DESIGN AND ANALYSIS OF ALGORITHM(RCS4C002)

MODULE-1

Introduction

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**.

.

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 Algorithms

The main characteristics of algorithms are as follows −

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

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 are 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**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

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.

To measure resource consumption of an algorithm, different strategies are used as discussed in this chapter.

Asymptotic Analysis

The asymptotic behavior of a function ***f(n)*** refers to the growth of ***f(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 $f(n)=d*n+k$f(n)=d∗n+k is always asymptotically better than a quadratic one, $f(n)=c.n_2+q$f(n)=c.n2+q.

To measure resource consumption of an algorithm, different strategies are used as discussed in this chapter.

Solving Recurrence Equations

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 **θ(1)**. If the division of the problem yields a number of sub-problems with size nbnb.

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 −

$$T(n)=\{\theta(1) aT(nb)+D(n)+C(n) if n\leqslant c otherwise T(n)=\{\theta(1) if n\leqslant c aT(nb)+D(n)+C(n) otherwise$$

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.

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 **T(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.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh
- **ω** − Little omega

O: Asymptotic Upper Bound

'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

$f(n)\leqslant c.g(n)$f(n)⩽c.g(n) for $n>n_0$n>n0 in all case

Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$f(n)=4.n3+10.n2+5.n+1
Considering $g(n)=n^3$g(n)=n3,
$f(n)\leqslant 5.g(n)$f(n)⩽5.g(n) for all the values of $n>2$n>2
Hence, the complexity of ***f(n)*** can be represented as $O(g(n))$O(g(n)), i.e. $O(n^3)$O(n3)

Ω: Asymptotic Lower Bound

We say that $f(n)=\Omega(g(n))$f(n)=Ω(g(n)) when there exists constant **c** that $f(n)\geqslant c.g(n)$f(n)⩾c.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$f(n)=4.n3+10.n2+5.n+1.
Considering $g(n)=n^3$g(n)=n3, $f(n)\geqslant 4.g(n)$f(n)⩾4.g(n) for all the values of $n>0$n>0.
Hence, the complexity of ***f(n)*** can be represented as $\Omega(g(n))$Ω(g(n)), i.e. $\Omega(n^3)$Ω(n3)

θ: Asymptotic Tight Bound

We say that $f(n)=\theta(g(n))$f(n)=θ(g(n)) when there exist constants **c_1** and **c_2** that $c_1.g(n)\leqslant f(n)\leqslant c_2.g(n)$c1.g(n)⩽f(n)⩽c2.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer.

This means function **g** is a tight bound for function **f**.

Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$f(n)=4.n3+10.n2+5.n+1
Considering $g(n)=n^3$g(n)=n3, $4.g(n)\leqslant f(n)\leqslant 5.g(n)$4.g(n)⩽f(n)⩽5.g(n) for all the large values of **n**.
Hence, the complexity of ***f(n)*** can be represented as $\theta(g(n))$θ(g(n)), i.e. $\theta(n^3)$θ(n3).

O - Notation

The asymptotic upper bound provided by **O-notation** may or may not be asymptotically tight. The bound $2.n^2=O(n^2)$2.n2=O(n2) is asymptotically tight, but the bound $2.n=O(n^2)$2.n=O(n2) is not.

We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define ***o(g(n))*** (little-oh of g of n) as the set ***f(n) = o(g(n))*** for any positive constant $c>0$c>0 and there exists a value $n_0>0$n0>0, such that $0\leqslant f(n)\leqslant c.g(n)$0⩽f(n)⩽c.g(n).

Intuitively, in the **o-notation**, the function ***f(n)*** becomes insignificant relative to ***g(n)*** as **n** approaches infinity; that is,

$$\lim_{n\to\infty}(f(n)g(n))=0$$ limn→∞(f(n)g(n))=0

Example

Let us consider the same function, $f(n)=4.n^3+10.n^2+5.n+1$f(n)=4.n3+10.n2+5.n+1
Considering $g(n)=n^4$g(n)=n4,

$$\lim_{n\to\infty}(4.n^3+10.n^2+5.n+1n^4)=0$$ limn→∞(4.n3+10.n2+5.n+1n4)=0

Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$o(g(n)), i.e. $o(n^4)$o(n4).

ω – Notation

We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define ***ω(g(n))*** (little-omega of g of n) as the set ***f(n) = ω(g(n))*** for any positive constant ***C > 0*** and there exists a value $n_0>0$n0>0, such that $0\leqslant c.g(n)<f(n)$0⩽c.g(n)<f(n).
For example, $n^2 2=\omega(n)$n22=ω(n), but $n^2 2\neq\omega(n^2)$n22≠ω(n2). The relation $f(n)=\omega(g(n))$f(n)=ω(g(n)) implies that the following limit exists

$$\lim_{n\to\infty}(f(n)g(n))=\infty$$ limn→∞(f(n)g(n))=∞

That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Example

Let us consider same function, $f(n)=4.n^3+10.n^2+5.n+1$f(n)=4.n3+10.n2+5.n+1
Considering $g(n)=n^2$g(n)=n2,

$$\lim_{n\to\infty}(4.n^3+10.n^2+5.n+1n^2)=\infty$$ limn→∞(4.n3+10.n2+5.n+1n2)=∞

Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$o(g(n)), i.e. $\omega(n^2)$ω(n2).

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 **T(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.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh

- **ω** − Little omega

O: Asymptotic Upper Bound

'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as **n₀** and a positive constant **c** such that –

$f(n) \leqslant c.g(n)$ for $n > n_0$n>n0

in all caseHence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

Example

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

Considering $g(n) = n^3$g(n)=n3,

f(n)⩽5.g(n) for all the values of $n > 2$n>2

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

Ω: Asymptotic Lower Bound

We say that

$f(n) = \Omega(g(n))$ when there exists constant **c** that $f(n) \geqslant c.g(n)$ for all sufficiently

large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

Example

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

Considering g(n)=n3, $f(n) \geqslant 4$ for all the values of $n > 0$.

