

2019

Admas University

THEORY OF ALGORITHMS

Lecture Note

Chapter One: Foundations

What is an algorithm?

Algorithm is a set of steps to complete a task.

For example:-

Task: to make a cup of tea.

Algorithm:

- ❖ add water to the kettle,
- ❖ boil it, add tea leaves,
- ❖ Add sugar, and then serve it in cup.

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space. An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming languages**.

Simply put, an *algorithm* is a step-by-step procedure for performing some task in a finite amount of time, and a *data structure* is a systematic way of organizing and accessing data

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well defined inputs.
- **Output** – An algorithm should have 1 or more well defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions which should be independent of any programming code.

Analysis of Algorithms

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm. Algorithms are often quite different from one another, though the objective of these algorithms is the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis:

Worst-case: The maximum number of steps taken on any instance of size **a**.

Best-case: The minimum number of steps taken on any instance of size **a**.

Average case: An average number of steps taken on any instance of size **a**.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Methodology of Analysis

To measure resource consumption of an algorithm, different strategies are used.

Asymptotic Analysis

The asymptotic behavior of a function **f(n)** refers to the growth of **g(n)** as **n** gets large. We typically ignore small values of **n**, since we are usually interested in estimating how slow the program will be on large inputs. A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it's not always true.

For example, a linear algorithm $(n) = d * n + k$ is always asymptotically better than a quadratic one, $(n) = c n^2 + q$

Amortized Analysis

Amortized analysis is generally used for certain algorithms where sequences of similar operations are performed.

- Amortized analysis provides a bound on the actual cost of the entire sequence, instead of bounding the cost of sequence of operations separately.
- Amortized analysis differs from average-case analysis; probability is not involved in amortized analysis. Amortized analysis guarantees the average performance of each operation in the worst case.

It is not just a tool for analysis; it's a way of thinking about the design, since designing and analysis are closely related.

Asymptotic Notations & Apriori Analysis

In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works. The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time. Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** – The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** – The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm $f(n)$ gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Time Complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function $T(n)$, where $T(n)$ can be measured as the number of steps, provided each step consumes constant time. For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for addition of two bits. Here, we observe that $T(n)$ grows linearly as input size increases.

Space Complexities

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components

- A fixed part that is a space required to store certain data and variables that are independent of the size of the problem. For example simple variables & constant used, program size etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stacks space etc.

Space complexity $S(P)$ of any algorithm P is $S(P) = C + SP(I)$ Where C is the fixed part and $S(I)$ is the variable part of the algorithm which depends on instance characteristic I . Following is a simple example that tries to explain the concept –

Algorithm: SUM(A, B)
 Step 1 - START
 Step 2 - $C \leftarrow A + B + 10$
 Step 3 - Stop

Here we have three variables A, B and C and one constant. Hence $S(P) = 1+3$. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

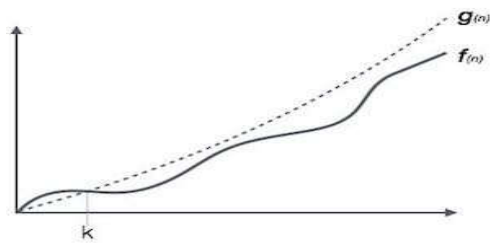
Time function of an algorithm is represented by $f(n)$, where n is the input size. Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- **O Notation** (“Big Oh Notation “)
- **Ω Notation** (“Omega Notation”)
- **θ Notation** (“Theta Notation”)

Big Oh Notation, O

The $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.



For example, for a function $f(n)$

$$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n > n_0. \}$$

Example

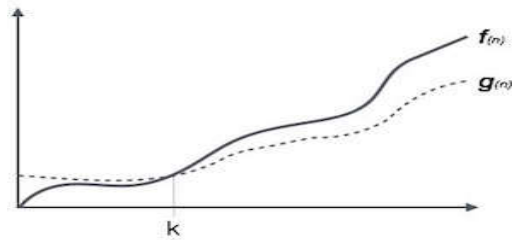
Let us consider a given function, $f(n) = 4n^3 + 10n^2 + 5n + 1$

Considering $g(n) = n^3$

$f(n) \leq 5 \cdot g(n)$ for all the values of $n > 2$. Hence, the complexity of $f(n)$ can be represented as $O(g(n))$, i.e. $O(n^3)$.

Omega Notation, Ω

The $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.



For example, for a function $f(n)$

$$\Omega(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n > n_0. \}$$

Example

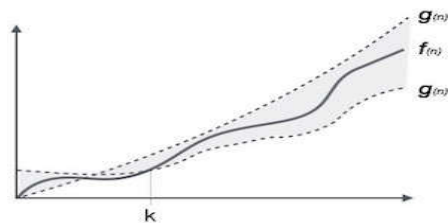
Let us consider a given function, $f(n) = 4n^3 + 10n^2 + 5n + 1$

Considering $g(n) = n^3$, $f(n) \geq 4 \cdot g(n)$ for all the values of $n > n_0$

Hence, the complexity of $f(n)$ can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$.

Theta Notation, Θ

The $\theta(n)$ is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following –



$$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$$

Example

Let us consider a given function, $f(n) = 4n^3 + 10n^2 + 5n + 1$

Considering $g(n) = n^3$, $4 \cdot g(n) \leq f(n) \leq 5 \cdot g(n)$ for all the large values of n

Hence, the complexity of $f(n)$ can be represented as $\Theta(g(n))$, i.e. $\Theta(n^3)$.

Growth rate of Asymptotic Notation

The growth rate of the asymptotic notation is given from smallest to the biggest as follows:

- 1) Constant $O(1)$
- 2) Logarithmic $O(\log n)$
- 3) Linear $O(n)$
- 4) $n \log n$ $O(n \log n)$
- 5) quadratic $O(n^2)$
- 6) cubic $O(n^3)$
- 7) exponential $O(2^n)$

EXAMPLE 1

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

We must find c_1 and c_2 such that

$$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$$

Dividing both sides by n^2 yields

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

For $n_0 \geq 7$, $\frac{1}{2} n^2 - 3n = \Theta(n^2)$

EXAMPLE 2

$$f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$$

Because :

$$3n^2 - 2n + 5 = \Omega(n^2)$$

$$3n^2 - 2n + 5 = O(n^2)$$

Recurrence

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm.

Let us consider **T(n)** to be the running time on a problem of size **n**.

If the problem size is small enough, say $n < C$ where C is a constant, the straightforward solution takes constant time, which is written as $\theta(1)$. If the division of the problem yields a number of sub-problems with size n/b .

To solve the problem, the required time is **a.T(n/b)** If we consider the time required for division is **D(n)** and the time required for combining the results of sub-problems is **C(n)**, the recurrence relation can be represented as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n \leq C \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$$

A recurrence relation can be solved using the following methods:

- **Substitution Method** – in this method, we guess a bound and using mathematical induction we prove that our assumption was correct.
- **Recursion Tree Method** – in this method, a recurrence tree is formed where each node represents the cost.
- **Master's Theorem** – this is another important technique to find the complexity of a recurrence relation.

Searching Techniques

Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if $i > n$ then go to step 7

Step 3: if $A[i] = x$ then go to step 6

Step 4: Set i to $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Binary search

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the right of the middle item. Otherwise, the item is searched for in the sub-array to the left of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

Sorting Techniques

Sorting algorithm

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios:

- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

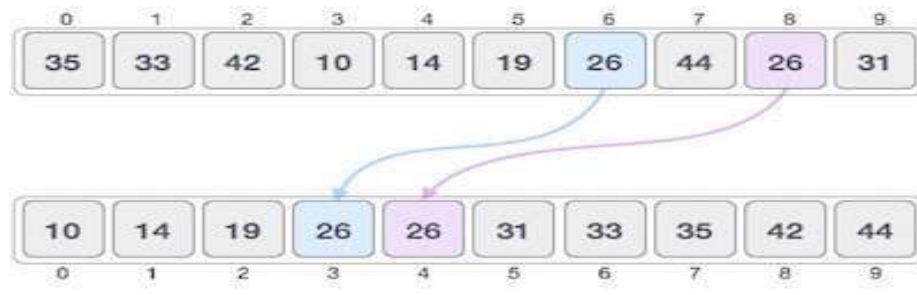
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

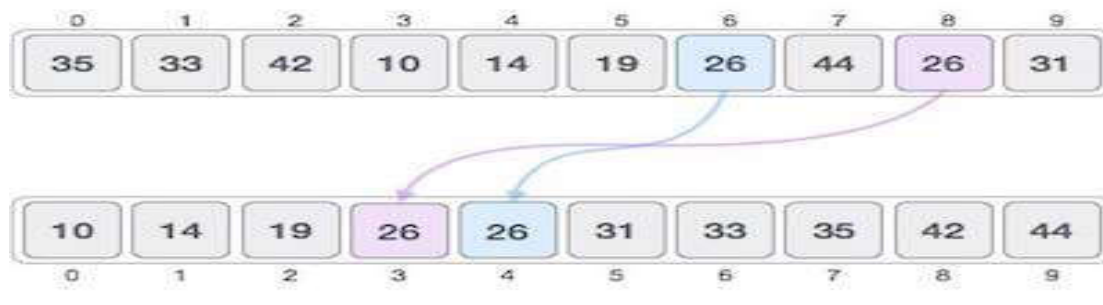
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **notin- place sorting**. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Bubble Sort

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted. This is the simplest technique among all sorting algorithms.

Algorithm: Sequential-Bubble-Sort (A)

```
for i ← 1 to length [A] do
  for j ← length [A] down-to i +1 do
    if A[j] < A[j-1] then
      Exchange A[j] ↔ A[j-1]
```

Analysis

Here, the number of comparisons are $1+2+3 \dots +(n-1)=n(n-1)/2=O(n^2)$. In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Memory Requirement

From the algorithm stated above, it is clear that bubble sort does not require extra memory.

Example

Unsorted list:

5	2	1	4	3	7	6
---	---	---	---	---	---	---

1st iteration:

5 > 2 swap

2	5	1	4	3	7	6
---	---	---	---	---	---	---

5 > 1 swap

2	1	5	4	3	7	6
---	---	---	---	---	---	---

5 > 4 swap

2	1	4	5	3	7	6
---	---	---	---	---	---	---

5 > 3 swap

2	1	4	3	5	6	7
---	---	---	---	---	---	---

5 < 7 no swap

2	1	4	3	5	7	6
---	---	---	---	---	---	---

7 > 6 swap

2	1	4	3	5	6	7
---	---	---	---	---	---	---

2nd iteration:

2 > 1 swap

1	2	4	3	5	6	7
---	---	---	---	---	---	---

2 < 4 no swap

1	2	4	3	5	6	7
---	---	---	---	---	---	---

4 > 3 swap

1	2	3	4	5	6	7
---	---	---	---	---	---	---

4 < 5 no swap

1	2	3	4	5	6	7
---	---	---	---	---	---	---

5 < 6 no swap

1	2	3	4	5	6	7
---	---	---	---	---	---	---

There is no change in 3rd, 4th, 5th and 6th iteration.

