hat{\varphi}$, which are given by the following formulas:

(3.22)
$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots, \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots. \end{aligned}$$

Specifically, we have

(3.23)
$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}},$$

which can be proved by induction . Since $|\widehat{\phi}| < 1$, we have $|\widehat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, so that the ith Fibonacci number F_i is equal to $\phi^i/\sqrt{5}$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially.

1.6 Let us Sum Up

In this chapter, we have learn the concept of Algorithm, characteristics of algorithm, asymptotic notation and standard functions. As per given in the "Introduction to Algorithm" book by CORMEN's, all authors define the asymptotic notations in the different way, although the various descriptions settle in most common situations. Some of the substitute definitions include functions that are not asymptotically nonnegative, as long as their absolute values are appropriately bounded.

1.7 List of References

1. Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, PHI Learning Pvt. Ltd-New Delhi (2009).

2. https://www.javatpoint.com/algorithms-and-functions

3. Dr. L. V. N. Prasad, Professor, lecture notes on Design And Analysis Of Algorithms

4. http://serverbob.3x.ro/IA/DDU0020.html#!/stealingyourhistory

1.8 Bibliography

Exercises

1. Calculate the analysis of simple for loop

```
for (i = 1; i<=n; i++)
    v[i] = v[i] + 1;
```

2. Calculate analysis of matrix multiply

```
for (i = 1; i<=n; i++)
    for (j = 1; j<=n; j++)
    C[i, j] = 0;
    for (k = 1; k<=n; k++)
    C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

3. Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in 64n lg n steps. For which values of n does insertion sort beat merge sort?

4. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Unit I

2

DESIGN STRATEGIES DIVIDE AND CONQUER AND RANDOMIZED ALGORITHM

Unit Structure

2.0 Objective

After going through this unit, you will be able to:

- What is divide and conquer concept?
- What is the need of Probabilistic algorithm?
- What is Randomized Algorithms?

2.1 Introduction

In this chapter, we will learn more algorithms based on divide-and-conquer. The first one solves the maximum-subarray problem: it is takings as input an array of numbers, and it determines the contiguous subarray whose values have the greatest sum. Then we shall see two divide-and-conquer algorithms for multiplying n x n matrices. We can also learn Strassen's algorithm, probabilistic Analysis and randomised algorithm.

2.2 Divide-and-Conquer

Divide and conquer is a design algorithm technique which is well known to breaking down efficiency barriers.

Divide and conquer algorithm consists of three parts:

Divide: Divide the problem into sub problems. The sub problems are solved recursively.

Conquer: The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

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

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not.

Divide–and–conquer is a very powerful use of recursion. When the subproblems are large enough to solve recursively, we call that the recursive case. Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the base case. Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem. We consider solving such subproblems as part of the combine step.

Recurrences

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms. A recurrence is an equation or inequality that describes a function in

terms of its value on smaller inputs. For example, the worst-case running time T(n) of the MERGE-SORT procedure by the recurrence.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases}$$

whose solution we claimed to be

$T(n) = \Theta(n \lg n)$.

There are 3 methods to solving recurrences

1. In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.
2. The recursion-tree method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
3. The master method provides bounds for recurrences of the form. It is used to determine the running times of the divide-and-conquer algorithms for the maximum-subarray problem and for matrix multiplication.

2.2.1 The maximum-subarray problem

As we can consider problem of maximum subarray sum. The problem of maximum subarray sum is basically finding the part of an array whose elements has the largest sum. If all the elements in an array are positive then it is easy, find the sum of all the elements of the array and it has the largest sum over any other subarrays you can make out from that array.

For example – 2,1,4,3,5

Maximum sum = 2+1+4+3+5 = 15

But the problem gets more interesting when some of the elements are negative then the subarray whose sum of the elements is largest over the sum of the elements of any other subarrays of that element can lie anywhere in the array.

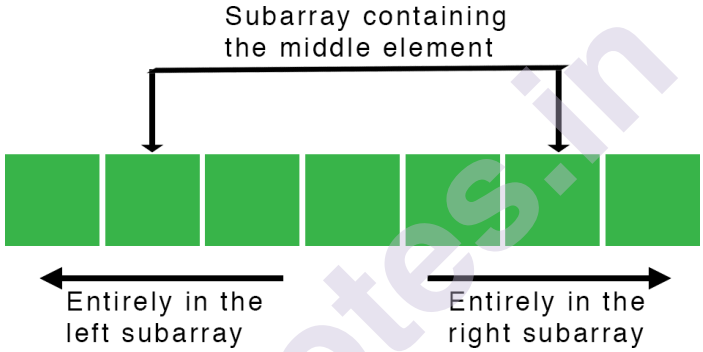
For example – 2,-1,4,-3,5

Maximum sum = 2-1+4-3+5 = 7

Brute force solution

The Brute Force technique to solve the problem is simple. Just iterate through every element of the array and check the sum of all the subarrays that can be made starting from that element i.e., check all the subarrays and this can be done in ${}^{n}C_{2}$ ways i.e., choosing two different elements of the array to make a subarray. Thus, the brute force technique is of Θ(n2) time. However, we can solve this in Θ(nlog(n)) time using divide and conquer.

As we know that the divide and conquer solves a problem by breaking into subproblems, so let's first break an array into two parts. Now, the subarray with maximum sum can either lie entirely in the left subarray or entirely in the right subarray or in a subarray consisting both i.e., crossing the middle element.



The first two cases where the subarray is entirely on right or on the left are actually the smaller instances of the original problem. So, we can solve them recursively by calling the function to calculate the maximum sum subarray on both the parts.

```
Max_sum_subarray(array, low, high)
{
 if (high == low) // only one element in an array
 {
  return array[high]
 }
 mid = (high+low)/2
 Max_sum_subarray(array, low, mid)
 Max_sum_subarray(array, mid+1, high)
}
```

We are making Max_sum_subarray is a function to calculate the maximum sum of the subarray in an array. Here, we are calling the function Max_sum_subarray for both the left and the right subarrays i.e., recursively checking the left and the right subarrays for the maximum sum subarray.

Now, we have to handle the third case i.e., when the subarray with the maximum sum contains both the right and the left subarrays (containing the middle element). At a glance, this could look like a smaller instance of the original problem also but it is not because it contains a restriction that the subarray must contain the middle element and thus makes our problem much more narrow and less time taking.

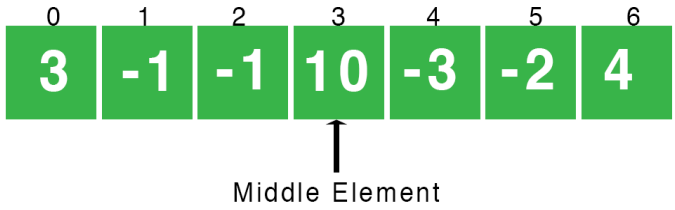
So, we will now make a function called Max_crossing_subarray to calculate the maximum sum of the subarray crossing the middle element and then call it inside the Max_sum_subarray function.

```
max_sum_subarray(array, low, high)
{
  if (high == low) // only one element in an array
  {
    return array[high];
  }
  mid = (high+low)/2;
  maximum_sum_left_subarray = max_sum_subarray(array, low, mid)
  maximum_sum_right_subarray = max_sum_subarray(array, mid+1, high)
  maximum_sum_crossing_subarray = max_crossing_subarray(array, low, mid, high);
  // returning the maximum among the above three numbers
  return maximum(maximum_sum_left_subarray, maximum_sum_right_subarray, maximum_sum_crossing_subarray);
}
```

Here, we are covering all three cases mentioned above to and then just returning the maximum of these three.

Now, let's write the max_crossing_subarray function.

The max_crossing_subarray function is simple, we just have to iterate over the right and the left sides of the middle element and find the maximum sum.



Left subarray
sum = 0
Start from index 3:
10 > 0 => sum = 10
10-1=9<sum(10)
9-1=8<sum(10)
8+3=11>sum => sum = 11

Right subarray
sum = 0
Start from index 4:
-3 < sum(0), sum is 0
-3-2=-5<sum(0), sum is 0
-5+4=-1<sum(0), sum is 0

Final crossing sum = 11+0=0 from index 0 to 3

```
max_crossing_subarray(int ar[], int low, int mid, int high)
{
 left_sum = -infinity
 sum = 0
 for (i=mid downto low)
 {
  sum = sum+ar[i]
  if (sum>left_sum)
    left_sum = sum
 }
 right_sum = -infinity;
 sum = 0
 for (i=mid+1 to high)
```

```
 {
   sum=sum+ar[i]
   if (sum>right_sum)
     right_sum = sum
 }
 return (left_sum+right_sum)
}
```

Here, our first loop is iterating from the middle element to the lowest element of the left subarray to find the maximum sum and similarly the second loop is iterating from the middle+1 element to the highest element of the subarray to calculate the maximum sum of the subarray on the right side. And finally, we are returning summing both of them and returning the sum which is calculated from the subarray crossing the middle element.

2.2.2 Strassen's algorithm for matrix multiplication

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969). The usual way to multiply two n x n matrices A and B, yielding result matrix C as follows :

```
for i := 1 to n do
  for j :=1 to n do
        c[i, j] := 0;
        for K: = 1 to n do
              c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n3 scalar multiplication"s (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon. We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then Cij can be found by the usual matrix multiplication algorithm,

$$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$$
$$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$$
$$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$$
$$C_{22} = A_{21} . B_{12} + A_{22} . B_{22}$$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$$T(1) = 1$$
$$T(n) = 8\ T(n/2)$$

Which leads to T (n) = O (n^3), where n is the power of 2. Strassens insight was to find an alternative method for calculating the Cij, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$$P = (A_{11} + A_{22}) (B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22}) B_{11}$$
$$R = A_{11} (B_{12} - B_{22})$$
$$S = A_{22} (B_{21} - B_{11})$$
$$T = (A_{11} + A_{12}) B_{22}$$
$$U = (A_{21} - A_{11}) (B_{11} + B_{12})$$
$$V = (A_{12} - A_{22}) (B_{21} + B_{22})$$
$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U.$$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$T(1) = 1$$
$$T(n) = 7\,T(n/2)$$

Solving this for the case of $n = 2^k$ is easy:

$$T(2^k) = 7\,T(2^{k-1})$$
$$= 7^2\,T(2^{k-2})$$
$$= \text{- - - - - -}$$
$$= \text{- - - - - -}$$
$$= 7^i\,T(2^{k-i})$$

Put i = k

$$= 7^k\,T(1)$$
$$= 7^k$$

That is, $T(n) = 7^{\log_2 n}$

$$= n^{\log_2 7}$$
$$= O(n^{\log_2 7}) = O(n^{2.81})$$

So, concluding that Strassen"s algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until „n" revolves the hundreds.