Finally, the sorted list is

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Insertion Sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game. The numbers, which are needed to be sorted, are known as **keys**. Here is the algorithm of the insertion sort method

Algorithm: Insertion-Sort(A)

```
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Analysis

Run time of this algorithm is very much dependent on the given input. If the given numbers are sorted, this algorithm runs in **O(n)** time. If the given numbers are in reverse order, the algorithm runs in **O(n²)** time.

Example

Unsorted list:

2	13	5	18	14
2	13	5	18	14

1st iteration:

Key = $a[2] = 13$

$a[1] = 2 < 13$

So, no swap.

2nd iteration:

Key = $a[3] = 5$

$a[2] = 13 > 5$

Swap 5 and 13

2	5	13	18	14
---	---	----	----	----

Next, $a[1] = 2 < 13$

So, no swap

2	5	13	18	14
---	---	----	----	----

3rd iteration:

Key = $a[4] = 18$

$a[3] = 13 < 18$,

$a[2] = 5 < 18$,

$a[1] = 2 < 18$

So, no swap

2	5	13	18	14
---	---	----	----	----

4th iteration:

Key = $a[5] = 14$

$a[4] = 18 > 14$

Swap 18 and 14

Next, $a[3] = 13 < 14$,

$a[2] = 5 < 14$,

$a[1] = 2 < 14$

So, no swap

Finally, the sorted list is

2	5	13	14	18
---	---	----	----	----

Selection Sort

This type of sorting is called **Selection Sort** as it works by repeatedly sorting elements. It works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Algorithm: Selection-Sort (A)

```
for i ← 1 to n-1 do
    min j ← i;
    min x ← A[i]
    for j ← i + 1 to n do
        if A[j] < min x then
            min j ← j
            min x ← A[j]
    A[min j] ← A[i]
```

Selection sort is among the simplest of sorting techniques and it works very well for small files. It has a quite important application as each item is actually moved at the most once. Selection sort is a method of choice for sorting files with very large objects (records) and small keys. The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.

Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if $A[j] < \min x$ is executed exactly the same number of times in every case.

Selection sort spends most of its time trying to find the minimum element in the unsorted part of the array. It clearly shows the similarity between Selection sort and Bubble sort.

- Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
- Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

For each i from 1 to $n - 1$, there is one exchange and $n - i$ comparisons, so there is a total of $n - 1$ exchanges and $(n-1)+(n-2)+\dots+2+1 = n(n-1)/2$ comparisons.

These observations hold, no matter what the input data is. In the worst case, this could be quadratic, but in the average case, this quantity is $O(n \log n)$. It implies that the running time of Selection sort is quite insensitive to the input.

Example

5	2	1	4	3
---	---	---	---	---

Unsorted list:

1st iteration:

Smallest = 5

$2 < 5$, smallest = 2

$1 < 2$, smallest = 1

$4 > 1$, smallest = 1

$3 > 1$, smallest = 1

Swap 5 and 1

1	2	5	4	3
---	---	---	---	---

2nd iteration:

Smallest = 2

$2 < 5$, smallest = 2

1	2	5	4	3
---	---	---	---	---

2 < 4, smallest = 2

2 < 3, smallest = 2

No swap

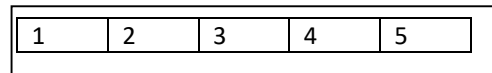
3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3

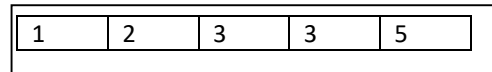


4th iteration:

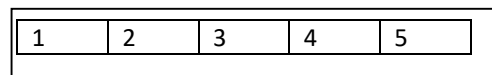
Smallest = 4

4 < 5, smallest = 4

No swap



Finally, the sorted list is



Merge sort

Merge-sort applies the divide-and-conquer technique to the sorting problem, where, for the sake of generality, let us consider the sorting problem to take a sequence, S , of objects as input, which could be represented with either a list or an array, and returns S in sorted order.

For the problem of sorting a sequence S with n elements, the three divide-and conquer steps are as follows:

1. **Divide:** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), put the elements of S into two sequences, S_1 and S_2 , each containing about half of the elements of S ; that is, S_1 contains the first $\lfloor n/2 \rfloor$ elements of S , and S_2 contains the remaining $n/2$ elements.
2. **Recur:** Recursively sort the sequences S_1 and S_2 .
3. **Conquer:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

We can visualize an execution of the merge-sort algorithm using a binary tree T , called the merge-sort tree.

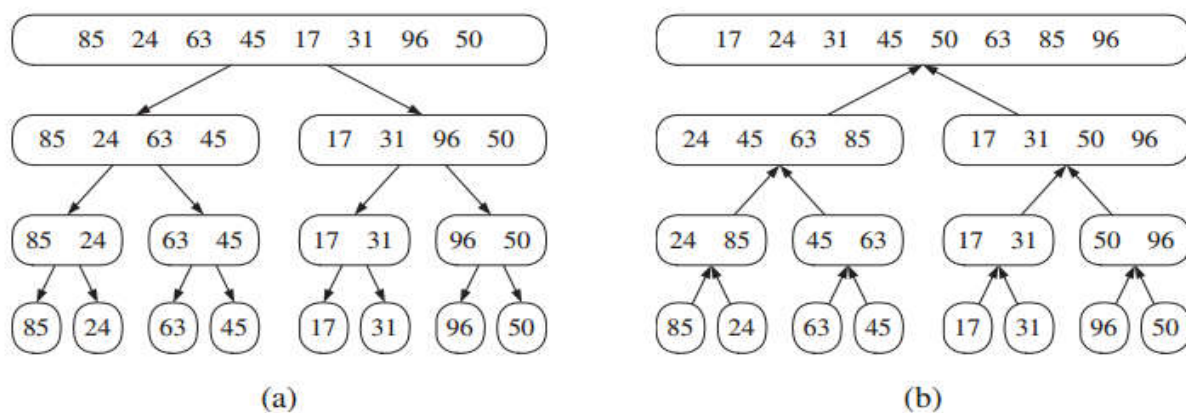


Figure: Merge-sort tree T for an execution of the merge-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T ; (b) output sequences generated at each node of T .

Having given an overview of merge-sort and an illustration of how it works, let us consider each of the steps of this divide-and-conquer algorithm in more detail. The divide and recur steps of the merge-sort algorithm are simple; dividing

a sequence of size n involves separating it at the element with rank $\lceil n/2 \rceil$, and the recursive calls simply involve passing these smaller sequences as parameters. The difficult step is the conquer step, which merges two sorted sequences into a single sorted sequence. We provide a pseudocode description of the method for merging two sorted arrays in Algorithm below. It merges two sorted arrays, S_1 and S_2 , by iteratively removing a smallest element from one of these two and adding it to the end of an output array, S , until one of these two arrays is empty, at which point we copy the remainder of the other array to the output array.

Algorithm merge(S_1, S_2, S):

Input: Two arrays, S_1 and S_2 , of size n_1 and n_2 , respectively, sorted in non-decreasing order, and an empty array, S , of size at least $n_1 + n_2$

Output: S , containing the elements from S_1 and S_2 in sorted order