2.2.3 The substitution method for solving recurrences

One way to solve a divide-and-conquer recurrence equation is to use the iterative substitution method. This is a "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we than substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge sort recurrence equation yields the equation.

$$T(n) = 2\,(2\,T(n/2^2) + b\,(n/2)) + b\,n$$

$$= 2^2\,T(n/2^2) + 2\,b\,n$$

Plugging the general equation for T again yields the equation.

$$T(n) = 2^2\,(2\,T(n/2^3) + b\,(n/2^2)) + 2\,b\,n$$

$$= 2^3\,T(n/2^3) + 3\,b\,n$$

The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side). In the case of merge-sort recurrence equation, the general form is:

$$T(n) = 2^i T(n/2^i) + i\, b\, n$$

Note that the general form of this equation shifts to the base case, T(n) = b, where $n = 2^i$, that is, when i = log n, which implies:

$$T(n) = b\, n + b\, n \log n.$$

In other words, T(n) is O(n log n). In a general application of the iterative substitution technique, we hope that we can determine a general pattern for T(n) and that we can also figure out when the general form of T(n) shifts to the base case.

2.3 Probabilistic Analysis and Randomized Algorithms

Probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or a computational problem. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm.

2.3.1 The hiring problem

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency sends you one candidate each day. You interview that person and then decide either to hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a substantial hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be. The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode. It assumes that the candidates for the office

assistant job are numbered 1 through n. The procedure assumes that you are able to, after interviewing candidate i, determine whether candidate i is the best candidate you have seen so far. To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

```
HIRE-ASSISTANT(n)
    best =0 // candidate 0 is a least-qualified dummy candidate
    for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
    best = i
    hire candidate i
```

We focus not on the running time of HIRE-ASSISTANT, but instead on the costs incurred by interviewing and hiring. On the surface, analyzing the cost of this algorithm may seem very different from analyzing the running time of, say, merge sort. The analytical techniques used, however, are identical whether we are analyzing cost or running time. In either case, we are counting the number of times certain basic operations are executed.

Interviewing has a low cost, say ci, whereas hiring is expensive, costing ch. Letting m be the number of people hired, the total cost associated with this algorithm is O(cin+chm). No matter how many people we hire, we always interview n candidates and thus always incur the cost cin associated with interviewing. We therefore concentrate on analyzing chm, the hiring cost. This quantity varies with each run of the algorithm. This scenario serves as a model for a common computational paradigm. We often need to find the maximum or minimum value in a sequence by examining each element of the sequence and maintaining a current "winner." The hiring problem models how often we update our notion of which element is currently winning.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview. This situation occurs if the candidates come in strictly increasing order of quality, in which case we hire n times, for a total hiring cost of O(Chn). Of course, the candidates do not always come in increasing order of quality. In fact, we have no idea about the order

in which they arrive, nor do we have any control over this order. Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm. Sometimes we use it to analyze other quantities, such as the hiring cost in procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs. Thus we are, in effect, averaging the running time over all possible inputs. When reporting such a running time, we will refer to it as the average-case running time. We must be very careful in deciding on the distribution of inputs. For some problems, we may reasonably assume something about the set of all possible inputs, and then we can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, we cannot describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis. For the hiring problem, we can assume that the applicants come in a random order. What does that mean for this problem? We assume that we can compare any two candidates and decide which one is better qualified; that is, there is a total order on the candidates. Thus, we can rank each candidate with a unique number from 1 through n, using rank(i) to denote the rank of applicant i, and adopt the convention that a higher rank corresponds to a better qualified applicant. The ordered list (rank(1),rank(2),…rank(n)) is a permutation of the list (1,2,…,n). Saying that the applicants come in a random order is equivalent to saying that this list of ranks is equally likely to be any one of the n! permutations of the numbers 1 through n. Alternatively, we say that the ranks form a uniform random permutation; that is, each of the possible n! permutations appears with equal probability.

Randomized algorithms

In order to use probabilistic analysis, we need to know something about the distribution of the inputs. In many cases, we know very little about the input distribution. Even if we do know something about the distribution, we may not be able to model this knowledge computationally. Yet we often can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random. In the hiring problem, it may seem as if the candidates are being presented to us in a random order, but we have no way of knowing whether or not they really are. Thus, in order to develop a randomized

algorithm for the hiring problem, we must have greater control over the order in which we interview the candidates. We will, therefore, change the model slightly. We say that the employment agency has n candidates, and they send us a list of the candidates in advance. On each day, we choose, randomly, which candidate to interview. Although we know nothing about the candidates (besides their names), we have made a significant change. Instead of relying on a guess that the candidates come to us in a random order, we have instead gained control of the process and enforced a random order. More generally, we call an algorithm randomized if its behavior is determined not only by its input but also by values produced by a random-number generator. We shall assume that we have at our disposal a random-number generator RANDOM. A call to RANDOM(a,b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM(0,1) produces 0 with probability 1/2, and it produces 1 with probability 1/2. A call to RANDOM(3,7) returns either 3, 4, 5, 6, or 7, each with probability 1/5. Each integer returned by RANDOM is independent of the integers returned on previous calls. You may imagine RANDOM as rolling a(b-a+1) -sided die to obtain its output. (In practice, most programming environments offer a pseudorandom-number generator: a deterministic algorithm returning numbers that "look" statistically random.) When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator. We distinguish these algorithms from those in which the input is random by referring to the running time of a randomized algorithm as an expected running time. In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm, and we discuss the expected running time when the algorithm itself makes random choices.

2.3.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we use indicator random variables. Indicator random variables provide a convenient method for converting between probabilities and expectations. Suppose we are given a sample space S and an event A. Then the indicator random variable I {A} associated with event A is defined as

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs}, \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

As a simple example, let us determine the expected number of heads that we obtain when flipping a fair coin. Our sample space is S = {H, T}, with Pr{H} = Pr{T}=1/2.

We can then define an indicator random variable XH , associated with the coin coming up heads, which is the event H. This variable counts the number of heads obtained in this flip, and it is 1 if the coin comes up heads and 0 otherwise. We write

$$\begin{aligned} X_H &= \mathrm{I}\{H\} \\ &= \begin{cases} 1 & \text{if } H \text{ occurs}, \\ 0 & \text{if } T \text{ occurs}. \end{cases} \end{aligned}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable XH :

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 . \end{aligned}$$

Thus the expected number of heads obtained by one flip of a fair coin is 1/2. As the following lemma shows, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

2.3.3 Randomized algorithms

In the previous section, we showed how knowing a distribution on the inputs can help us to analyze the average-case behavior of an algorithm. Many times, we do not have such knowledge, thus precluding an average-case analysis. As mentioned in Section 5.1, we may be able to use a randomized algorithm. For a problem such as the hiring problem, in which it is helpful to assume that all permutations of the input are equally likely, a probabilistic analysis can guide the development of a randomized algorithm. Instead of assuming a distribution of inputs, we impose a distribution. In particular, before running the algorithm, we randomly permute the candidates in order to enforce the property that every permutation is equally likely. Although we have modified the algorithm, we still expect to hire a new office assistant approximately ln n times. But now we expect this to be the case for any input, rather than for inputs drawn from a particular distribution.

Let us further explore the distinction between probabilistic analysis and randomized algorithms. In Section 5.2, we claimed that, assuming that the candidates arrive in a random order, the expected number of times we hire a new office assistant is about ln n. Note that the algorithm here is deterministic; for any particular input, the number of times a new office assistant is hired is always the

same. Furthermore, the number of times we hire a new office assistant differs for different inputs, and it depends on the ranks of the various candidates. Since this number depends only on the ranks of the candidates, we can represent a particular input by listing, in order, the ranks of the candidates, i.e., ((rank(1), rank(2),…rank(n)). Given the rank list A1 = (1,2,3,4,5,6,7,8,910), a new office assistant is always hired 10 times, since each successive candidate is better than the previous one, and lines 5–6 are executed in each iteration. Given the list of ranks A2 =(10,9,8,7,6,5,4,3,2,1), a new office assistant is hired only once, in the first iteration. Given a list of ranks A3=(5, 2, 1, 8, 4, 7, 10, 9, 3, 6), a new office assistant is hired three times, upon interviewing the candidates with ranks 5, 8, and 10. Recalling that the cost of our algorithm depends on how many times we hire a new office assistant, we see that there are expensive inputs such as A1, inexpensive inputs such as A2, and moderately expensive inputs such as A3. Consider, on the other hand, the randomized algorithm that first permutes the candidates and then determines the best candidate. In this case, we randomize in the algorithm, not in the input distribution. Given a particular input, say A3 above, we cannot say how many times the maximum is updated, because this quantity differs with each run of the algorithm. The first time we run the algorithm on A3, it may produce the permutation A1 and perform 10 updates; but the second time we run the algorithm, we may produce the permutation A2 and perform only one update. The third time we run it, we may perform some other number of updates. Each time we run the algorithm, the execution depends on the random choices made and is likely to differ from the previous execution of the algorithm. For this algorithm and many other randomized algorithms, no particular input elicits its worst-case behavior. Even your worst enemy cannot produce a bad input array, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an “unlucky” permutation. For the hiring problem, the only change needed in the code is to randomly permute the array.

```
RANDOMIZED-HIRE-ASSISTANT(n)
randomly permute the list of candidates
best = 0 // candidate 0 is a least-qualified dummy candidate
        for i = 1 to n
        interview candidate i
        if candidate i is better than candidate best
                best  = i
                hire candidate i
```

With this simple change, we have created a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

2.4 Let us Sum Up

In this chapter we have learned various probabilistic Analysis and Randomised algorithms.

2.5 List of references

1. https://www.codesdope.com/blog/article/maximum-subarray-sum-using-divide-and-conquer/

2. Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, PHI Learning Pvt. Ltd-New Delhi (2009).

3. https://www.javatpoint.com/algorithms-and-functions

4. Dr. L. V. N. Prasad, Professor, lecture notes on Design And Analysis Of Algorithms

2.6 Exercise

1. Explain the concept of divide and conquer.

2. Explain the idea of Strassen's algorithm for matrix multiplication.

3. Describe the concept of randomized algorithm. Write down algorithm for the same.

Unit II

3

ADVANCED DESIGN AND ANALYSIS TECHNIQUES

ROLE OF ALGORITHM

Unit Structure

3.1 Objective

After going through this unit, you will be able to:

- What is dynamic programming?
- What is the need of greedy algorithm?
- What is the elementary graph algorithm?
- How to solve the single source shortest path problems?

3.2 Introduction

In this section, we will learn dynamic programming, greedy algorithms, graph algorithms, representation of graph, spanning tree and single source shortest path.

3.3 Dynamic Programming

We typically apply dynamic programming to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information.

3.3.1 Rod cutting

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length | 1 2 3 4 5 6 7 8

---

price | 1 5 8 9 10 17 17 20

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length | 1 2 3 4 5 6 7 8

---

price | 3 5 8 9 10 17 17 20

A naive solution for this problem is to generate all configurations of different pieces and find the highest priced configuration. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.
Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as following.
cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

2) Overlapping Subproblems

Following is simple recursive implementation of the Rod Cutting problem. The implementation simply follows the recursive structure.