```

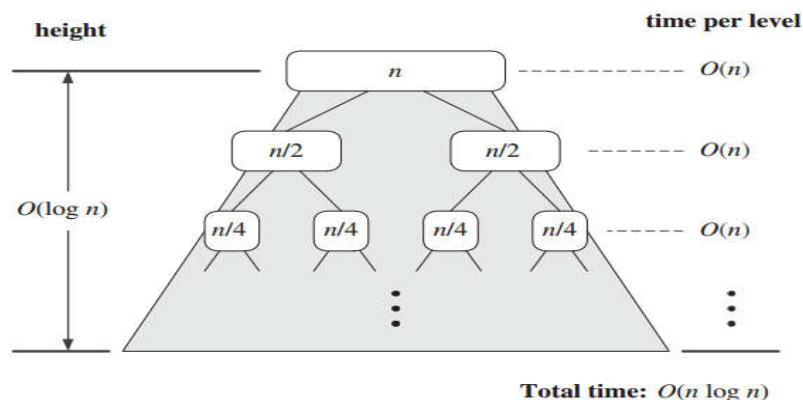
 $i \leftarrow 1$ 
 $j \leftarrow 1$ 
while  $i \leq n$  and  $j \leq n$  do
    if  $S_1[i] \leq S_2[j]$  then
         $S[i + j - 1] \leftarrow S_1[i]$ 
         $i \leftarrow i + 1$ 
    else
         $S[i + j - 1] \leftarrow S_2[j]$ 
         $j \leftarrow j + 1$ 
while  $i \leq n$  do
     $S[i + j - 1] \leftarrow S_1[i]$ 
     $i \leftarrow i + 1$ 
while  $j \leq n$  do
     $S[i + j - 1] \leftarrow S_2[j]$ 
     $j \leftarrow j + 1$ 

```

Algorithm : Merging two sorted arrays, with indexing beginning at 1.

Merging two sorted arrays S_1 and S_2 takes $O(n_1 + n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2

Having given the details of the merge algorithm, let us analyze the running time of the entire merge-sort algorithm, assuming it is given an input sequence of n elements. For simplicity, let us also assume n is a power of 2. We analyze the merge-sort algorithm by referring to the merge-sort tree, T .



Merge-sort on sequence of n elements runs in $O(n \log n)$ time.

The running time of the merge-sort algorithm is $O(n \log n)$. Let the function $t(n)$ denote the worst-case running time of merge-sort on an input sequence of size n . Since merge-sort is recursive, we can characterize function $t(n)$ by means of the following equalities, where function $t(n)$ is recursively expressed in terms of itself, as follows:

$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{otherwise} \end{cases}$$

Quick – Sort

The quick-sort algorithm sorts a sequence S using a simple divide-and-conquer approach, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps:

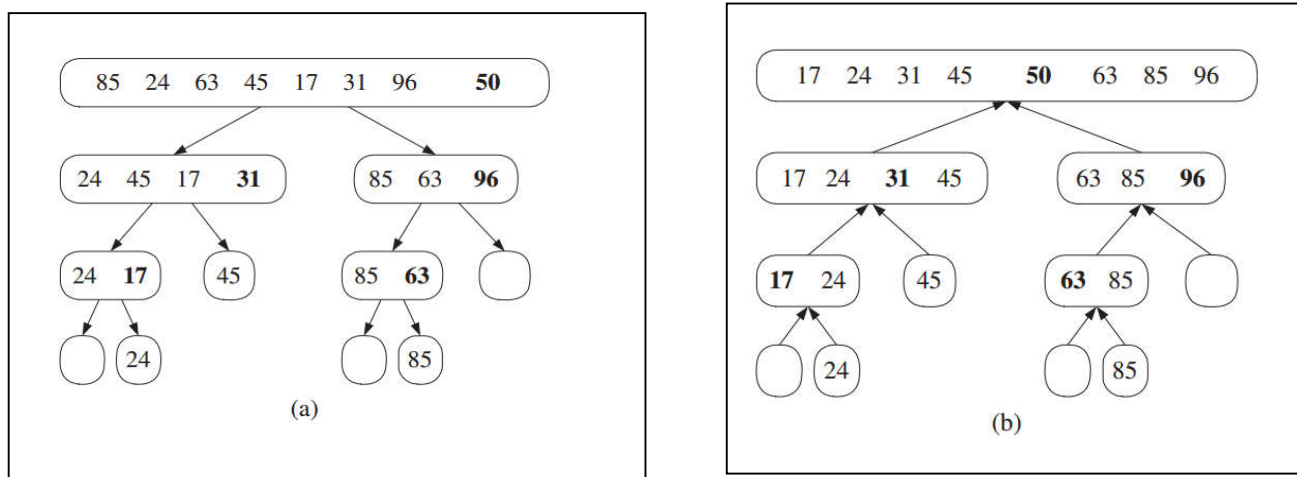
1. **Divide:** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S , which is called the **pivot**. As is common practice, choose the pivot x to be the last element in S . Remove all the elements from S and put them into three sequences:
 - L , storing the elements in S less than x
 - E , storing the elements in S equal to x
 - G , storing the elements in S greater than x .

(If the elements of S are all distinct, E holds just one element—the pivot.)

2. **Recur:** Recursively sort sequences L and G .

3. **Conquer:** Put the elements back into S in order by first inserting the elements of L , then those of E , and finally those of G .

Like merge-sort, we can visualize quick-sort using a binary recursion tree, called the quick-sort tree. Figure below visualizes the quick-sort algorithm



Running Time of Quick - Sort

we can analyze the running time of quick-sort with the same technique used for merge-sort. We identify the time spent at each node of the quick-sort tree T and we sum up the running times for all the nodes.

Quick-sort runs in $O(n^2)$ worst-case time In the best case, T has height $O(\log n)$ and quick-sort runs in $O(n \log n)$ time. Average running time of quick-sort to be similar to the best-case running time, that is, $O(n \log n)$.

Algorithm inPlacePartition(S, a, b):

Input: An array, S , of distinct elements; integers a and b such that $a \leq b$

Output: An integer, l , such that the subarray $S[a..b]$ is partitioned into $S[a..l-1]$ and $S[l..b]$ so that every element in $S[a..l-1]$ is less than each element in $S[l..b]$

Let r be a random integer in the range $[a, b]$

Swap $S[r]$ and $S[b]$

$p \leftarrow S[b]$ // the pivot

$l \leftarrow a$ // l will scan rightward

$r \leftarrow b - 1$ // r will scan leftward

while $l \leq r$ **do** // find an element larger than the pivot

while $l \leq r$ and $S[l] \leq p$ **do**

$l \leftarrow l + 1$

while $r \geq l$ and $S[r] \geq p$ **do** // find an element smaller than the pivot

$r \leftarrow r - 1$

if $l < r$ **then**

 Swap $S[l]$ and $S[r]$

Swap $S[l]$ and $S[b]$ // put the pivot into its final place

return l

Algorithm inPlaceQuickSort(S, a, b):

Input: An array, S , of distinct elements; integers a and b

Output: The subarray $S[a..b]$ arranged in nondecreasing order

if $a \geq b$ **then return** // subarray with 0 or 1 elements

$l \leftarrow \text{inPlacePartition}(S, a, b)$

inPlaceQuickSort($S, a, l - 1$)

inPlaceQuickSort($S, l + 1, b$)

Linear and non Linear Data Structure

Simply put, a **data structure** is a systematic way of organizing and accessing data, and an **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.

LINKED LIST

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- ✓ **Link** – Each link of a linked list can store a data called an element.
- ✓ **Next** – Each link of a linked list contains a link to the next link called Next.
- ✓ **Linked List** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

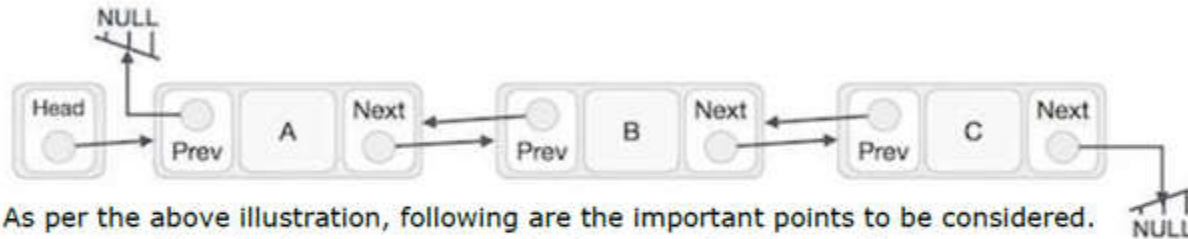
- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward or backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **Linked List** – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

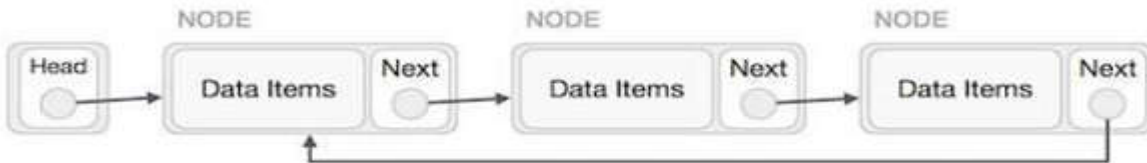
- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Circularly Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Stack

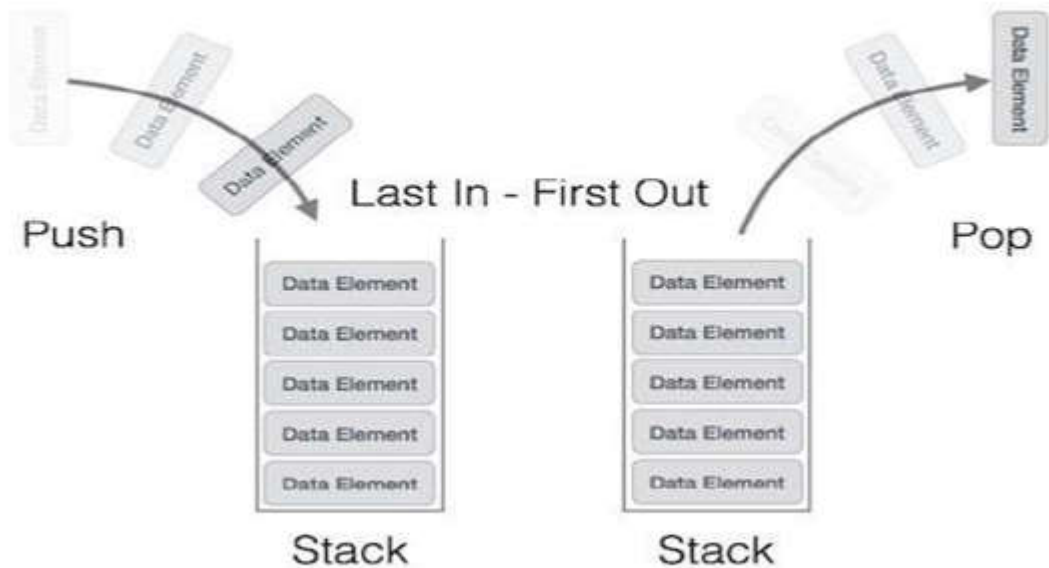
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack. This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack Representation

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

QUEUE

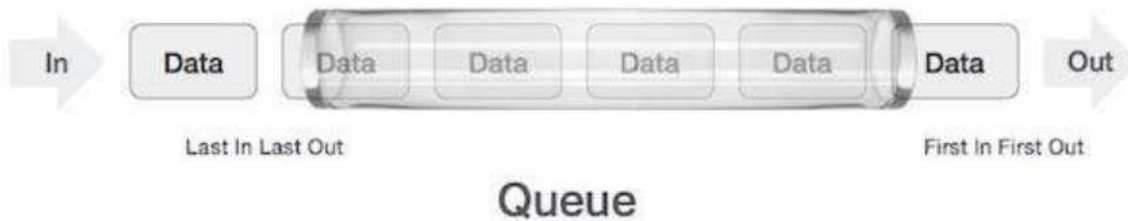
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows **First-In-First-Out** methodology, i.e., the data item stored first will be accessed first



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Graphs

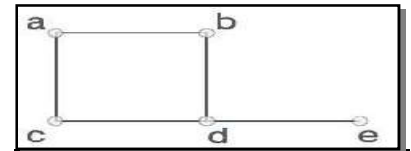
A **graph** is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

→ Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph:

In the above graph,

$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$



Graph Data Structure

- ✚ **Vertex** – Each node of the graph is represented as a vertex
- ✚ **Edge** – Edge represents a path between two vertices or a line between two vertices.
- ✚ **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the above diagram, **b** is adjacent to **a**, **c** is adjacent to **b**, and so on.
- ✚ **Path** – Path represents a sequence of edges between the two vertices. In the following example, **abde** represents a path from **a** to **e**.

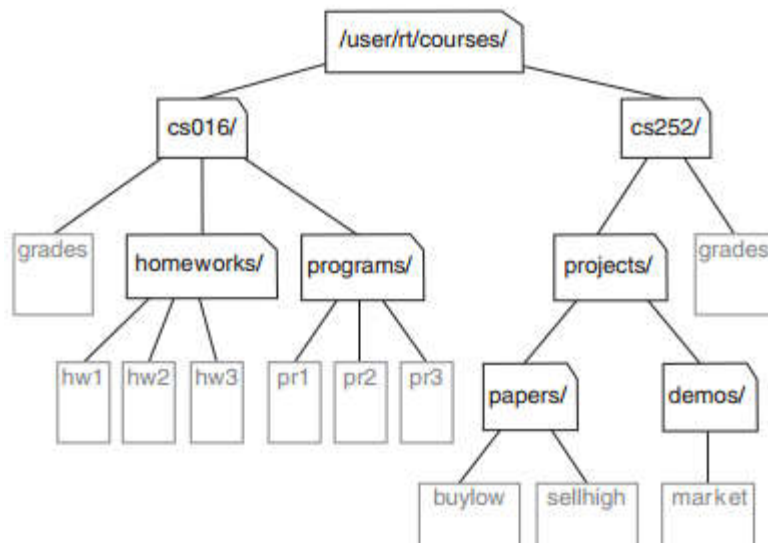
Basic Operations

Following are the basic primary operations that can be performed on a Graph:

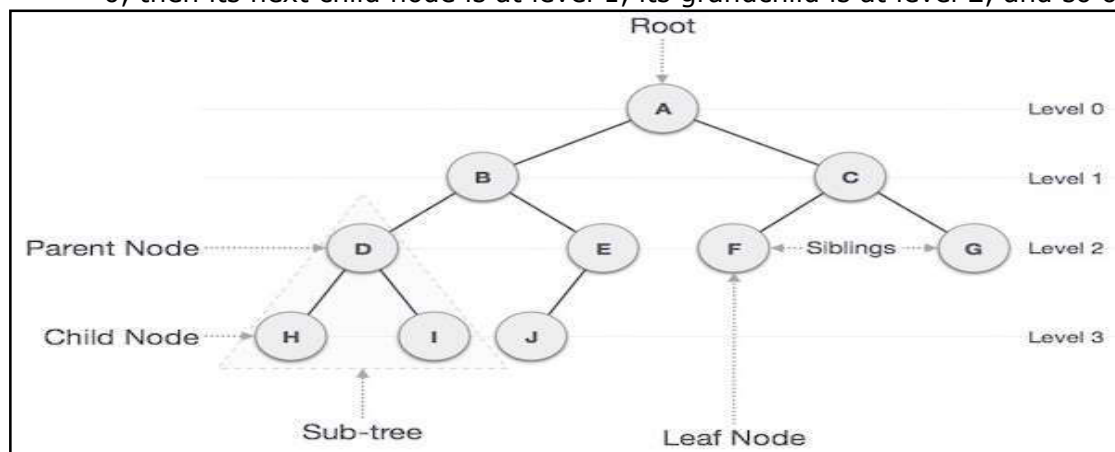
- ✚ **Add Vertex** – Adds a vertex to the graph.
- ✚ **Add Edge** – Adds an edge between the two vertices of the graph.
- ✚ **Display Vertex** – Displays a vertex of the graph.

Tree

The data structures that we have discussed that the linked list and stack are linear data structures. In these structures, the elements are in a line. We put and get elements in and from a stack in linear order. Queue is also a linear data structure as a line is developed in it. There are a number of applications where linear data structures are not appropriate. In such cases, there is need of some non-linear data structure. Some examples will show us that why nonlinear data structures are important. Tree is one of the non-linear data structures. Look at the following figure.



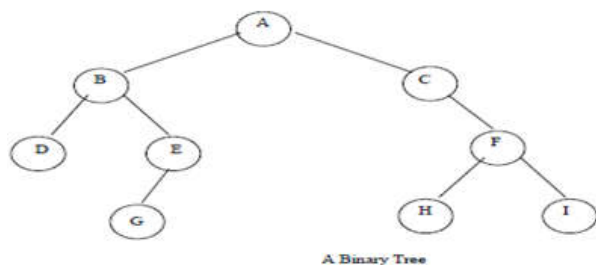
- A **tree** is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and sub-trees of children with a parent node, represented as a set of linked nodes.
- A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children").
- A tree is a data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the *null or empty tree*. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy. Terminologies used in tree include root, children, sibling, parent, descendant, leaf & ancestor.
- **Degree**: The number of sub trees of a node &
- **Depth**: The depth of a node is the number of edges from the tree's root node to the node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Sub tree** – Sub-tree represents the descendants of a node
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.



Binary Tree

The mathematical definition of a binary tree is “A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the

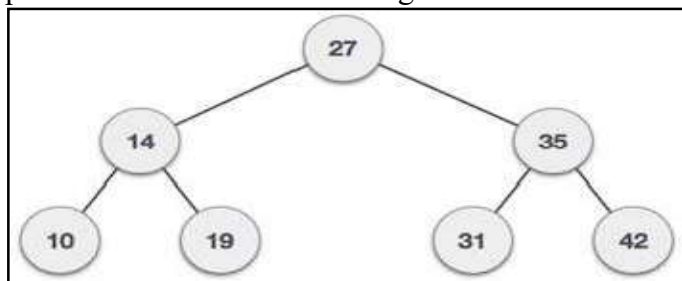
root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees”. Each element of a binary tree is called a node of the tree. Following figure shows a binary tree.



A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- ✓ The left sub-tree of a node has a key less than or equal to its parent node's key.
- ✓ The right sub-tree of a node has a key greater than or equal to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub tree and can be defined as –

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

Binary Search Tree Basic Operations

The basic operations that can be performed on a binary search tree data structure are the following

➔ **Insert** – Inserts an element in a tree/create a tree.

➔ **Search** – Searches an element in a tree.

➔ **In-order Traversal:** In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree.

➔ **Pre-order Traversal:** In this traversal method, the root node is visited first, then the left sub tree and finally the right sub tree.

➔ **Post-order Traversal:** In this traversal method, the root node is visited last, hence the name. First we traverse the left sub tree, then the right sub tree and finally the root node

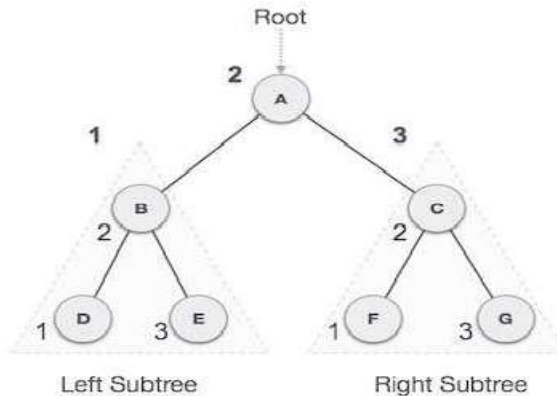
Tree traversal & applications

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- ✓ *In-order Traversal*
- ✓ *Pre-order Traversal*
- ✓ *Post-order Traversal*

In-order Traversal

In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a sub tree itself. If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left sub tree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in order traversal of this tree will be –

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm

Until all nodes are traversed –

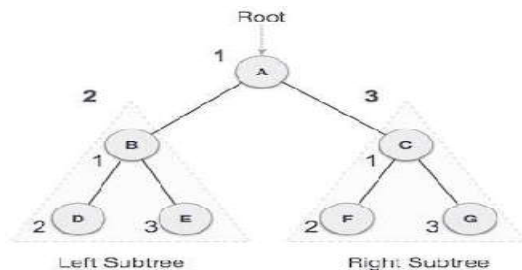
Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left sub tree and finally the right sub tree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left sub tree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Algorithm

Until all nodes are traversed –

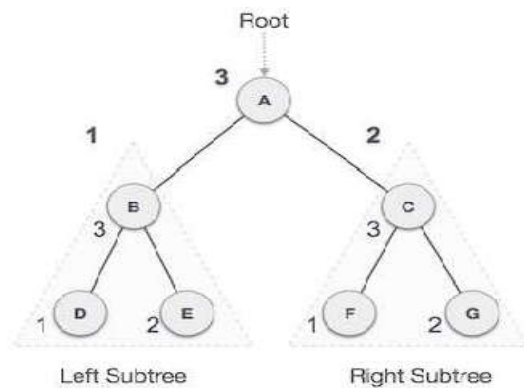
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left sub tree, then the right sub tree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left sub tree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

Chapter Two: Advanced Design and Analysis Techniques

The Greedy Method

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen. **Greedy algorithms** try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

In general, greedy algorithms have five components:

- i. A candidate set, from which a solution is created
- ii. A selection function, which chooses the best candidate to be added to the solution
- iii. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
- iv. An objective function, which assigns a value to a solution, or a partial solution, and
- v. A solution function, which will indicate when we have discovered a complete solution

A game like chess can be won only by *thinking ahead*: a player who is focused entirely on immediate advantage is easy to defeat. But in many other games, it is possible to do quite well by simply making whichever move seems best at the moment and not worrying too much about future consequences. This sort of myopic behavior is easy and convenient, making it an attractive algorithmic strategy.

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal. Our first example is that of **minimum spanning trees**.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming.

Examples

Most networking algorithms use the greedy approach. Here is a list of few of them –

- ✓ *Dijkstra's Minimal Spanning Tree Algorithm*
- ✓ *Knapsack Problem*
- ✓ *Job Scheduling Problem*
- ✓ *Prim's Minimal Spanning Tree Algorithm*
- ✓ *Kruskal's Minimal Spanning Tree Algorithm*

Fractional Knapsack

The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Although the same problem could be solved by employing other algorithmic approaches, Greedy approach solves Fractional Knapsack problem reasonably in a good time. Let us discuss the Knapsack problem in detail.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a sub problem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i . What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are n items in the store
- Weight of i^{th} item $w_i > 0$
- Profit for i^{th} item $p_i > 0$ and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0 \leq x_i \leq 1$$

The i^{th} item contributes the weight $x_i \cdot w_i$ to the total weight in the knapsack and profit $x_i \cdot p_i$ to the total profit. Hence, the objective of this algorithm is to

$$\text{maximize } \sum_{i=1}^n (x_i \cdot p_i)$$

subject to constraint,

$$\sum_{i=1}^n (x_i \cdot w_i) \leq W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{i=1}^n (x_i \cdot w_i) = W$$

In this context, first we need to sort those items according to the value of p_i/w_i , so that $\frac{p_{i+1}}{w_{i+1}} \leq \frac{p_i}{w_i}$. Here, x is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack ($w[1..n]$, $p[1..n]$, W)

for $i = 1$ to n

do $x[i] = 0$

weight = 0

for $i = 1$ to n

if weight + $w[i] \leq W$ then

$x[i] = 1$

weight = weight + $w[i]$

else

$x[i] = (W - \text{weight}) / w[i]$

weight = W

break

return x

Analysis

If the provided items are already sorted into a decreasing order of p_i/w_i then the while loop takes a time in (n) ; Therefore, the total time including the sort is in $(n \log n)$.

Example

Let us consider that the capacity of the knapsack $w = 60$ and the list of provided items are shown in the following table:

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	7	10	6	5

As the provided items are not sorted based on p_i / w_i . After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	10	7	6	5

Solution

After sorting all the items according to p_i/w_i First all of **B** is chosen as weight of **B** is less than the capacity of the knapsack. Next, item **A** is chosen, as the available capacity of the knapsack is greater than the weight of **A**. Now, **C** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of **C**.

Hence, fraction of C (i.e. $(60 - 50)/20$) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected. The total weight of the selected items is $10 + 40 + 20 * (10/20) = 60$

And the total profit is $100 + 280 + 120 * (10/20) = 380 + 60 = 440$

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Job Sequencing with Deadline

Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution

Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit. It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of i^{th} job J_i is d_i and the profit received from this job is p_i . Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, $(i) > 0$ for $1 \leq i \leq n$

Initially, these jobs are ordered according to profit, i.e. $p_1 \geq p_2 \geq p_3 \geq \dots \geq p_n$

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```

D(0) := J(0) := 0
k := 1
J(1) := 1 // means first job is selected
for i = 2 ... n do
    r := k
    while D(J(r)) > D(i) and D(J(r)) ≠ r do
        r := r - 1
    if D(J(r)) ≤ D(i) and D(i) > r then
        for l = k ... r + 1 by -1 do
            J(l + 1) := J(l)
        J(r + 1) := i
        k := k + 1

```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is (n^2) .

Example: Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J ₂	J ₁	J ₄	J ₃	J ₅
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J₂, as it can be completed within its deadline and contributes maximum profit.

- Next, J₁ is selected as it gives more profit compared to J₄.
- In the next clock, J₄ cannot be selected as its deadline is over, hence J₃ is selected as it executes within its deadline.
- The job J₅ is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J₂, J₁, J₄), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**

Optimal Merge Pattern

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time. If the numbers of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns. As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a **p-record file** and a **q-record file** requires possibly p+q record moves, the obvious choice being, merge the two smallest files together at each step. Two-way merge patterns can be represented by binary merge trees. Let us consider a set of n sorted files {f₁, f₂, f₃, ..., f_n}. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Algorithm: TREE (n)

```

for i := 1 to n - 1 do
    declare new node
    node.leftchild := least (list)
    node.rightchild := least (list)
    node.weight := ((node.leftchild).weight) + ((node.rightchild).weight)
    insert (list, node);
return least (list);

```

Example

Let us consider the given files, f_1 , f_2 , f_3 , f_4 and f_5 with **20, 30, 10, 5** and **30** number of elements respectively. If merge operations are performed according to the provided sequence, then

$$M1 = \text{merge } f1 \text{ and } f2 \Rightarrow 20 + 30 = 50$$

$$M2 = \text{merge } M1 \text{ and } f3 \Rightarrow 50 + 10 = 60$$

$$M3 = \text{merge } M2 \text{ and } f4 \Rightarrow 60 + 5 = 65$$

$$M4 = \text{merge } M3 \text{ and } f5 \Rightarrow 65 + 30 = 95$$

Hence, the total number of operations is **50 + 60 + 65 + 95 = 270**

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence: f_4 , f_3 , f_1 , f_2 , f_5

Hence, merge operations can be performed on this sequence

$$M1 = \text{merge } f4 \text{ and } f3 \Rightarrow 5 + 10 = 15$$

$$M2 = \text{merge } M1 \text{ and } f1 \Rightarrow 15 + 20 = 35$$

$$M3 = \text{merge } M2 \text{ and } f2 \Rightarrow 35 + 30 = 65$$

$$M4 = \text{merge } M3 \text{ and } f5 \Rightarrow 65 + 30 = 95$$

Therefore, the total number of operations is **15 + 35 + 65 + 95 = 210**

Obviously, this is better than the previous one.

In this context, we are now going to solve the problem using this algorithm.

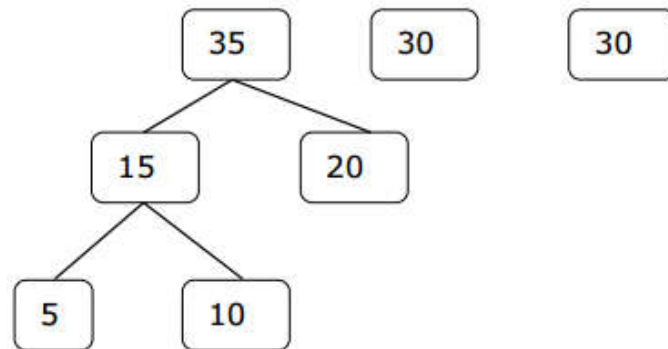
Initial Set



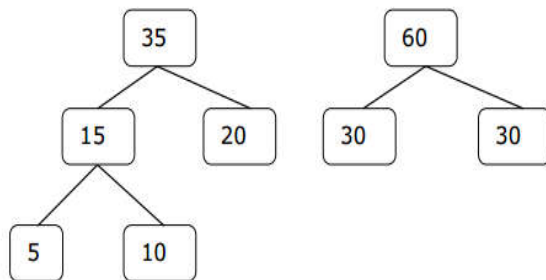
Step-1



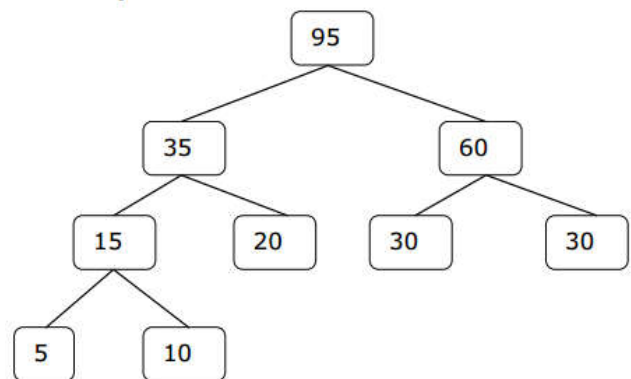
Step-2



Step-3



Step-4

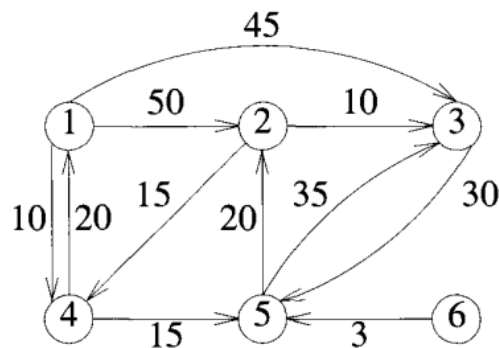


Hence, the solution takes $15 + 35 + 60 + 95 = 205$ number of comparisons.

SINGLE-SOURCE SHORTEST PATHS

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway.

Example:



(a) Graph

Path	Length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Graph and shortest paths from vertex 1 to all destinations

Dynamic Programming

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time. Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems** and **optimal substructure**.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists. For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems. For example, the Shortest Path problem has

the following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v , then the shortest path from u to v is the combination of the shortest path from u to x , and the shortest path from x to v . The standard All Pair Shortest Path algorithms like Floyd Warshall and Bellman-Ford are typical examples of Dynamic Programming.