3.3.2 Elements of dynamic programming

There are basically three elements that characterize a dynamic programming algorithm:-

1. **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.

2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are

reused many times, and we do not want to repeatedly solve the same problem over and over again.

3. **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Bottom-up means:-

i. Start with smallest subproblems.

ii. Combining their solutions obtain the solution to sub-problems of increasing size.

iii. Until solving at the solution of the original problem.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

3.3.3 Longest common subsequence

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence S = <s_1, s_2, s_3, s_4, …,s_n>.

A sequence Z = <z_1, z_2, z_3, z_4, …,z_m> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X*** and ***Y***.

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

Algorithm

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

LCS-LENGTH takes two sequences X = (x1, x2,…, xm) and Y = (y1, y2,…,yn) as inputs. It stores the c[i, j] values in a table c[0..m, 0..n] and it computes the entries in row-major order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table b[1..m,1..n] to help us construct an optimal solution. Intuitively, b[i, j] points

to the table entry corresponding to the optimal subproblem solution chosen when computing c[i, j]. The procedure returns the b and c tables; c[m,n] contains the length of an LCS of X and Y.

3.4Greedy algorithm

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions.

3.4.1 An activity-selection problem

The problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Suppose we have a set S = {a1, a2,…an} of n proposed **activities** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity ai has **a start time si and a finish time fi** , where 0 si < fi < ∞. If selected, activity ai takes place during the half-open time interval (si,fi). Activities ai and aj are compatible if the intervals (si,fi) and (si,fi) do not overlap. That is, ai and aj are compatible if si >= fj or sj >= fi. In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

f1<= f2 <= f3 <= fn1 <= fn

i	1	2	3	4	5	6	7	8	9	10	11
si	1	3	0	5	3	5	6	8	8	2	12
fi	4	5	6	7	9	9	10	11	12	14	16

For this example, the subset {a3, a9, a11} consists of mutually compatible activities. It is not a maximum subset, however, since the subset {a1, a4, a8, a11} is larger. In fact, {a1, a4, a8, a11} is a largest subset of mutually compatible activities; another largest subset is {a2, a4, a9, a11}.

3.4.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Elements are as follows -

Greedy-choice property The first key ingredient is the greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. We usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms.

The **0-1 knapsack problem** is the following. A thief robbing a store finds n items. The ith item is worth i dollars and weighs wi pounds, where i and wi are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

In the **fractional knapsack problem**, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

3.3.3 Huffman codes

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

That is, only 6 different characters appear, and the character a occurs 45,000 times. We have many options for how to represent such a file of information. Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a codeword. If we use a fixed-length code, we need 3 bits to represent 6 characters: a = 000, b = 001, ..., f = 101. This method requires 300,000 bits to code the entire file. Can we do better? A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure shows such a code; here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f.

This code requires

(45x1) + (13x3) + (12x3) + (16x3) + (9x4) + (5x4) x 1,000 = 224,000 bits.

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file.

3.5 Elementary Graph Algorithms

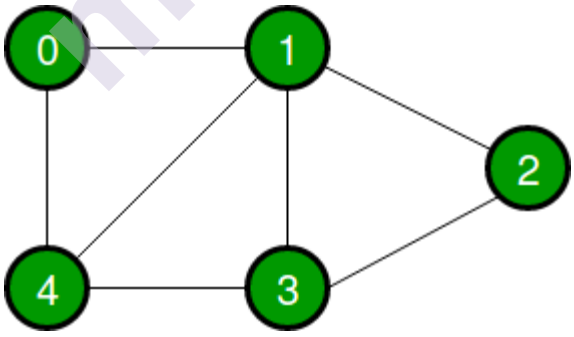
In this topic presents methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at the heart of the field of graph algorithms.

3.5.1 Representations of graphs

A graph is a data structure that consists of the following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See this for more applications of graph.

Following is an example of an undirected graph with 5 vertices.



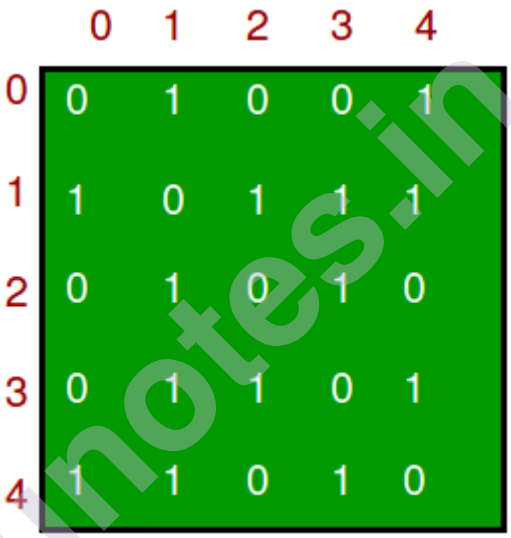
The following two are the most commonly used representations of a graph.
1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

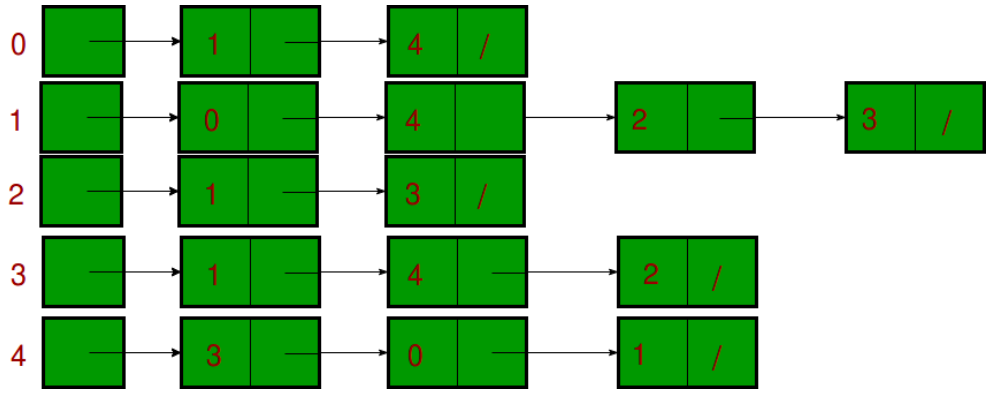


Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time. Please see this for a sample Python implementation of adjacency matrix.

Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

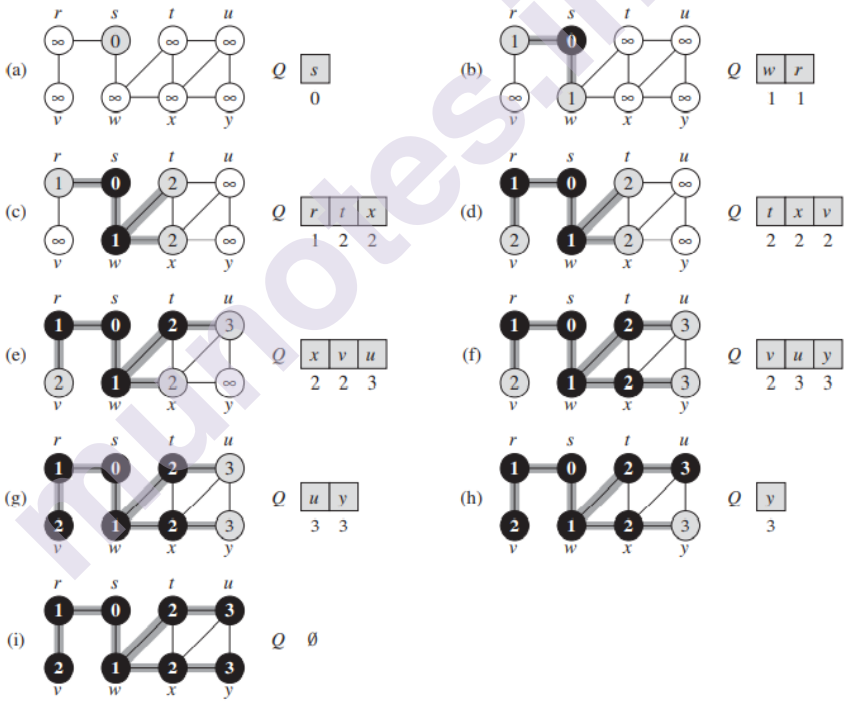


3.5.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas similar to those in breadth-first search. Given a graph G = (V; E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex reachable from s, the simple path in the breadth-first tree from s to corresponds to a "shortest path" from s to in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

```
BFS(G, s)
        for each vertex u ∈ G.V -  {s}
                u.color = WHITE
                u.d = ∞
                u.π =  NIL
        s.color = GRAY
        s.d - 0
        s.π = NIL
        Q = φ
        ENQUEUE(Q,s)
                while Q ≠ φ
```

```
u = DEQUEUE(Q)
for each v ∈ G.Adj[u]
        if v.color = = WHITE
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                ENQUEUE(Q,v)
u.color = BLACK
```



The procedure BFS works as follows. With the exception of the source vertex s, lines 1–4 paint every vertex white, set u:d to be infinity for each vertex u, and set the parent of every vertex to be NIL. Line 5 paints s gray, since we consider it to be discovered as the procedure begins. Line 6 initializes s.d to 0, and line 7 sets the predecessor of the source to be NIL. Lines 8–9 initialize Q to the queue containing just the vertex s. The while loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.

3.5.3 Depth-first search

As in breadth-first search, whenever depth-first search discovers a vertex during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute v.π to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources. Therefore, we define the predecessor subgraph of a depth-first search slightly differently from that of a breadth-first search:

$G_\pi = (V, E_\pi)$, where

$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$.

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in E are tree edges. As in breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

3.6 Minimum Spanning Trees

Minimum spanning tree. An edge-weighted graph is a graph where we associate weights or costs with each edge. A minimum spanning tree (MST) of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree. Assumptions. The graph is connected.

3.6.1 Growing a minimum spanning tree

Assume that we have a connected, undirected graph G = (V, E) with a weight function w : E -> R, and we wish to find a minimum spanning tree for G. The two algorithms we consider in this chapter use a greedy approach to the problem, although they differ in how they apply this approach. This greedy strategy is captured by the following generic method, which grows the minimum spanning tree one edge at a time. The generic method manages a set of edges A, maintaining the following loop invariant: Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we determine an edge (u,v) that we can add to A without violating this invariant, in the sense that AU {(u,v)} is also a subset of a minimum spanning tree.