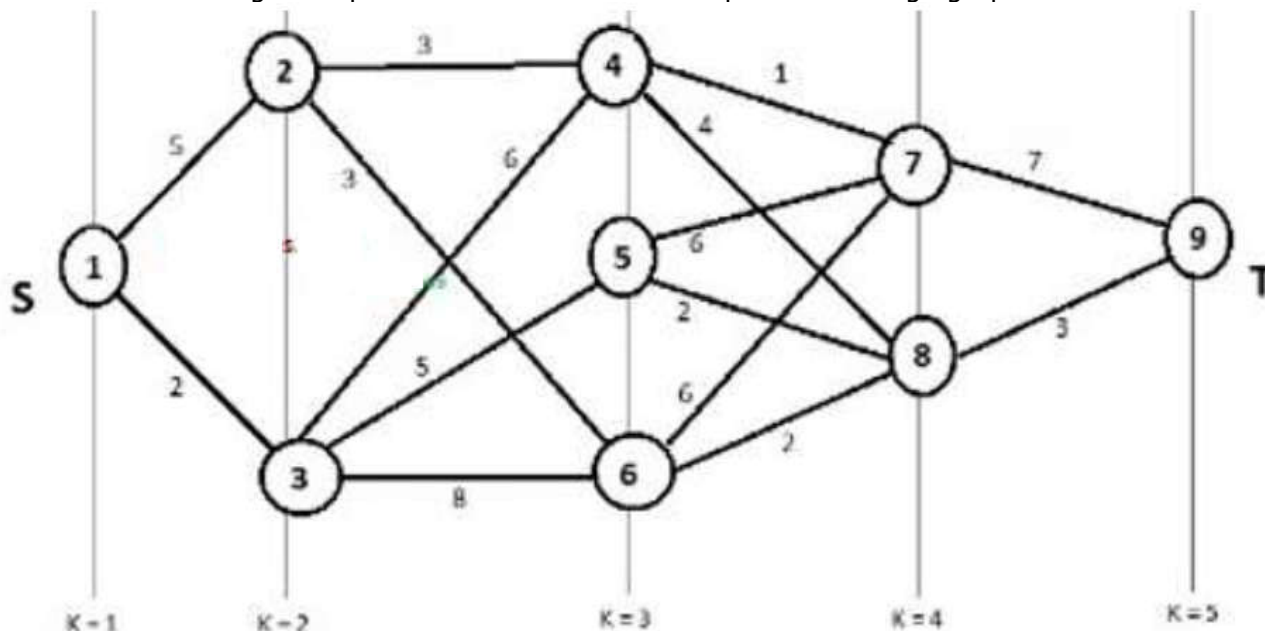
Multistage Graph

A multistage graph $G=(V,E)$ is a directed graph where vertices are partitioned into k (where $K>1$) number of disjoint subsets $S = \{s_1, s_2, \dots, s_k\}$ such that edges (u,v) is in E , then $u \in s_i$ and $v \in s_{i+1}$ for some subsets in the partition and $|s_1|=|s_k|=1$

The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**. G is usually assumed to be a weighted graph. In this graph, cost of an edge (i,j) is represented by $c(i,j)$. Hence, the cost of path from source s to t is the sum of costs of each edges in this path. The multistage graph problem is finding the path with minimum cost from source s to sink t .

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost (i, j) using the following steps:

Step-1: Cost $(K-2, j)$

In this step, three nodes (node 4, 5, 6) are selected as j . Hence, we have three options to choose the minimum cost at this step.

$$\text{Cost}(3,4) = \min\{C(4,7)+C(7,9), C(4,8)+C(8,9)\} = 7$$

$$\text{Cost}(3,5) = \min\{C(5,7)+C(7,9), C(5,8)+C(8,9)\} = 5$$

$$\text{Cost}(3,6) = \min\{C(6,7)+C(7,9), C(6,8)+C(8,9)\} = 5$$

Step-2: Cost $(K-3, j)$

Two nodes are selected as j because at stage $K-3=2$ there are two nodes 2 and 3. So, the value $i=2$ and $j=2$ and 3.

$$\text{Cost}(2,2) = \min\{c(2,4)+C(4,8)+C(8,9), C(2,6)+C(6,8), C(6,8), C(8,9)\} = 8$$

$$\text{Cost}(2,3) = \min\{C(3,4)+C(4,9)+C(8,9), C(3,5)+C(5,9)\} = 7$$

Step-3: Cost $(K-4, j)$

$$\text{Cost}(1,1) = \min\{C(1,2)+C(2,6)+C(6,8)+C(8,9), C(1,3)+C(3,6)+C(6,8)+C(8,9)\} = 13$$

Hence, the path having the minimum cost is **1→2→6→8→9**

All-Pairs Shortest Paths

Suppose we wish to compute the shortest-path distance between every pair of vertices in a directed graph G with n vertices and m edges. Of course, if G has no negative-weight edges, then we could run Dijkstra's algorithm from each vertex in G in turn. This approach would take $O(n(n + m) \log n)$ time, assuming G is represented using an adjacency list structure. In the worst case, this bound could be as large as $O(n^3 \log n)$. Likewise, if G contains no negative-weight cycles, then we could run the Bellman-Ford algorithm starting from each vertex in G in turn. This approach would run in $O(n^2 m)$ time, which, in the worst case, could be as large as $O(n^4)$. In this section, we consider algorithms for solving the all-pairs shortest path problem in $O(n^3)$ time, even if the digraph contains negative-weight edges (but not negative-weight cycles).

A Dynamic Programming Shortest-Path Algorithm

Let G be a given weighted directed graph. We number the vertices of G arbitrarily as $(v_1, v_2, \dots, v_n)$. As in any dynamic programming algorithm, the key construct in the algorithm is to define a parametrized cost function that is easy to compute and also allows us to ultimately compute a final solution. In this case, we use the cost function, $D_{i,j}^k$, which is defined as the distance from v_i to v_j using only intermediate vertices in the set $\{v_1, v_2, \dots, v_k\}$. Initially,

$$D_{i,j}^0 = \begin{cases} 0 & \text{if } i = j \\ w((v_i, v_j)) & \text{if } (v_i, v_j) \text{ is an edge in } \vec{G} \\ +\infty & \text{otherwise.} \end{cases}$$

Given this parametrized cost function $D_{i,j}^k$, and its initial value $D_{i,j}^0$, we can then easily define the value for an arbitrary $k > 0$ as

$$D_{i,j}^k = \min \{ D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1} \}$$

In other words, the cost for going from v_i to v_j using vertices numbered 1 through k is equal to the shorter of two possible paths. The first path is simply the shortest path from v_i to v_j using vertices numbered 1 through $k - 1$. The second path is the sum of the costs of the shortest path from v_i to v_k using vertices numbered 1 through $k - 1$ and the shortest path from v_k to v_j using vertices numbered 1 through $k - 1$. Moreover, there is no other shorter path from v_i to v_j using vertices of $\{v_1, v_2, \dots, v_k\}$ than these two. If there was such a shorter path and it excluded

v_k , then it would violate the definition of $D_{i,j}^{k-1}$ and if there was such a shorter path and it included v_k , then it would violate the definition of $D_{i,k}^{k-1}$ or $D_{k,j}^{k-1}$. In fact, note that this argument still holds even if there are negative cost edges in G , just so long as there are no negative cost cycles. In Algorithm below, we show how this cost-function definition allows us to build an efficient solution to the all-pairs shortest path problem. The running time for this dynamic programming algorithm is clearly $O(n^3)$.

Algorithm AllPairsShortestPaths($\vec{G}$):**Input:** A simple weighted directed graph $\vec{G}$ without negative-weight cycles**Output:** A numbering $v_1, v_2, \dots, v_n$ of the vertices of $\vec{G}$ and a matrix D , such that $D[i, j]$ is the distance from v_i to v_j in $\vec{G}$ let $v_1, v_2, \dots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$ **for** $i \leftarrow 1$ **to** n **do** **for** $j \leftarrow 1$ **to** n **do** **if** $i = j$ **then** $D^0[i, i] \leftarrow 0$ **if** (v_i, v_j) is an edge in $\vec{G}$ **then** $D^0[i, j] \leftarrow w((v_i, v_j))$ **else** $D^0[i, j] \leftarrow +\infty$ **for** $k \leftarrow 1$ **to** n **do** **for** $i \leftarrow 1$ **to** n **do** **for** $j \leftarrow 1$ **to** n **do** $D^k[i, j] \leftarrow \min\{D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]\}$ **return** matrix D^n

Algorithm: A dynamic programming algorithm to compute all-pairs shortest path distances in a digraph without negative cycles.

Matrix Chain-Products

Suppose we are given a collection of n two-dimensional matrices for which we wish to compute the product

$$A_0 \cdot A_1 \cdot A_2 \cdot \dots \cdot A_{n-1},$$

where A_i is a $d_i \times d_{i+1}$ matrix, for $i = 0, 1, 2, \dots, n - 1$. In the standard matrix multiplication algorithm (which is the one we will use), to multiply a $d \times e$ matrix A times an $e \times f$ matrix B , we compute the (i, j) entry of the $d \times f$ matrix product, C , as follows (see Figure below):

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] \cdot B[k, j].$$

Thus, the computation of a single entry of the product matrix $B \cdot C$ takes e (scalar) multiplications.

Overall, the computation of all the entries takes def (scalar) multiplications.

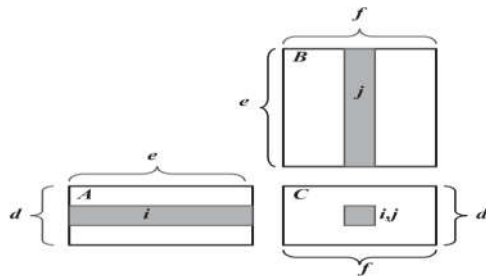


Figure 2.2: Multiplication of a $d \times e$ matrix, A , and an $e \times f$ matrix, B , which produces a $d \times f$ matrix, C .

This definition implies that matrix multiplication is associative, that is, it implies that $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Thus, we can parenthesize the expression for A any way we wish and we will end up with the same answer. We will not necessarily perform the same number of scalar multiplications in each parenthesization, however, as is illustrated in the following example.

Example: Let B be a 2×10 matrix, let C be a 10×50 matrix, and let D be a 50×20 matrix. Computing $B \cdot (C \cdot D)$ requires $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10400$ multiplications, whereas computing $(B \cdot C) \cdot D$ requires $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ multiplications.

The matrix chain-product problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed. Of course, one way to solve this problem is to simply enumerate all the possible ways of parenthesizing the expression for A and determine the number of multiplications performed by each one.

Unfortunately, the set of all different parenthesizations of the expression for A is equal in number to the set of all different binary trees that have n external nodes. This number is exponential in n. Thus, this straightforward (“brute-force”) algorithm runs in exponential time, for there are an exponential number of ways to parenthesize an associative arithmetic expression.

Algorithm MatrixChain($d_0, \dots, d_n$):

Input: Sequence $d_0, \dots, d_n$ of integers

Output: For $i, j = 0, \dots, n - 1$, the minimum number of multiplications $N_{i,j}$ needed to compute the product $A_i \cdot A_{i+1} \cdots A_j$, where A_k is a $d_k \times d_{k+1}$ matrix

for $i \leftarrow 0$ to $n - 1$ **do**

$N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ to $n - 1$ **do**

for $i \leftarrow 0$ to $n - b - 1$ **do**

$j \leftarrow i + b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ to $j - 1$ **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}.$

Algorithm: Dynamic programming algorithm for the matrix chain-product problem

Chapter Three: Graphs Algorithms

Connectivity information is present in a multitude of different applications, including social networks, road networks, computer games and puzzles, and computer networks. In computer networks, computers are connected together using routers, switches, and (wired or radio) communication links. Information flows in the form of individual packets, from one computer to another, by tracing out paths that hop through intermediate connections. Alternatively, in computer puzzle and game applications, game positions are connected by the transitions that are allowed between them. In either case, however, we might be interested in paths that can be mapped out in the connectivity structures that are represented in these applications. For instance, a path in a puzzle application could be a solution that starts from an initial game position and leads to a goal position.

In some applications, we might not really care whether we find a shortest path from a source to a destination, whereas in others we might be interested in finding the shortest paths possible. For example, in interactive applications, such as in online games or video chat sessions, we would naturally want packets to flow in our computer network along shortest paths, since we want to minimize any delays that occur between one person's action and the reaction of communicating partner. In other applications, such as in solving a maze puzzle, there may be only a single path between a source and destination, with several dead ends that need to be ruled out in order find this path. Thus, we would like to design algorithms that can efficiently identify paths in connectivity structures.

In addition to these motivating applications, connectivity information can be defined by all kinds of relationships that exist between pairs of objects. Additional examples include mapping (in geographic information systems), transportation (in road and flight networks), and electrical engineering (in circuits). The topic we study in this chapter—**graphs**—is therefore focused on representations and algorithms for dealing efficiently with such relationships. That is, a graph is a set of objects, called “vertices,” together with a collection of pair wise connections between them, which define “edges.”

Graph Terminology and Representations

A **graph** G is a set, V , of **vertices** and a collection, E , of pairs of vertices from V , which are called **edges**. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V . Incidentally, some people use different terminology for graphs and refer to what we call vertices as “nodes” and what we call edges as “arcs” or “ties.”

Edges in a graph are either **directed** or **undirected**. An edge (u, v) is said to be **directed** from u to v if the pair (u, v) is ordered, with u preceding v . An edge (u, v) is said to be **undirected** if the pair (u, v) is not ordered. Undirected edges are sometimes denoted with set notation, as $\{u, v\}$, but for simplicity we use the pair notation (u, v) , noting that in the undirected case (u, v) is the same as (v, u) . Graphs are typically visualized by drawing the vertices as circles or rectangles and the edges as segments or curves connecting pairs of these circles or rectangles.

Graph Traversal Techniques

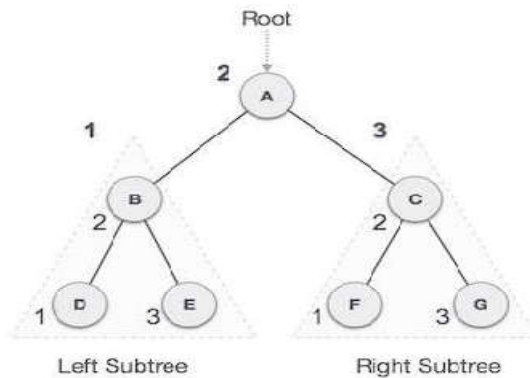
Tree traversal & applications

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- ✓ ***In-order Traversal***
- ✓ ***Pre-order Traversal***
- ✓ ***Post-order Traversal***

In-order Traversal

In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a sub tree itself. If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left sub tree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in order traversal of this tree will be –

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm

Until all nodes are traversed –

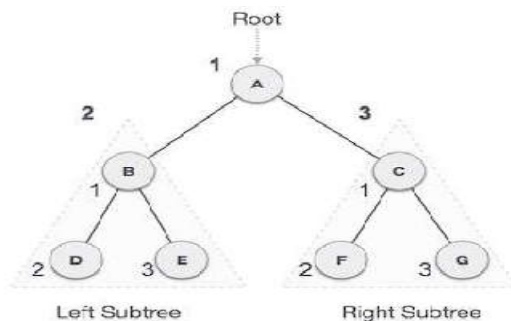
Step 1 - Recursively traverse left subtree.

Step 2 - Visit root node.

Step 3 - Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left sub tree and finally the right sub tree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left sub tree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Algorithm

Until all nodes are traversed –

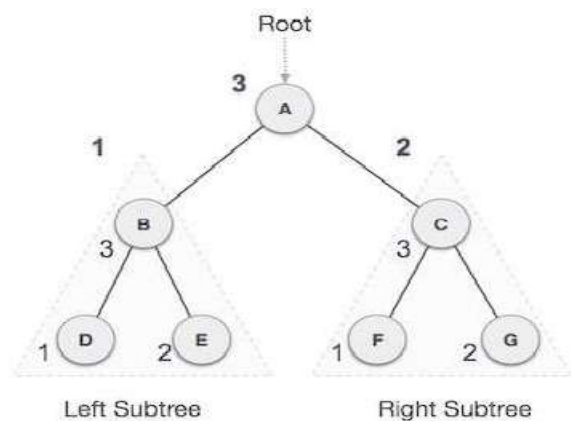
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left sub tree, then the right sub tree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left sub tree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

Until all nodes are traversed –

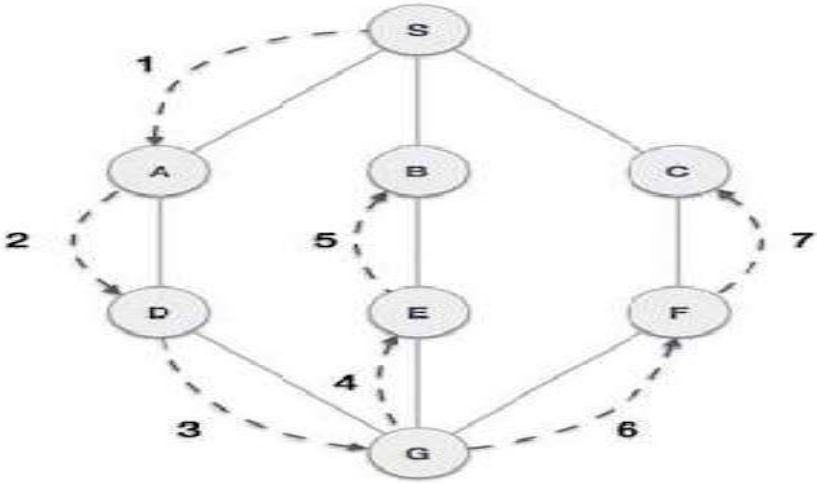
Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

Depth First Traversal

Depth First Search (DFS) algorithm traverses a graph in a depth-ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

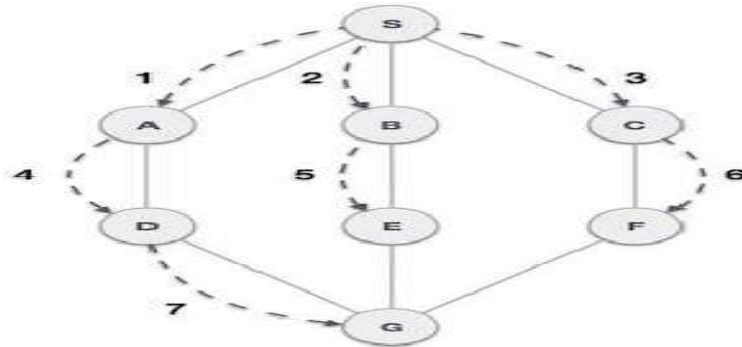


As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Breadth First Traversal

Breadth First Search (BFS) algorithm traverses a graph in a breadth-ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- ✓ **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- ✓ **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- ✓ **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty

Biconnected Components

Let G be a connected undirected graph. A separation edge of G is an edge whose removal disconnects G. A separation vertex is a vertex whose removal disconnects G. Separation edges and vertices correspond to single points of failure in a network; hence, we often wish to identify them. A

connected graph G is biconnected if, for any two vertices u and v of G , there are two disjoint paths between u and v , that is, two paths sharing no common edges or vertices, except u and v . A biconnected component of G is a subgraph satisfying one of the following:

- A subgraph of G that is biconnected and for which adding any additional vertices or edges of G would force it to stop being biconnected
- A single edge of G consisting of a separation edge and its endpoints.

If G is biconnected, it has one biconnected component: G itself. If G has no cycles, on the other hand, then each edge of G is a biconnected component. Biconnected components are important in computer networks, where vertices represent routers and edges represent connections, for even if a router in a biconnected component fails, messages can still be routed in that component using the remaining routers.

Let G be a connected graph. The following are equivalent:

1. G is biconnected.
2. For any two vertices of G , there is a simple cycle containing them.
3. G does not have separation vertices or separation edges.

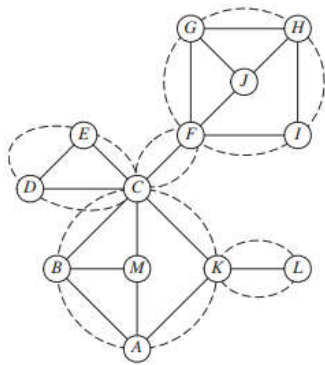
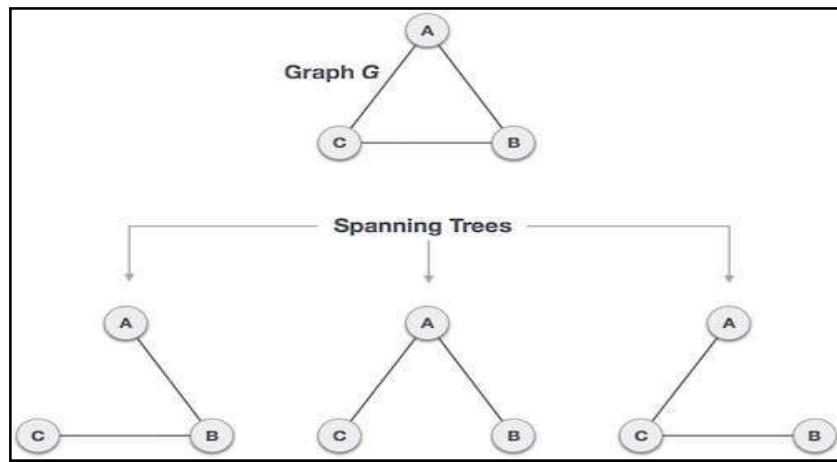


Figure: Biconnected components, shown circled with dashed lines. C , F , and K are separation vertices; (C, F) and (K, L) are separation edges.

Minimum spanning trees

A spanning tree is a subset of Graph G , which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected. By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

Mathematical Properties of Spanning Tree

- Spanning tree has $n-1$ edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum $e-n+1$ edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

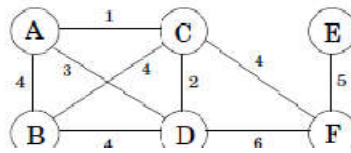
General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G -

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G , have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).

Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Suppose you are asked to network a collection of computers by linking selected pairs of them. This translates into a graph problem in which nodes are computers, undirected edges are potential links, and the goal is to pick enough of these edges that the nodes are connected. But this is not all; each link also has a maintenance cost, reflected in that edge's weight. What is the cheapest possible network?



Property 1 : Removing a cycle edge cannot disconnect a graph.

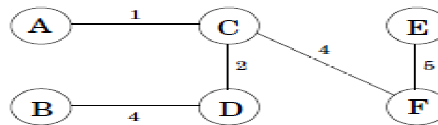
So the solution must be connected and acyclic: undirected graphs of this kind are called trees. The particular tree we want is the one with minimum total weight, known as the minimum spanning tree. Here is its formal definition.

Input: An undirected graph $G = (V, E)$; edge weights w_e .

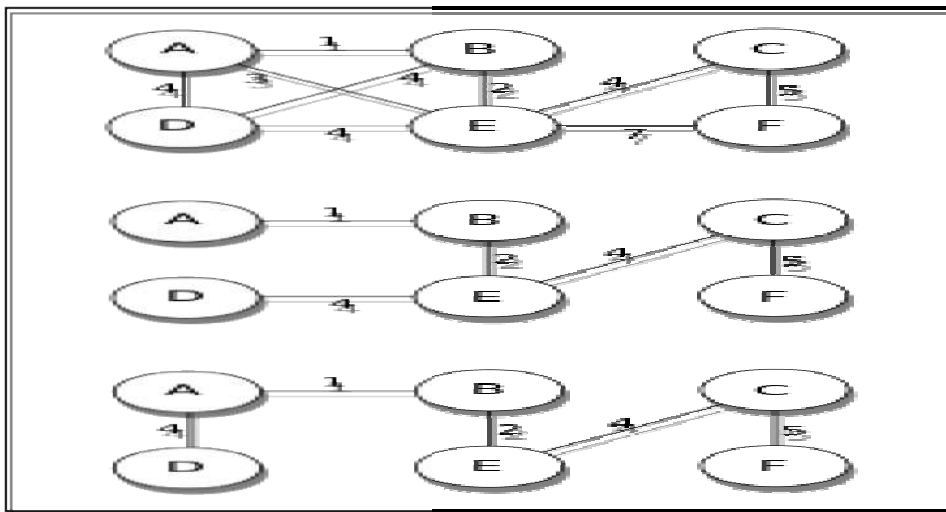
Output: A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

In the preceding example, the minimum spanning tree has a cost of 16:



However, this is not the only optimal solution. Can you spot another?



This figure shows there may be more than one minimum spanning tree in a graph. In the figure, the two trees below the graph are two possibilities of minimum spanning tree of the given graph

Minimum Spanning-Tree Algorithm

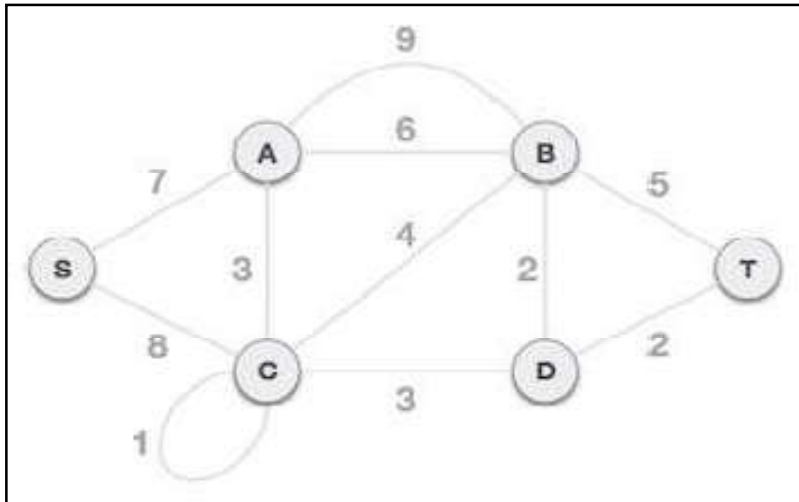
We shall learn about two most important spanning tree algorithms here –

- [Kruskal's Algorithm](#)
- [Prim's Algorithm](#)