We call such an edge a safe edge for A, since we can add it safely to A while maintaining the invariant.


```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

We use the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

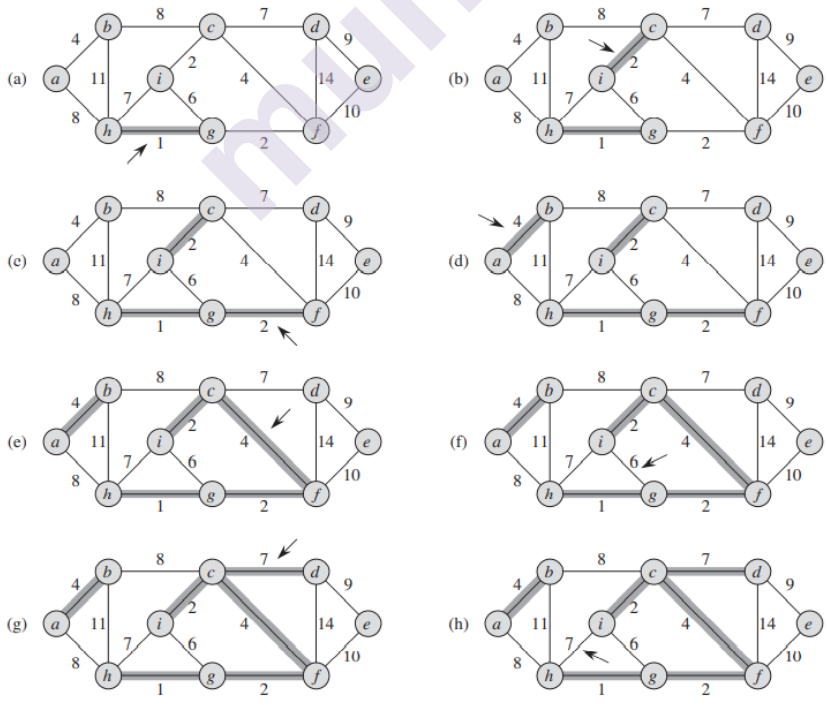
3.6.2 Algorithms of Kruskal and Prim

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight. Let

C1 and C2 denote the two trees that are connected by (u,v). Since (u,v) must be a light edge connecting C1 to some other tree, Corollary 23.2 implies that (u,v) is a safe edge for C1. Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight. Our implementation of Kruskal's algorithm is like the algorithm to compute connected components. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest.

The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v) To combine trees, Kruskal's algorithm calls the UNION procedure.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```



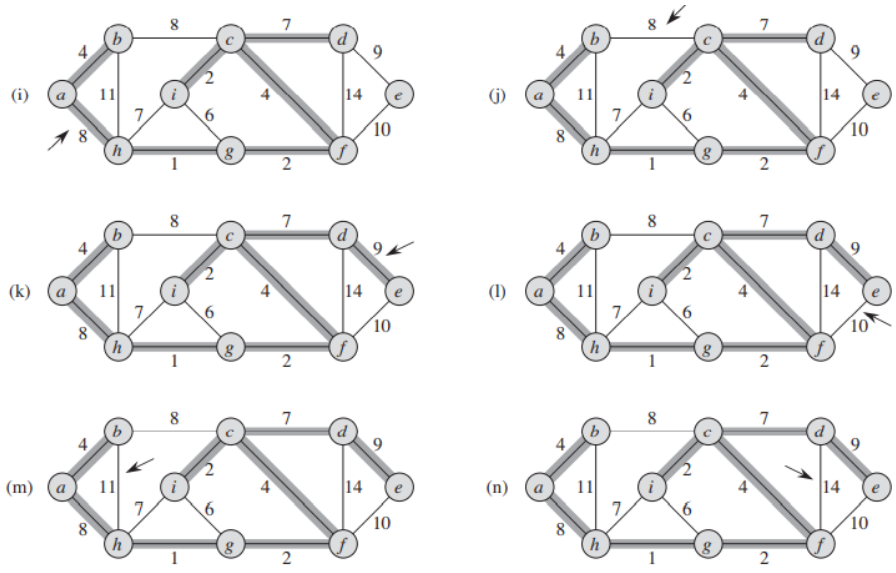


Figure shows how Kruskal's algorithm works. Lines 1–3 initialize the set A to the empty set and create |V| trees, one containing each vertex. The for loop in lines 5–8 examines edges in order of weight, from lowest to highest.

The loop checks, for each edge (u,v) whether the endpoints u and belong to the same tree. If they do, then the edge (u,v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, line 7 adds the edge (u,v) to A, and line 8 merges the vertices in the two trees.

Prim's algorithm

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

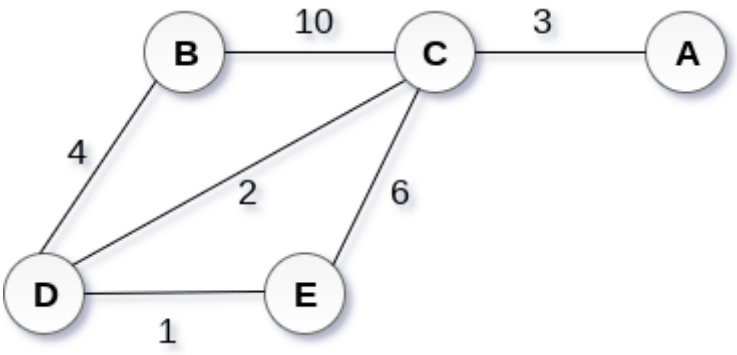
Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T [END OF LOOP]
- **Step 5:** EXIT

Example –

Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



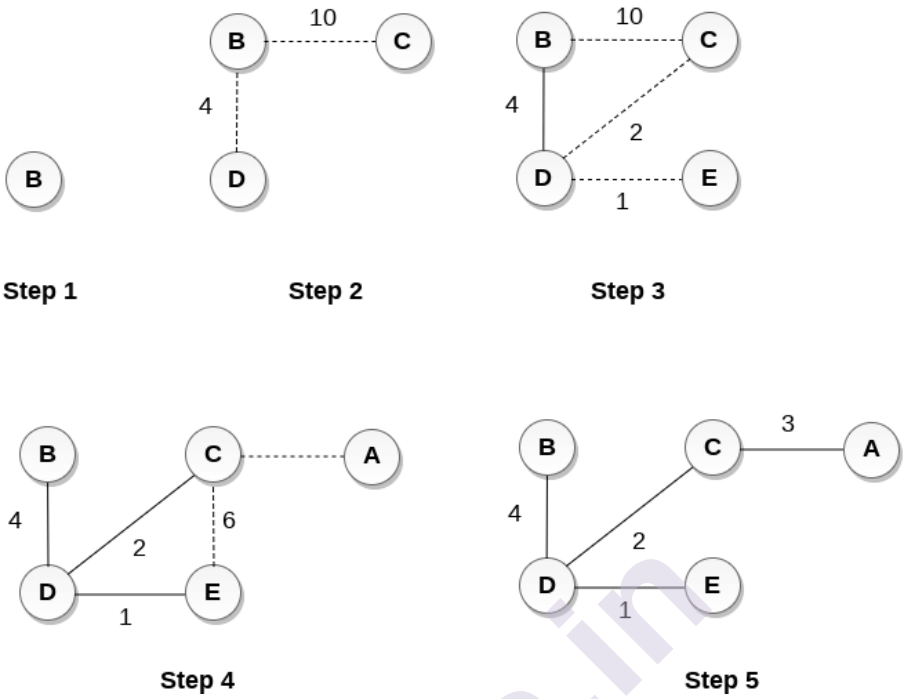
Solution –

- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
- **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
- **Step 3:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
- **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

The cost of MST will be calculated as;

cost(MST) = 4 + 2 + 1 + 3 = 10 units.



3.7 Single-Source Shortest Paths

The single source shortest path algorithm (for arbitrary weight positive or negative) is also known Bellman-Ford algorithm is used to find minimum distance from source vertex to any other vertex. The single-destination shortest path problem, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.

3.7.1 The Bellman-Ford algorithm

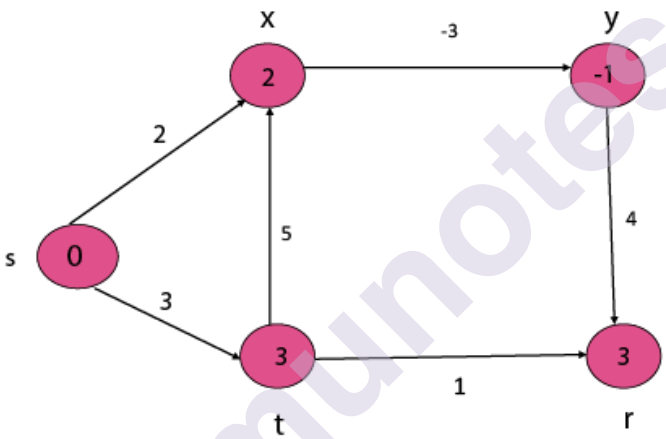
The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function w : E -> R, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights. The algorithm relaxes edges, progressively decreasing an estimate :d on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight (s,v). The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
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
```

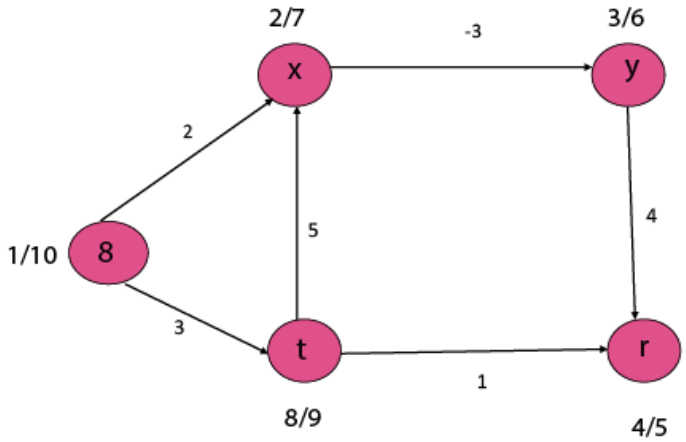
3.7.2 Single-source shortest paths in directed acyclic graphs

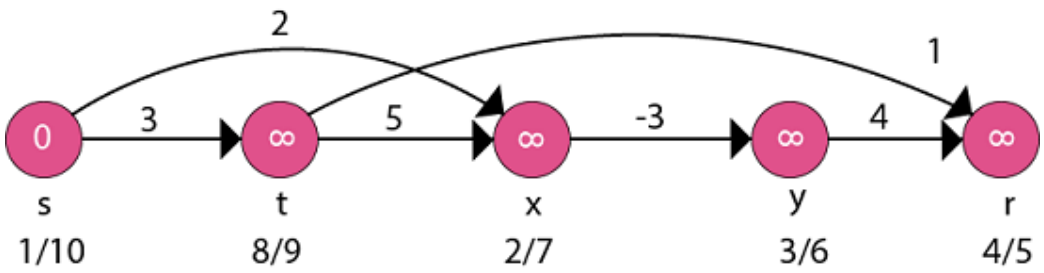
By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, we can figure out shortest paths from a single source in ∅(V+E) time. Shortest paths are always well described in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

Example –



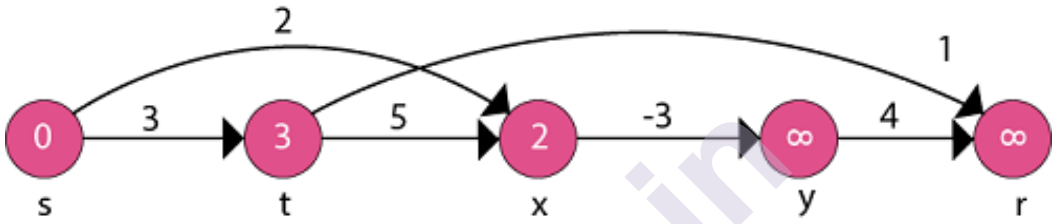
Step1: To topologically sort vertices apply **DFS (Depth First Search)** and then arrange vertices in linear order by decreasing order of finish time.



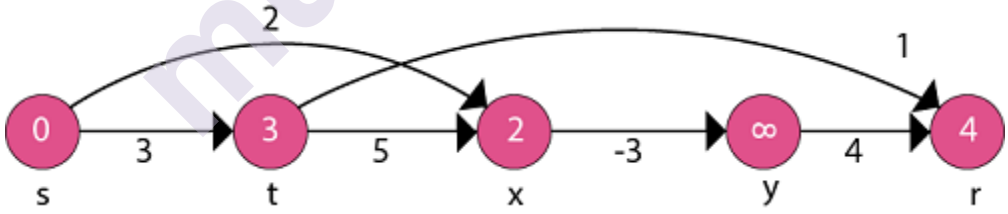


Initialize Single Source

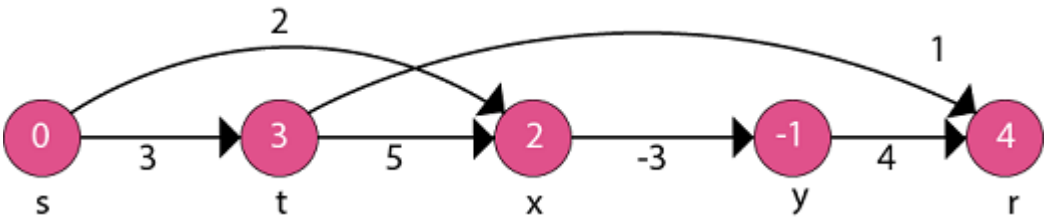
Now, take each vertex in topologically sorted order and relax each edge.



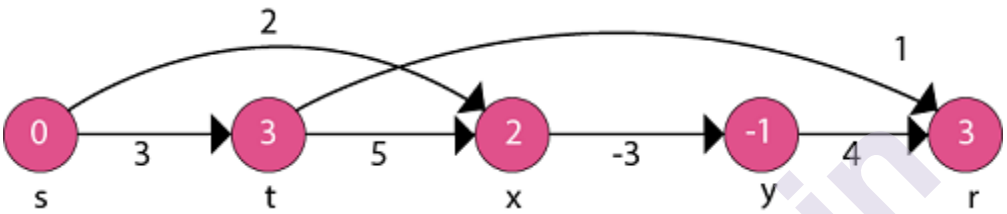
1. adj [s] →t, x
2. 0 + 3 < ∞
3. d [t] ← 3
4. 0 + 2 < ∞
5. d [x] ← 2



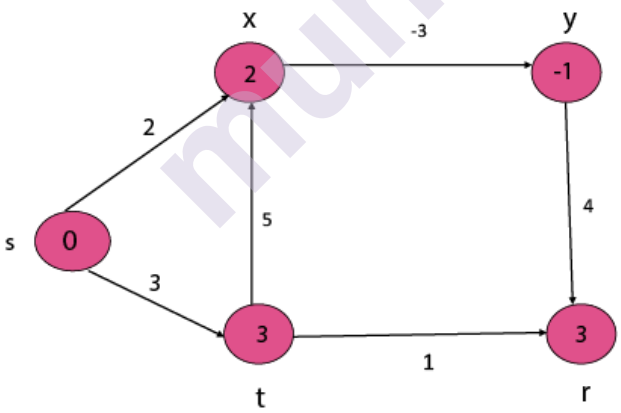
1. adj [t] → r, x
2. 3 + 1 < ∞
3. d [r] ← 4
4. 3 + 5 ≤ 2



1. adj [x] → y
2. 2 - 3 < ∞
3. d [y] ← -1



1. adj [y] → r
2. -1 + 4 < 4
3. 3 <4
4. d [r] ← 3



Thus the Shortest Path is:

1. s to x is 2
2. s to y is -1
3. s to t is 3
4. s to r is 3

3.7.3 Dijkstra's algorithm

4 Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

5 Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph. Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

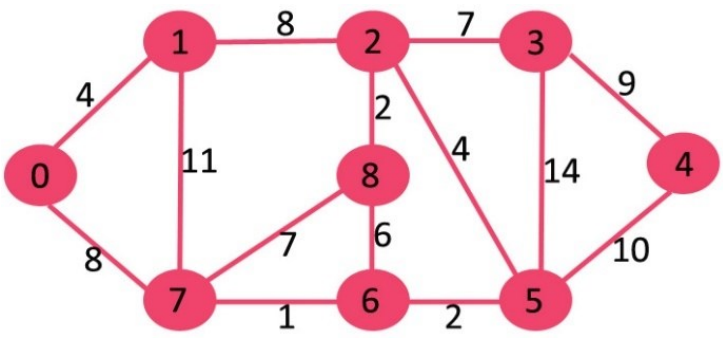
3) While *sptSet* doesn't include all vertices

a) Pick a vertex u which is not there in *sptSet* and has minimum distance value.

b) Include u to *sptSet*.

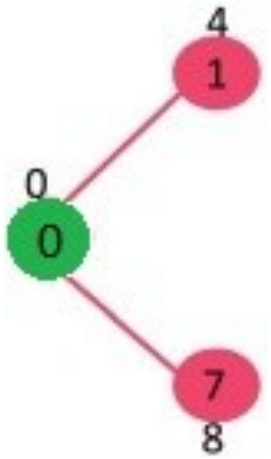
c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

6 Let us understand with the following example:

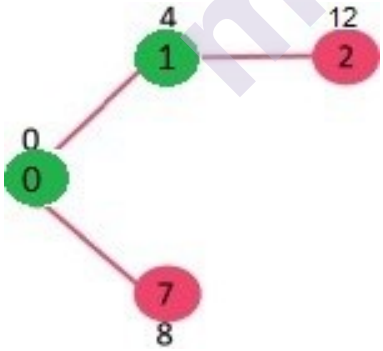


7 The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

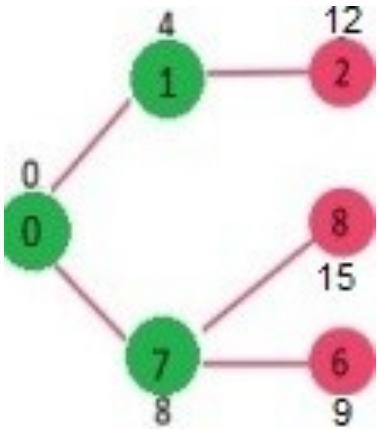
8



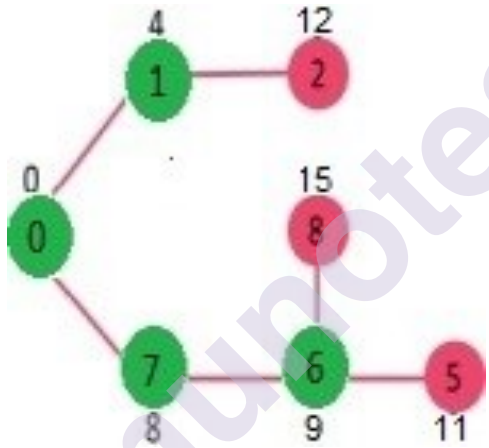
9 Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



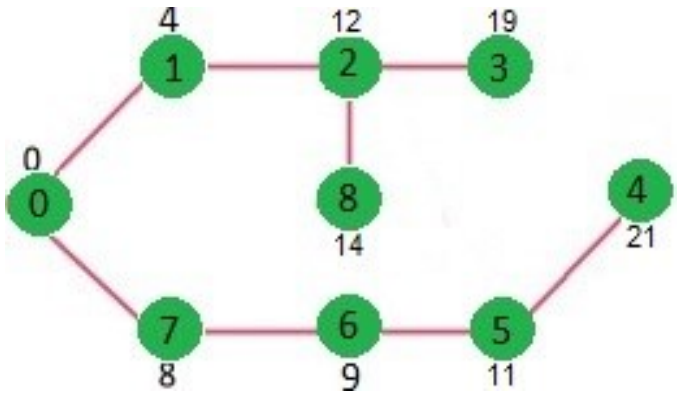
10 Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



11 Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



12 We repeat the above steps until *sptSet* does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



3.8 Let us Sum Up

In this section, we have studied dynamic programming, greedy approach, graph algorithm, spanning tree and single source shortest path algorithm precisely.

3.9 List of References

1. https://www.javatpoint.com/prim-algorithm

2. Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, PHI Learning Pvt. Ltd-New Delhi (2009).

3.10 Exercises

1. Determine an LCS of (1, 0, 0, 1, 0, 1, 0, 1) and (0, 1, 0, 1, 1, 0, 1, 1, 0).

2. Prove that the fractional knapsack problem has the greedy-choice property.

3. Show how to solve the fractional knapsack problem in O(n) time.

4. Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Unit III

4

NUMBER-THEORETIC ALGORITHMS AND NP – COMPLETENESS

Unit Structure: -

4.0 Objective

Algorithms are heavily based on Number Theory. So many day to day problems can be solved with simple numerical methods. We are trying to include such materials in this topic.

4.1 Introduction

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in large part to the invention of cryptographic schemes based on large prime numbers. These schemes are feasible because we can find large primes easily, and they are secure because we do not know how to factor the product of large primes (or solve related problems, such as computing discrete logarithms) efficiently. This chapter presents some of the number theory and related algorithms that underlie such applications

4.2 Number Theory

Number Theory is widely used in Computer science as well as Mathematics. Many algorithms and techniques are dependent on Number Theory. Mostly Number theory is used in cryptographic study.

4.3 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set of integers and the set of natural numbers.

4.3.1 Divisibility and divisors

Let us consider set Z= {…., - 2, -1,0,1, 2….} of Integers and Set N = {0, 1, 2 ….}, the divisibility and divisors are defined as follows:

The notation d | a (d divides a) means a = kd for some integer k. d is called as divisor of a.

4.3.2 Prime and composite numbers

Let us consider set Z= {…., - 2, -1,0,1, 2….} of Integers and Set N = {0, 1, 2 ….}, Prime and composite numbers are defined as follows:

An integer a > 1 whose only divisors are the trivial divisors 1 and a is a prime number or, more simply, a prime. Primes have many special properties and play a critical role in number theory. The first 20 primes, in order, are

2, 3, 5, 7 ,11, 13, 17, 19; 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71

if integer a > 1 is not prime then it is a composite number.