Both are greedy algorithms

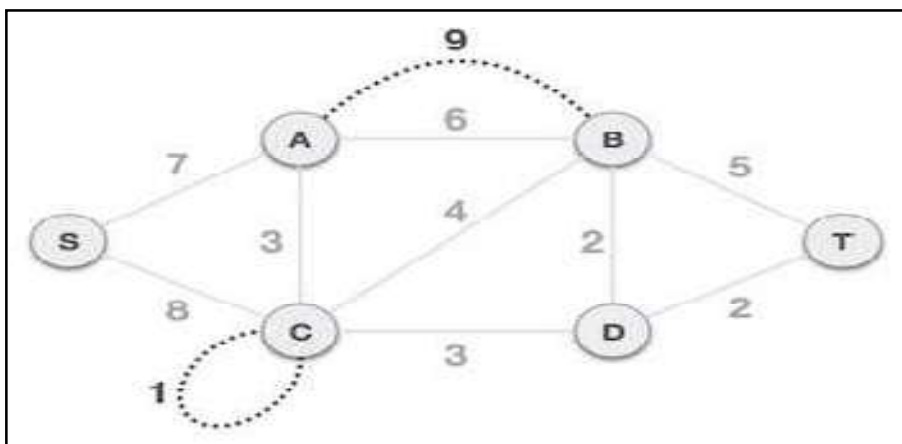
Kruskal's Spanning Tree Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties. To understand Kruskal's algorithm let us consider the following example –

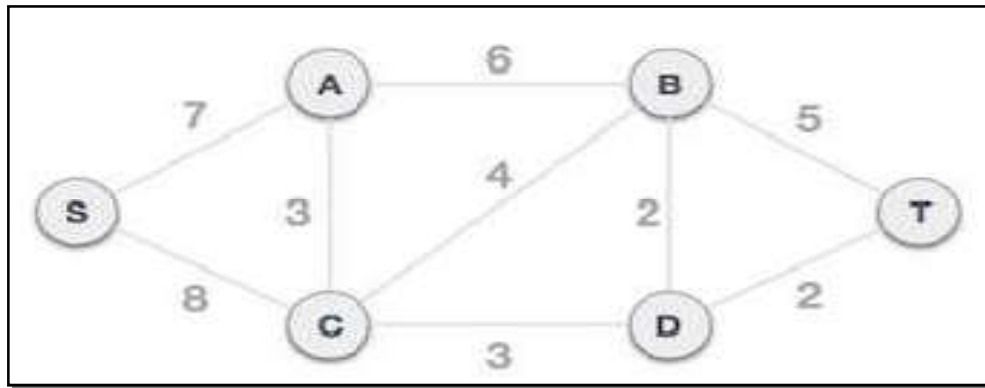


Step 1 - Remove all loops and parallel edges

Remove all loops and parallel edges from the given graph.



➔ In case of parallel edges, keep the one which has the least cost associated and remove all others.



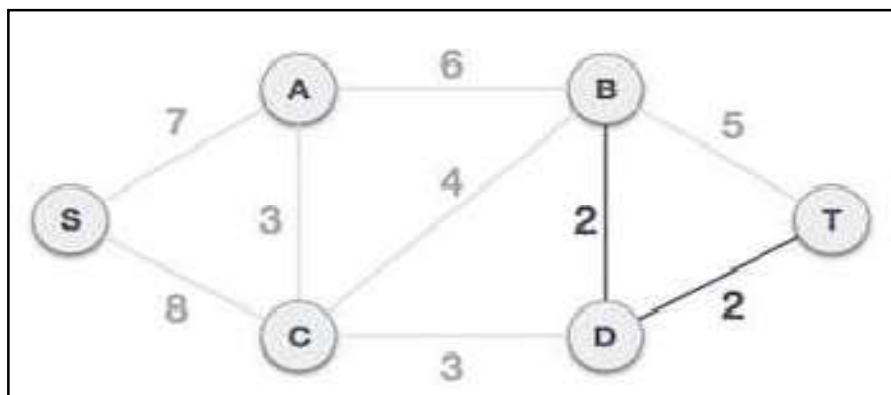
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

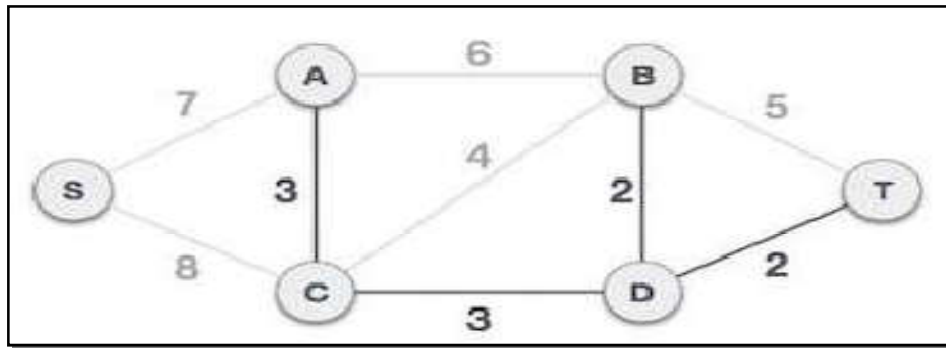
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

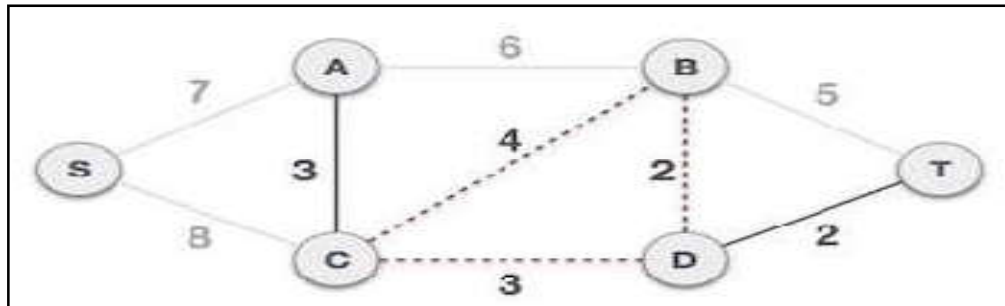
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning tree properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection. Next cost is 3, and associated edges are A,C and C,D. We add them again -



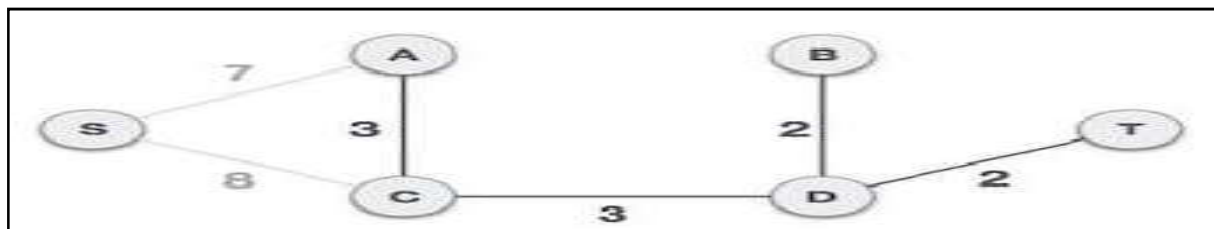
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



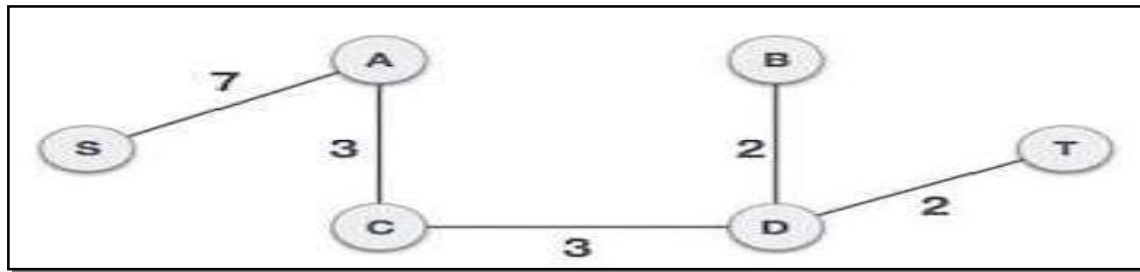
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7



→ By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Figure :- Kruskal's minimum spanning tree algorithm.

procedure kruskal (G, w)

Input: A connected undirected graph $G = (V, E)$ with edge weights w_e

Output: A minimum spanning tree defined by the edges X

for all $u \in V$:

 makeset(u)

$X = \{\}$

Sort the edges E by weight

for all edges $\{u, v\} \in E$, in increasing order of weight:

 if find(u) $\neq$ find(v):

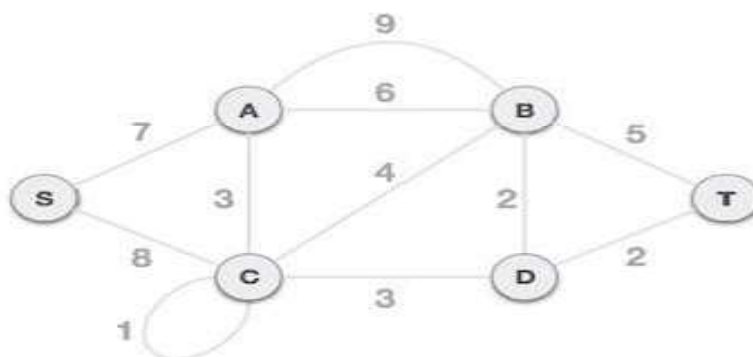
 add edge $\{u, v\}$ to X

 union(u, v)

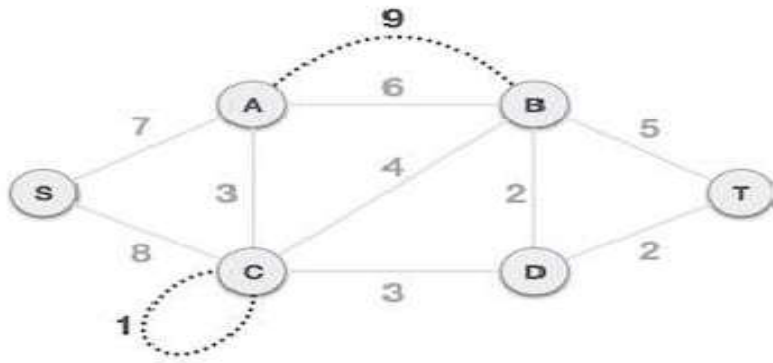
Prim's Spanning Tree Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms. Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

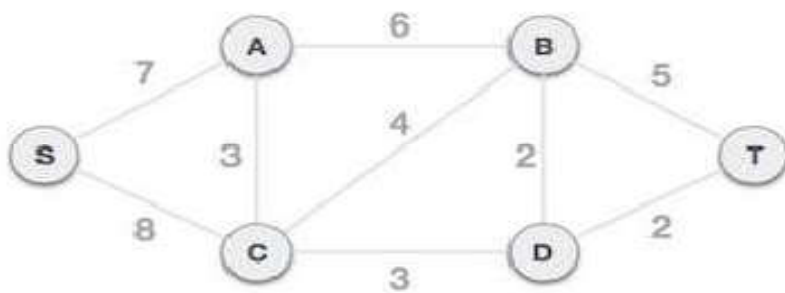
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

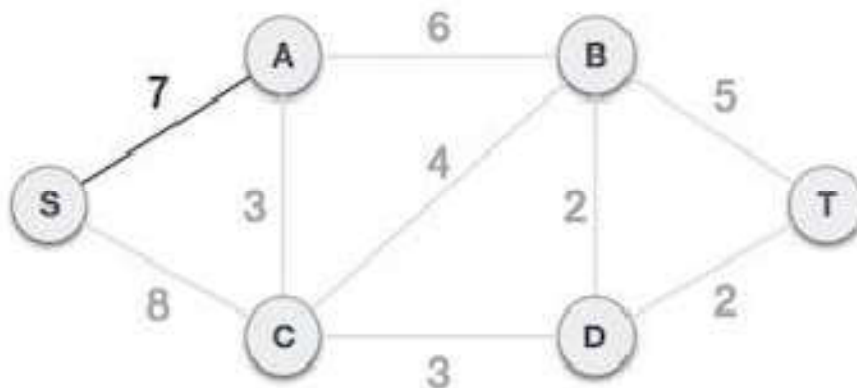


Step 2 - Choose any arbitrary node as root node

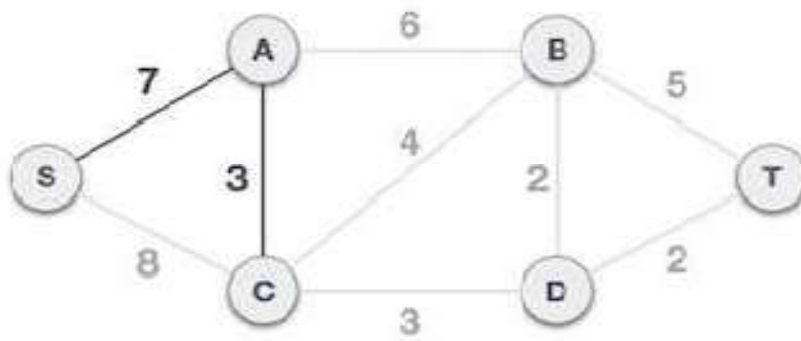
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