4.4 Greatest common divisor

In this section, we describe Euclid's algorithm for efficiently computing the greatest common divisor of two integers. When we analyze the running time, we shall see a surprising connection with the Fibonacci numbers, which yield a worst-case input for Euclid's algorithm.

3.4.1 Euclid's algorithm

Euclid's algorithm is recursive program to find out GCD of a non-negative number. In principle, we can compute gcd(a,b), for positive integers a and b from the prime factorizations of a and b. The inputs a and b are arbitrary non-negative numbers.

```
EUCLID (a,b)
        if(b = = 0)
                return a
        else return EUCLID(b, a mod b)
```

Consider example of calculating GCD(30,21)

```
EUCLID (30, 21)
EUCLID (30, 21)   =   EUCLID (21, 30 mod 21)
                  =   EUCLID (21, 9)
EUCLID (21, 9)    =   EUCLID (21, 21 mod 9)
                  =   EUCLID (9, 3)
EUCLID (9, 3)     =   EUCLID (3, 9 mod 3)
                  =   EUCLID (3, 0ddd)
```

The worst case time complexity is calculated with the sum of a + b. As all positive integers have common divisor as 1, time complexity will have an upper bound of O(a+b).

4.5 Modular Arithmetic

Modular Arithmetic is system of integers where we are considering the remainders. This concept is highly used in cryptographic services. Given an integer x > 1, defined as modulus, two integers a and b are said to be congruent modulo x, if x is a divisor of their difference (i.e., if there is an integer x such that a − b = kx).

Congruence modulo x is a congruence relation, meaning that it is an equivalence relation that is compatible with the operations of addition, subtraction, and multiplication. Congruence modulo n is denoted:

A === b mod x.

Solving modular linear equations:-

We now consider the problem of finding solutions to the equation

$$ax === b \pmod{n}$$

Where a > 0 and n > 0. This problem has several applications like RSA public-key cryptosystem. Assuming a, b, and n are given, and aim to find all values of x, modulo n, that satisfy equation The equation may have zero, one, or more than one such solution.

```
MODULAR-LINEAR-EQUATION-SOLVER (a,b,n)
        (d, x',y') = EXTENDED-EUCLID (a , n ) ……x0 = x'(b/d)
        mod n
        if d | b
        for i = 0 to d-1
                print (x0 + i( n | d) ) mod n
        else print "no solutions"
```

MODULAR-LINEAR-EQUATION-SOLVER performs O (lg n + gcd (a, n)) arithmetic operations, since EXTENDED-EUCLID performs O(lg n) arithmetic operations, and each iteration of the for loop of lines 4–5 performs a constant number of arithmetic operations.

4.6 Chinese Remainder Theorem.

Let us consider set of pairwise relatively prime integers $m_1, m_2, \ldots, m_r$. Then the system of simultaneous congruence's

$$x \equiv a_1 \pmod{m_1}$$
$$x \equiv a_2 \pmod{m_2}$$
$$\vdots$$
$$x \equiv a_r \pmod{m_r}$$

The above equation has a unique solution modulo $M = m_1, m_2, \cdots m_r$, for any given integers $a_1, a_2, \ldots, a_r$.

Proof of CRT.

Put $M = m_1 \cdots m_r$ and for each $k = 1, 2, \ldots, r$ let $Mk = M/m_k$.

Then $\gcd(M_k, m_k) = 1$ for all k.

Let y_k be an inverse of M_k modulo m_k, for each k. Then by definition of inverse we have $M_k y_k \equiv 1 \pmod{m_k}$. Let,

$$x = a_1 M_1 x_1 + a_2 M_2 x_2 + \cdots + a_r M_r x_r . \text{MOD } M$$

Here x is the solution for above problem. Since operator $m_1, \ldots, m_r$ are pairwise relatively prime, any two simultaneous solutions to the system must be congruent modulo M. Thus the solution is a unique congruence class modulo M, and the value of x computed above is in that class.

Example:-

x===1 (mod 5)

x=== 1(mod 7)

x=== 3(mod 11)

Here 5, 7 and 11 are co-prime to each other therefore we will find out solution with Chinese Remainder Theorem.

As all numbers are relatively co-prime. We can write

gcd (5,7) = gcd (7,11) = gcd(5, 11) =1

Find M,

We have formula for $M = m_1 * m_2 * m_3$. (m_1…..m_n are co-prime numbers)

Here $m_1 = 5$, $m_2 = 7$, $m_3 = 11$.

Therefore, $M = m_1 * m_2 * m_3 = 5 * 7 * 11 = 385$.

Modulo $M_1 \ldots M_n$ are defined as $M_i = M / m_i$

Therefore,

$M_1 = M / m_1 = 385 / 5 = 77$.

$M_2 = M / m_2 = 385 / 7 = 55$.

$M_3 = M / m_3 = 385 / 11 = 35$.

Now we have to calculate x_i value,

$M_i x_i \pmod{m_i} = 1$.

Let us Calculate x_1

$M_1 x_1 \pmod{m_1} = 1$

$77. x_1 \pmod 5 = 1$

$2. x_1 \pmod 5 = 1$ …… (77 mod 5 = 2)

$x_1 = 3$ ……. ((2 * 3) mod 5 = 1)

Let us Calculate x_2

$M_2 x_2 \pmod{m_2} = 1$

$55. x_2 \pmod 7 = 1$

$6. x_2 \pmod 7 = 1$ …… (55 mod 7 = 6)

$x_2 = 6$ ……. ((6 * 6) mod 7 = 1)

Let us Calculate x_3

$M_3 x_3 \pmod{m_3} = 1$

$35. x_3 \pmod{11} = 1$

$2. x_3 \pmod{11} = 1$ …… (35 mod 11 = 2)

$X_3 = 6$ ……. ((2 * 6) mod 11 = 1)

Let us find out x

$$x = (a_1M_1x_1 + a_2M_2x_2 + \cdots + a_rM_rx_r.) \bmod M$$

$$= ((77 * 3 * 1) + (55 * 6 * 1) + (35 * 6 * 3)) \bmod 385$$

$$= (231 + 330 + 631) \bmod 385$$

$$= 1191 \bmod 385$$

$$= 36$$

4.7 Powers of an element:

Just as we often consider the multiples of a given element a, modulo n, we consider the sequence of powers of a,modulo n.

$a^0, a^1, a^2, a^3, \ldots$

modulo n. Indexing from 0, the 0th value in this sequence is a0 mod n D 1, and the ith value is ai mod n. For example, the powers of 3 modulo 7 are

i	0	1	2	3	4	5	6	7	8
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2

4.8 The RSA public-key cryptosystem

In the RSA public-key cryptosystem, a participant creates his or her public and secret keys with the following procedure:

1. Select at random two large prime numbers p and q such that p =/ q. The primes p and q might be, say, 1024 bits each.
2. Compute n = pq.
3. Select a small odd integer e that is relatively prime to pi (n) which, by equation equals (p-1)(q-1)
4. Compute d as the multiplicative inverse of e, modulo pi(n). guarantees that d exists and is uniquely defined. We can use the technique of
5. Publish the pair P =(e, n) as the participant's RSA public key.
6. Keep secret the pair S =(d,n)as the participant's RSA secret key.

4.9 NP-Completeness:-

P v/s NP Problem is very famous issue in computer Science. The Abbrevations are defined as follows:-

P is set of problems that can be solved by a deterministic Turing machine in Polynomial time.

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time.

Here, Polynomial time is defined as fixed or known amount of interval. Further we can say, P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).

Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm" ,a magical algorithm that always makes a right guess among the given set of choices.

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).

2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

4.9.1 Decision vs Optimization Problems:-

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

4.9.2 What is Reduction?

Let L1 and L2 be two decision problems. Suppose algorithm A2 solves L2. That is, if y is an input for L2 then algorithm A2 will answer Yes or No depending upon whether y belongs to L2 or not.

The idea is to find a transformation from L1 to L2 so that the algorithm A2 can be part of an algorithm A1 to solve L1.

Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

4.9.3 How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

4.10 Approximation Algorithms:-

An Approximate Algorithm is considered as solution for NP-COMPLETENESS in a optimized way. E.g. vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

4.10.1 Introduction to Approximation Algorithms

An approximation scheme for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\varepsilon > 0$ such that for any fixed ε, the scheme is a $(1 + \varepsilon)$ approximation algorithm. We say that an approximation scheme is a polynomial-time approximation scheme if

for any fixed ε > 0, the scheme runs in time polynomial in the size n of its input instance.

The running time of a polynomial-time approximation scheme can increase very rapidly as ε decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\varepsilon})$. Ideally, if ε decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor (though not necessarily the same constant factor by which ε decreased).

4.10.2 Vertex Cover

The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an optimal vertex cover. This problem is the optimization version of an NP-complete decision problem. A vertex cover of a graph is a subset of vertices which covers all edges. An edge is said to be covered if its endpoints are covered.

A vertex cover of an undirected graph G =(V, E) is a subset V' $\subseteq$ V such that if (u, v) is an edge of G, then either u ∈ V'or v ∈ V' (or both). The size of a vertex cover is the number of vertices in it.

APPROX-VERTEX-COVER(G)

C = ∅

E' = G. E

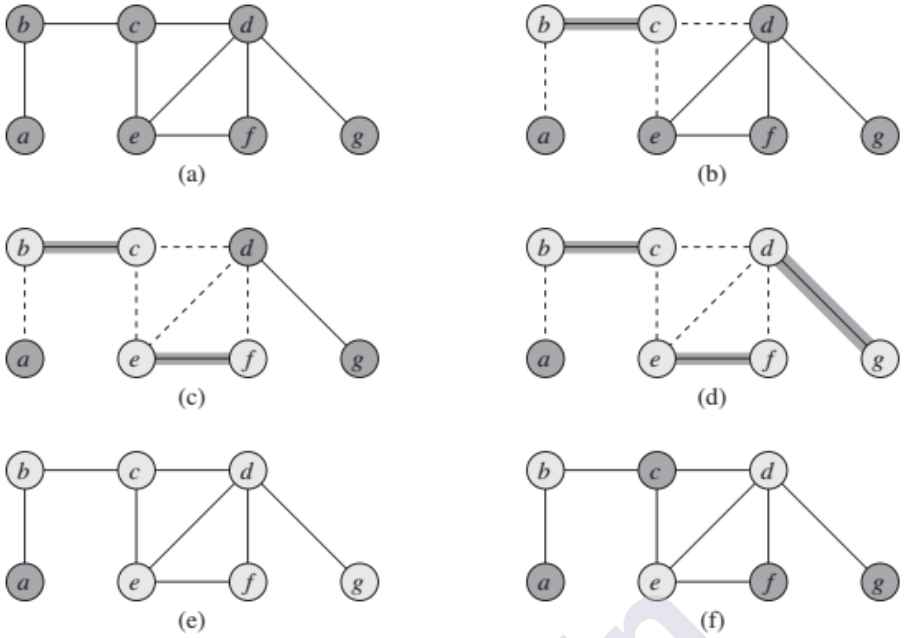
While E ≠ ∅

let (u, v) be an arbitrary edge of E'

C = C ∪ $\{u, v\}$

remove from E' every edge incident on either u or v

return C



The operation of APPROX-VERTEX-COVER.

(a) The input graph G, which has 7 vertices and 8 edges.

(b) The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEXCOVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C.

(c) Edge (e, f) is chosen; vertices e and f are added to C.

(d) Edge (d, g) is chosen; vertices d and g are added to C.

(e) The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g.

(f) The optimal vertex cover for this problem contains only three vertices: b, d, and e.

4.10.3 Travelling Salesman Problem (TSP):

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

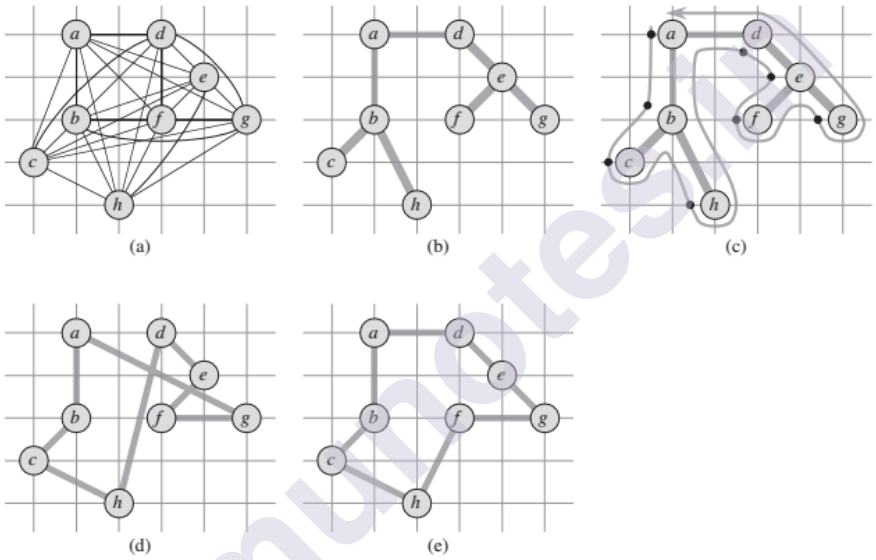
APPROX-TSP-TOUR(G,c)

select a vertex r ∈ G. *V* to be a "root" vertex

compute a minimum spanning tree T for G from root r

Let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T

return the hamiltonian cycle H



The operation of APPROX-TSP-TOUR.

(a) A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary Euclidean distance.

(b) A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order.

(c) A walk of T , starting at a. A full walk of the tree visits the vertices in the order a, b, c, b, h, b, a, d, e, f, e, g, e, d ,a. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g.

(d) A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19:074. (e) An optimal tour H^* for the original complete graph. Its total cost is approximately 14,715.

4.10.4 The set-covering problem

The set-covering problem is an optimization problem that models many problems that require resources to be allocated. Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard. The approximation algorithm developed to handle the vertex-cover problem doesn't apply here, however, and so we need to try other approaches.

GREEDY-SET-COVER(X, F)

U = X

C = D

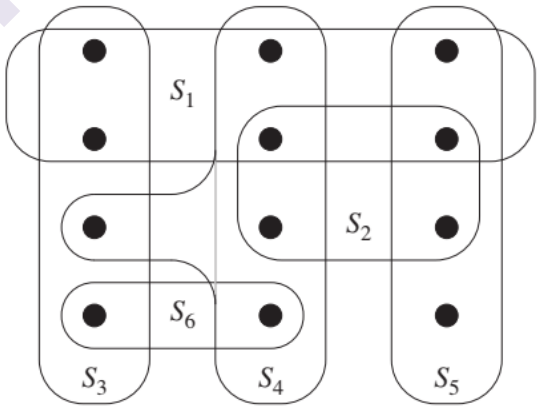
while U ≠ ∅

select an S ∈ F that maximizes |S ∩ U |

U = U - S

C = C ∪ {S}

return C



An instance (X, F) of the set-covering problem, where X consists of the 12 black points and F = {S1, S2, S3, S4, S5, S6}. A minimum-size set cover is C = {S3, S4, S5} , with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S1, S4, S5, and S3 or the sets S1, S4, S5, and S6, in order.

The subset sum problem is a decision problem in computer science. In its most general formulation, there is a multiset S of integers and a target sum T, and the question is to decide whether any subset of the integers sum to precisely T.[1] The problem is known to be NP-complete. Moreover, some restricted variants of it are NP-complete too, for example:[1]

The variant in which all input integers are positive.

The variant in which input integers may be positive or negative, and T = 0. For example, given the set { 7,-3,-2,9000,5,8|}.{ 7,-3,-2,9000,5,8}, the answer is yes because the subset{-3,-2,5\}{-3,-2,5} sums to zero.

Subset sum can also be regarded as an optimization problem: find a subset whose sum is as close as possible to T. It is NP-hard, but there are several algorithms that can solve it reasonably fast in practice.

4.10.5 The Subset Sum Problem:-

For the given the set S = {x_1, x_2, x_3, … x_n } of positive integers and t, is there a subset of S that adds up to t. Subset Sum is an optimization problem, what subset of S adds up to the greatest total <= t. Here, we use the notation S + x = { s+x : s S}. we have a merge lists algorithm that runs in time proportional to the sum of the lengths of the two lists.

EXACT-SUBSET-SUM(S,t)

1 n <- |S|

2 L_0 = <0>

3 for i = 1 to n

4 L_i = MERGE-LISTS(L_{i-1}, L_{i-1} +x_i)

5 remove from Li every element that is greater than t

6 return the largest element in L_n

4.11 Summary:-

Generally algorithm writing is a novel idea. Writing optimized algorithm is necessary thing in computer science. Here we have attempted to provide timewise and spacewise running of given algorithm.

4.12 References: -

- Algorithms, Sanjoy Dasgupta , Christos H. Papadimitriou, Umesh Vazirani, McGraw-Hill Higher Education (2006)
- Grokking Algorithms: An illustrated guide for programmers and other curious people, MEAP, Aditya Bhargava, http://www.manning.com/bhargava
- Research Methodology, Methods and Techniques, Kothari, C.R.,1985, third edition, New Age International (2014) .
- Basic of Qualitative Research (3rd Edition), Juliet Corbin & Anselm Strauss:, Sage Publications (2008)

4.13 Bibliography: -

- Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, PHI Learning Pvt. Ltd-New Delhi (2009).
- Researching Information Systems and Computing, Brinoy J Oates, Sage Publications India Pvt Ltd (2006).
- https://en.wikipedia.org/wiki/NP_(complexity)

4.14 Exercise: -

- What is NP Problem?
- Explain Vertex Cover Problem?
- Explain Set Cover Problem?
- How travelling salesman problem comes under NP Problem?
- What is running time of Vertex Cover problem in worst case?

Unit IV

5

RESEARCHING COMPUTING

Unit Structure

5.0 Introduction

Generally, Research is creation and presentation of new knowledge, ideas based on existing concept. These ideas can be presented in regular or creative approaches. These are used to generate new ideas, concepts ,methodologies and products as an output. One need to thoroughly

5.1 Purpose of research

- To add to the body of knowledge
- To solve a problem

- To find out what happens
- To find the evidence to inform practice
- To develop a greater understanding of people and their world
- To predict, plan and control
- To contribute to other people's wellbeing
- To test or disprove a theory
- To come up with a better way
- To understand another person's point of view
- To create more interest in the researcher;

5.1 Products of research

- A new or improved product
- A new theory
- A re-interpretation of an existing theory
- A new or improved research tool or technique
- A new or improved model or perspective
- An in-depth study of a particular situation
- An exploration of a topic, media, or field
- A critical analysis.

5.2 The Research Process

Research writing is an innovative idea to work on. It is nothing but organization of new concepts with the support of old concepts and hypothesis. One need to present his / her ideas in a logical way and larger context. There are few steps which together make research.

- ***Identify a Research Problem***

  The problem is based on general topic and issues. We need to find out the correct issue and problem to find out the exact solution.

- ***Review the Literature***

  Literature is nothing but the available material which will decide the direction of the research. Literature is findings of experienced person and we can embed their experienced views in our research.

- ***Determine Research Question***

  Any Research should be guided with a proper research question. A good question should have following characteristics:-

  - It must be understandable to researcher and to others.
  - It should be capable of developing into a manageable research design, so data may be collected in relation to it. Extremely abstract terms are unlikely to be suitable.
  - Connect with established theory and research. There should be a literature on which you can draw to illuminate how your research question(s) should be approached.
  - Be neither too broad nor too narrow. See Appendix A for a brief explanation of the narrowing process and how your research question, purpose statement, and hypothesis(es) are interconnected.

- ***Develop Research Methods:-***

  A good research is based on good research methods. Research methods provides input required to carry out a research. The Data is retrieved from Online/Offline Survey, Field visit and past research papers or documentation.

- ***Collect & Analyze Data***

  Collecting and analyzing data provides an excellent view regarding to output of the problems.

- ***Document the Work***

  Writing or documenting work is a disciplined work. While presenting research process is quite difficult to organize the data and presenting.

- **Communicate Your Research**

  It is very important to verify your result with targeted audience. While communicating your research researcher should follow research methods followed while developing research work.s

5.3 Internet research

Internet research is a research method in which information and data collected from Internet. Various qualitative journals and articles are available which are used to retrieve the information. As a practice there are variety of journals available which can be used to support the research.

One can collect various reviews and surveys through internet. There are various tools and techniques available to categorize the information. After searching you can get thousands of quick results for some topics. You can subscribe freely to several sites to receive updates regarding to your topic. You can subscribe to several groups also. Internet research is not like offline resource who are bound to certain limitation. Offline resources also makes restriction on availability. Internet research can provide quick, immediate, and worldwide access to information, although results may be affected by unrecognized bias, difficulties in verifying a writer's credentials (and therefore the accuracy or pertinence of the information obtained) and whether the searcher has sufficient skill to draw meaningful results from the abundance of material typically available.[2]

5.4 Participants and research ethics

- ***Discuss intellectual property frankly***

  Researcher can work on any predefined or well-known topic. It is highly possible that there might be a valuable research that is carried out by other. Even there are so many people who directly or indirectly support and contribute your research. It is important to give credit in the form of citation to all people and resources who support in your research. Here authorship should reflect the contribution.

  Researchers also need to meet their ethical obligations once their research is published: If authors learn of errors that change the interpretation of research findings, they are ethically obligated to promptly correct the errors in a correction, retraction, erratum or by other means.