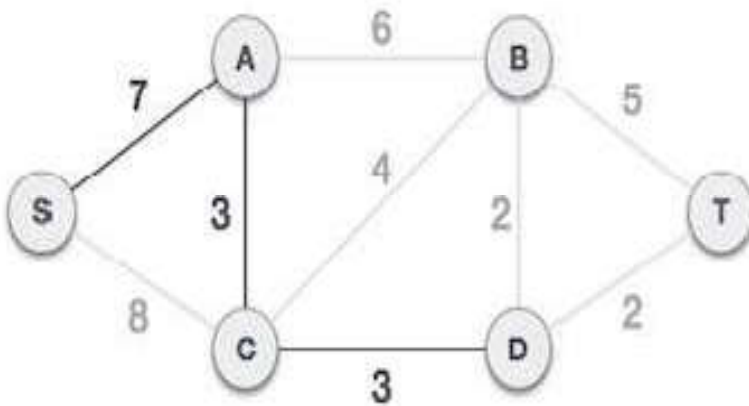
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



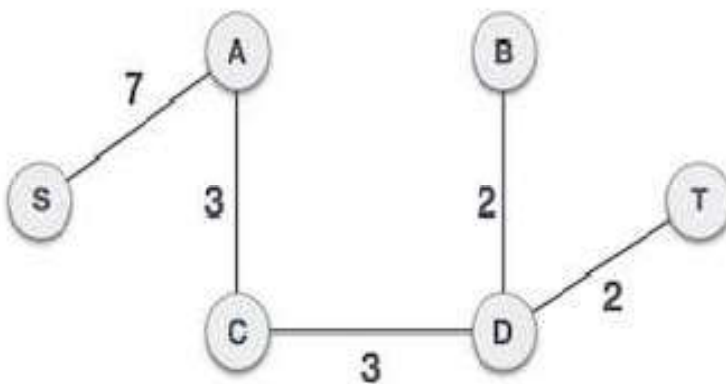
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Prim's minimum spanning tree algorithm.

```

procedure prim( $G, w$ )
Input:    A connected undirected graph  $G = (V, E)$  with edge weights  $w_e$ 
Output:   A minimum spanning tree defined by the array prev

for all  $u \in V$ :
    cost( $u$ ) =  $\infty$ 
    prev( $u$ ) = nil
Pick any initial node  $u_0$ 
cost( $u_0$ ) = 0

 $H = \text{makequeue}(V)$     (priority queue, using cost-values as keys)
while  $H$  is not empty:
     $v = \text{deletemin}(H)$ 
    for each  $\{v, z\} \in E$ :
        if cost( $z$ ) >  $w(v, z)$ :
            cost( $z$ ) =  $w(v, z)$ 
            prev( $z$ ) =  $v$ 
            decreasekey( $H, z$ )

```

Chapter 4

Selected Topics

String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem—called “string matching”—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

We formalize the string-matching problem as follows. We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ . For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a,b,c,\dots,z\}$. The character arrays P and T are often called *strings* of characters.

Referring to Figure 4.1, we say that pattern P *occurs with shift* s in text T (or, equivalently, that pattern P *occurs beginning at position* $s + 1$ in text T) if $0 \leq s \leq n - m$ and $T[s+1:s+m] = P[1:m]$ (that is, if $T[s+j] = P[j]$, for $1 \leq j \leq m$). If P occurs with shift s in T , then we call s a *valid shift*; otherwise, we call s an *invalid shift*. The *string-matching problem* is the problem of finding all valid shifts with which a given pattern P occurs in a given test T .

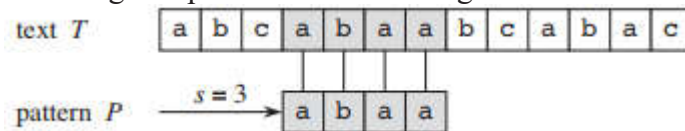


Figure 4.1 An example of the string-matching problem, where we want to find all occurrences of the pattern $P = abaa$ in the text $T = abcabaabcbabac$. The pattern occurs only once in the text, at shift $s = 3$, which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n - m + 1)m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Finite automaton	$O(m \Sigma)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 4.2 The string-matching algorithms in this chapter and their preprocessing and matching times

Except for the naive brute-force algorithm, each string-matching algorithm in this chapter performs some preprocessing based on the pattern and then finds all valid shifts; we call this latter phase “matching.” Figure 4.2 shows the preprocessing and matching times for each of the algorithms in this chapter. The total running time of each algorithm is the sum of the preprocessing and matching times. An interesting string-matching algorithm we call it Rabin and Karp has the worst-case running $\Theta((n-m+1)m)$. This algorithm is no better than that of the naive method, but it works much better on average and in practice. The other type of string matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern P in a text. This algorithm takes $O(m|\Sigma|)$ preprocessing time, but only $\Theta(n)$ matching time. Knuth-Morris-Pratt (or KMP) algorithm similar but much clever than finite automaton, has the same $\Theta(n)$ matching time, and it reduces the preprocessing time to only $\Theta(m)$.

The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n - m + 1$ possible values of s .

```
NAIVE-STRING-MATCHER (T,P)
1  $n = T.length$ 
2  $m = P.length$ 
3 for  $s=0$  to  $n - m$ 
4   if  $P[1..m] == T[s+1..s+m]$ 
5     print "Pattern occurs with shift"  $s$ 
```

Figure 4.3 portrays the naive string-matching procedure as sliding a “template” containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop of lines 3–5 considers each possible shift explicitly. The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s .

Procedure NAIVE-STRING-MATCHER takes time $O((n - m + 1)m)$, and this bound is tight in the worst case. For example, consider the text string a^n (a string of n a ’s) and the pattern a^m . For each of the $n - m + 1$ possible values of the shift s , the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus $\Theta((n - m + 1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. Because it requires no preprocessing, NAIVESTRING-MATCHER’s running time equals its matching time.

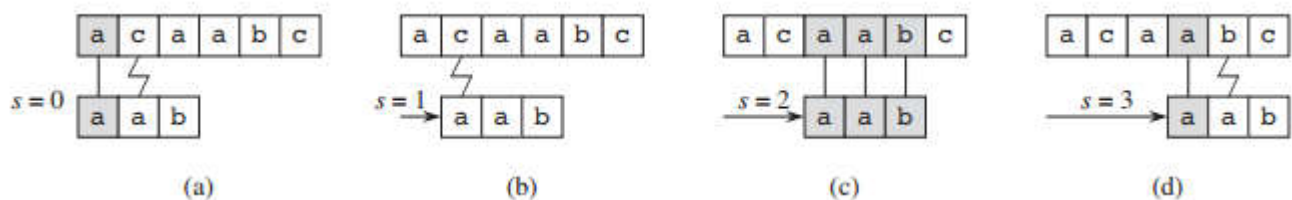


Figure 4.3 The operation of the naive string matcher for the pattern $P = aab$ and the text $T = acaabbc$. We can imagine the pattern P as a template that we slide next to the text. (a)–(d) The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c)

The Rabin-Karp algorithm

Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n - m + 1)m)$. Based on certain assumptions, however, its average-case running time is better.

String matching with finite automata

Many string-matching algorithms build a finite automaton—a simple machine for processing information—that scans the text string T for all occurrences of the pattern P . This section presents a method for building such an automaton. These string-matching automata are very efficient: they examine each text character *exactly once*, taking constant time per text character. The matching time used—after preprocessing the pattern to build the automaton—is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large.

We begin this section with the definition of a finite automaton. We then examine a special string-matching automaton and show how to use it to find occurrences of a pattern in a text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

Finite automata

A **finite automaton** M , illustrated in Figure 4.4, is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

Q is a finite set of **states**,

$q_0 \in Q$ is the **start state**,

$A \subseteq Q$ is a distinguished set of **accepting states**,

Σ is a finite **input alphabet**,

δ is a function from $Q \times \Sigma$ into Q , called the **transition function** of M .

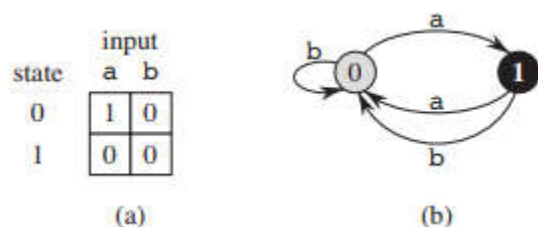


Figure 4.4 A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$. **(a)** A tabular representation of the transition function δ . **(b)** An equivalent state-transition diagram. State 1, shown blackend, is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that $\delta(1, b) = 0$. This automaton accepts those strings that end in an odd number of a 's. More precisely, it accepts a string x if and only if $x = yz$, where $y = \epsilon$ or y ends with a b , and $z = a^k$, where k is odd. For example, on input $abaaaa$, including the start state, this automaton enters the sequence of states $(0, 1, 0, 1, 0, 1)$, and so it accepts this input. For input $abbaa$, it enters the sequence of states $(0, 1; 0, 0, 1, 0)$, and so it rejects this input.

String-matching automata

FINITE-AUTOMATON-MATCHER (T, δ, m)

1 $n = T.length$

2 $q = 0$

3 **for** $i = 1$ **to** n

4 $q = \delta(q, T[i])$

5 **if** $q == m$

6 print "Pattern occurs with shift" $i - m$

From the simple loop structure of FINITE-AUTOMATON-MATCHER, we can easily see that its matching time on a text string of length n is $\Theta(n)$. This matching time, however, does not include the preprocessing time required to compute the transition function δ .

The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. This algorithm avoids computing the transition function δ altogether, and its matching time is $\theta(n)$ using just an auxiliary function π , which we precompute from the pattern in time $\Theta(m)$ and store in an array $\pi[1..m]$. The array π allows us to compute the transition function δ efficiently (in an amortized sense) "on the fly" as needed. Loosely speaking, for any state $q = 0, 1 \dots m$ and any character $a \in \Sigma$, the value $\pi[q]$ contains the information we need to compute $\delta(q, a)$ but that does not depend on a . Since the array π has only m entries, whereas δ has $\Theta(m|\Sigma|)$ entries, we save a factor of $|\Sigma|$ in the preprocessing time by computing π rather than δ .

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. We can take advantage of this information to avoid testing useless shifts in the naive pattern-matching algorithm and to avoid precomputing the full transition function δ for a string-matching automaton.

The pseudocode below gives the Knuth-Morris-Pratt matching algorithm as the procedure KMP-MATCHER. For the most part, the procedure follows from FINITE-AUTOMATON-MATCHER, as we shall see. KMP MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π .

KMP-MATCHER.T; P /

```
1  n = T.length
2  m = P.length
3   $\pi$  = COMPUTE-PREFIX-FUNCTION(P)
4  q = 0 // number of characters matched
5  for i = 1 to n // scan the text from left to right
6      while q > 0 and P[q + 1]  $\neq$  T[i]
7          q =  $\pi$ [q] // next character does not match
8      if P[q + 1] == T[i]
9          q = q + 1 // next character matches
10     if q == m // is all of P matched?
11         print "Pattern occurs with shift" i - m
12     q =  $\pi$ [q] // look for the next match
```

COMPUTE-PREFIX-FUNCTION (P)

```
1  m = P.length
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4  k = 0
5  for q = 2 to m
6      while k > 0 and P[k + 1]  $\neq$  P[q]
7          k =  $\pi$ [k]
8      if P[k + 1] == P[q]
9          k = k + 1
10      $\pi[q] = k$ 
11  return  $\pi$ 
```

These two procedures have much in common, because both match a string against the pattern P: KMP-MATCHER matches the text T against P, and COMPUTEPREFIX-FUNCTION matches P against itself. We begin with an analysis of the running times of these procedures. Proving these procedures correct will be more complicated.

Computational Geometry

Computational geometry is the branch of computer science that studies algorithms for solving geometric problems. In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VLSI design, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics. The input to a computational geometry problem is typically a description of a set of geometric objects,

such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

We look at a few computational-geometry algorithms in two dimensions, that is, in the plane. We represent each input object by a set of points $\{p_1, p_2, p_3, \dots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$. For example, we represent an n -vertex polygon P by a sequence $(p_0, p_1, p_2, \dots, p_{n-1})$ of its vertices in order of their appearance on the boundary of P . Computational geometry can also apply to three dimensions, and even higher-dimensional spaces, but such problems and their solutions can be very difficult to visualize. Even in two dimensions, however, we can see a good sample of computational-geometry techniques.