- ***Be conscious of multiple roles***

  Perhaps one of the most common multiple roles for researchers is being both a mentor and lab supervisor to students they also teach in class. Psychologists need to be especially cautious that they don't abuse the power differential between themselves and students, say experts. They shouldn't, for example,

use their clout as professors to coerce students into taking on additional research duties.

- ***Follow informed-consent rules***

  Experts also suggest covering the likelihood, magnitude and duration of harm or benefit of participation, emphasizing that their involvement is voluntary and discussing treatment alternatives, if relevant to the research. Keep in mind that the Ethics Code includes specific mandates for researchers who conduct experimental treatment research. Specifically, they must inform individuals about the experimental nature of the treatment, services that will or will not be available to the control groups, how participants will be assigned to treatments and control groups, available treatment alternatives and compensation or monetary costs of participation. If research participants or clients are not competent to evaluate the risks and benefits of participation themselves--for example, minors or people with cognitive disabilities--then the person who's giving permission must have access to that same information, says Koocher.

- ***Respect confidentiality and privacy***

  It is very important to preserve confidentiality and privacy of your concepts and idea. One should also keep privacy on research method also.

5.5 Reviewing literature

Literature review is nothing but analyzing , summarizing previous work and opinion of expert researchers. There are two kinds of literature review :-

Dissertation literature review

Researcher can write literature review for dissertation purpose. In this kind of analysis researcher has to focus on detailed analysis. Researcher should write brief summery on given topic.

Stand-alone literature review

If Researcher are writing a stand-alone paper, give some background on the topic and its importance, discuss the scope of the literature you will review (for example, the time period of your sources), and state your objective.

We can write review in following columns.

- ***Introduction***

  Introduction is used to define focus and objective of literature review.

- ***Body***

  This section is more divided into subsections depending on length of the review. This is quite important and interesting section of writing. One has to be very dedicated while writing this section. There are some guidelines to write the statement which is listed below:-

  - **Summarize and synthesize:**

    Give an overview of the main points of each source and combine them into a coherent whole

  - **Analyze and interpret:**

    Don't just paraphrase other researchers—add your own interpretations where possible, discussing the significance of findings in relation to the literature as a whole

  - **Critically evaluate:**

    Mention the strengths and weaknesses of your sources

  - **Write in well-structured paragraphs:**

    Use transition words and topic sentences to draw connections, comparisons and contrasts

5.6 Defining Design and Creation

Research design is the framework of research methods and techniques that allows researchers to work in research methods that are suitable for the subject matter and set up their studies up for success. It includes the type of research (experimental, survey, correlational, semi-experimental, review) and also its sub-type (experimental design, research problem, descriptive case-study). Design is made up of three categories viz. Data collection, measurement, and analysis.

The essential elements of the research design are:

- Accurate purpose statement
- Techniques to be implemented for collecting and analyzing research
- The method applied for analyzing collected details

- Type of research methodology
- Probable objections for research
- Settings for the research study
- Timeline
- Measurement of analysis

Following characteristics are considered while developing research design:-

- **Neutrality**:

  The design should be neutral, it should not be in influence of any other pre-concept.

- **Reliability:**

  Research design should be according to standard rules and regularity which improves reliability.

- **Validity:**

  Design must be validated and verified with available tools.

- **Generalization:**

  The outcome of your design should apply to a population and not just a restricted sample.

Research design is broadly categorized in five categories: -

- **Descriptive research design:**

  In a descriptive design, a researcher is solely interested in describing the situation or case under their research study. It is a theory-based design method which is created by gathering, analyzing, and presenting collected data.

- **Experimental research design:**

  Experimental research design establishes a relationship between the cause and effect of a situation. It is a causal design where one observes the impact caused by the independent variable on the dependent variable.

- **Correlational research design:**

Correlational research is a non-experimental research design technique that helps researchers establish a relationship between two closely connected variables.

- **Diagnostic research design:**

  In diagnostic design, the researcher is looking to evaluate the underlying cause of a specific topic or phenomenon. This method helps one learn more about the factors that create troublesome situations.

5.7 Experimental research

Experimental research is research conducted with a scientific approach using two sets of variables. The first set acts as a constant, and second as Quantitative research methods. This method is used when we don't have previous data and we need to generate new data.

There are three primary types of experimental design:

- **Pre-experimental research design**: A group, or various groups, are kept under observation after implementing factors of cause and effect.
- **True experimental research design:** True experimental research relies on statistical analysis to prove or disprove a hypothesis, making it the most accurate form of research.
- **Quasi-experimental research design:** The word "Quasi" indicates similarity. A quasi-experimental design is similar to experimental, but it is not the same. The difference between the two is the assignment of a control group.

Advantages of experimental research

- Experimental research allows you to test your idea in a controlled environment before taking it to market. It also provides the best method to test your theory, thanks to the following advantages:
- Researchers have a stronger hold over variables to obtain desired results.
- The subject or industry does not impact the effectiveness of experimental research. Any industry can implement it for research purposes.
- The results are specific.

- After analyzing the results, you can apply your findings to similar ideas or situations.
- You can identify the cause and effect of a hypothesis. Researchers can further analyze this relationship to determine more in-depth ideas.
- Experimental research makes an ideal starting point. The data you collect is a foundation on which to build more ideas and conduct more research.
- Whether you want to know how the public will react to a new product or if a certain food increases the chance of disease, experimental research is the best place to start. Begin your research by finding subjects using QuestionPro Audience and other tools today.

5.8 Quantitative Data: Definition

Quantitative data is defined as the value of data in the form of counts or numbers where each data-set has an unique numerical value associated with it.

Quantitative data use various available tools for measuring

The most common types of quantitative data are as below:

Counter: Count equated with entities.

Measurement of physical objects: Calculating measurement of any physical thing.

Sensory calculation: Mechanism to naturally "sense" the measured parameters to create a constant source of information.

Projection of data: Future projection of data can be done using algorithms and other mathematical analysis tools.

Quantification of qualitative entities: Identify numbers to qualitative information.

Quantitative Data: Collection Methods

There are two main Quantitative Data Collection Methods:

Surveys: Traditionally, surveys were conducted using paper-based methods and have gradually evolved into online mediums. It is always effective method of collecting data. Questions are based on relative topics.

Longitudinal Studies: A type of observational research in which the market researcher conducts surveys from a specific time period to another, i.e., over a considerable course of time, is called longitudinal survey.

Cross-sectional Studies: A type of observational research in which the market research conducts surveys at a particular time period across the target sample is known as cross-sectional survey. This survey type implements a questionnaire to understand a specific subject from the sample at a definite time period.

Online/Telephonic Interviews: Telephone-based interviews are no more a novelty but these quantitative interviews have also moved to online mediums such as Skype or Zoom. Irrespective of the distance between the interviewer and the interviewee and their corresponding time zones, communication becomes one-click away with online interviews. In case of telephone interviews, the interview is merely a phone call away.

Computer Assisted Personal Interview: This is a one-on-one interview technique where the interviewer enters all the collected data directly into a laptop or any other similar device. The processing time is reduced and also the interviewers don't have to carry physical questionnaires and merely enter the answers in the laptop.

Advantages of Quantitative Data

- **Conduct in-depth research:** Since quantitative data can be statistically analyzed, it is highly likely that the research will be detailed.
- **Minimum bias:** There are instances in research, where personal bias is involved which leads to incorrect results. Due to the numerical nature of quantitative data, the personal bias is reduced to a great extent.
- **Accurate results:** As the results obtained are objective in nature, they are extremely accurate.

Disadvantages of Quantitative Data

- **Restricted information:** Because quantitative data is not descriptive, it becomes difficult for researchers to make decisions based solely on the collected information.
- **Depends on question types:** Bias in results is dependent on the question types included to collect quantitative data. The researcher's knowledge of questions and the objective of research are exceedingly important while collecting quantitative data.

5.9 Presentation in brief:

While presenting a research you should clearly mention your objective and research question before audience. Author or Presenter should reach along with research question an solution. It is very important, when considering your audience, to know:

- who they are?
- what their prior knowledge of the topic will be ?
- why they are likely to be interested?
- what their needs are and how you can help them?

The presentation should include: a short intro, your hypotheses, a brief description of the methods, tables and/or graphs related to your findings, and an interpretation of your data.

The trick to giving good presentations is distilling your information down into a few bulleted lists, diagrams, tables and graphs.

The Presentation should be divided into following groups:

- **Introduction .** Explain why your work is interesting. Place the study in context – how does it relate to / follow from the scientific literature on this subject. If it relates to any applied issues (e.g., environmental problems), mention this here. Use some pretty visuals (photographs, drawings, etc.) to get the audience excited about the issue and questions you are addressing. Clearly state your hypotheses.

- **Materials and Methods .** Clearly summarize the design. Shows a picture of your organisms and justify why they are appropriate for addressing the questions mentioned above. Show a picture of your lab setup and/or of a person doing some of the lab work. Show a diorama of your experimental design (with sample sizes, number of replicates, sampling frequency, etc.). Mention what parameters you measured but do not go into detail on exact procedures used. Do state what statistical tests you used to analyze your data.

- **Results.** First show a photograph (or sketch) that shows an interesting qualitative results (e.g., trays of plants in which one set is noticeably bigger

than the other, a drawing of a happy Daphnia) and state that result. Then display the results in graphical form, reminding the audience of your hypothesis and stating whether it was supported as you do so. Use simple, clean, clearly labeled graphs with proper axis labels (no extraneous 3-D effects please). Do not use light colors (yellow, light green, or pink) in your figures, they do not show up well when projected. Indicate the results of the statistical tests on the slides by including values (or asterisks/letters that indicate the significance level) on the same slides with the graphs.

- **Implications and Conclusions .** Correctly interpret your results. Constructively address sources of error and methodological difficulties. Place your results in context and draw mplications from them.

- **Acknowledgments.** Thank anyone who provided advice or assistance. Verbally thank your audience for their attention and tell them you would be happy to answer any questions.

5.10 References and Bibliography

- Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, PHI Learning Pvt. Ltd-New Delhi (2009).

- Researching Information Systems and Computing, Brinoy J Oates, Sage Publications India Pvt Ltd (2006).

- Algorithms, Sanjoy Dasgupta , Christos H. Papadimitriou, Umesh Vazirani, McGraw-Hill Higher Education (2006)

- Grokking Algorithms: An illustrated guide for programmers and other curious people, MEAP, Aditya Bhargava, http://www.manning.com/bhargava

- Research Methodology, Methods and Techniques, Kothari, C.R.,1985, third edition, New Age International (2014) .

- Basic of Qualitative Research (3rd Edition), Juliet Corbin & Anselm Strauss:, Sage Publications (2008)

5.10 Exercise:-

- What is research?
- What do you mean by plagiarism?
- What research ethics should be followed by researcher?
- How to write a research paper?