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

θ: Asymptotic Tight Bound

We say that $f(n) = \theta(g(n))$ when there exist constants **c₁** and **c₂** that $c_1.g(n) \leqslant f(n) \leqslant c_2.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer.

This means function **g** is a tight bound for function **f**.

Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$
Considering $g(n)=n^3$, $4.g(n)\leqslant f(n)\leqslant 5.g(n)$ for all the large values of **n**.
Hence, the complexity of ***f(n)*** can be represented as $\theta(g(n))$θ(g(n)), i.e. $\theta(n^3)$θ(n3).

O - Notation

The asymptotic upper bound provided by **O-notation** may or may not be asymptotically tight. The bound $2.n^2=O(n^2)$ is asymptotically tight, but the bound $2.n=O(n^2)$ is not.

We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define ***o(g(n))*** (little-oh of g of n) as the set ***f(n) = o(g(n))*** for any positive constant $c>0$ and there exists a value $n_0>0$, such that $0\leqslant f(n)\leqslant c.g(n)$

Intuitively, in the **o-notation**, the function ***f(n)*** becomes insignificant relative to ***g(n)*** as **n** approaches infinity; that is,

$$\mathrm{Lim}_{n\to\infty}(f(n)/g(n))$$

MODULE-2

Binary search

Binary search can be performed on a sorted array. In this approach, the index of an element **x** is determined if the element belongs to the list of elements. If the array is unsorted, linear search is used to determine the position.

Solution

In this algorithm, we want to find whether element **x** belongs to a set of numbers stored in an array ***numbers[]***. Where ***l*** and ***r*** represent the left and right index of a sub-array in which searching operation should be performed.

Analysis

Linear search runs in ***O(n)*** time. Whereas binary search produces the result in ***O(log n)*** time

Let **T(n)** be the number of comparisons in worst-case in an array of **n** elements.

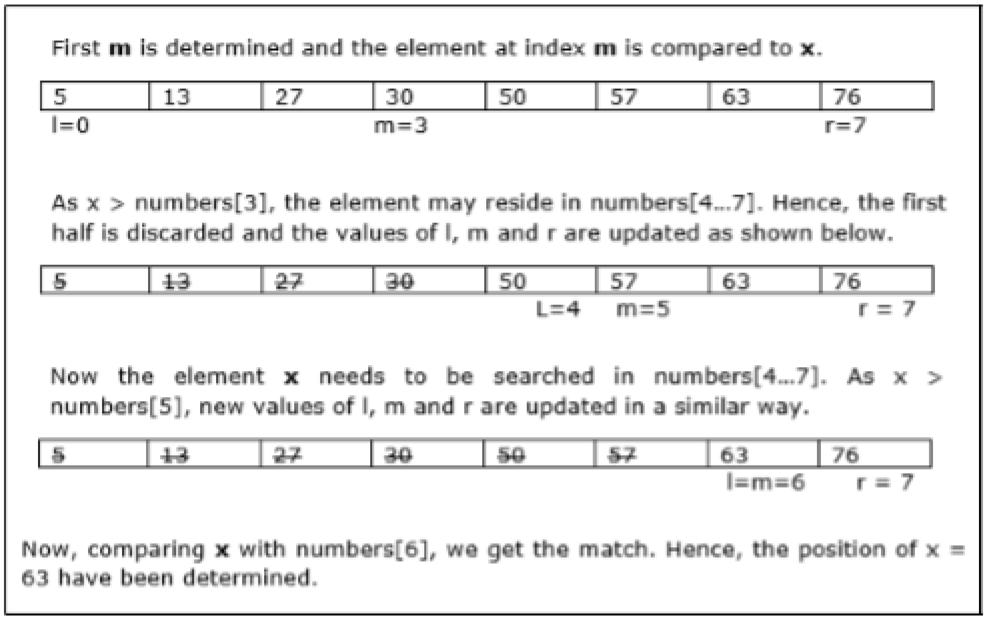
Hence,

$$T(n)=\{0T(\tfrac{n}{2})+1ifn=1otherwise$$ T(n)={0ifn=1T(n2)+1otherwise

Using this recurrence relation $T(n)=\log n$T(n)=logn.
Therefore, binary search uses $O(\log n)$O(logn) time.

Example

In this example, we are going to search element 63.



First **m** is determined and the element at index **m** is compared to **x**.

5	13	27	30	50	57	63	76
l=0			m=3				r=7

As x > numbers[3], the element may reside in numbers[4...7]. Hence, the first half is discarded and the values of l, m and r are updated as shown below.

~~5~~	~~13~~	~~27~~	~~30~~	50	57	63	76
				L=4	m=5		r = 7

Now the element **x** needs to be searched in numbers[4...7]. As x > numbers[5], new values of l, m and r are updated in a similar way.

~~5~~	~~13~~	~~27~~	~~30~~	~~50~~	~~57~~	63	76
						l=m=6	r = 7

Now, comparing **x** with numbers[6], we get the match. Hence, the position of x = 63 have been determined.

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm

Greedy algorithms have the following five components −

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution.
- **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function** − Used to assign a value to a solution or a partial solution.
- **A solution function** − Used to indicate whether a complete solution has been reached.

Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Where Greedy Approach Fails

In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.

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 subproblem 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 $\boldsymbol{W}$ into his knapsack. There are n items available in the store and weight of $\boldsymbol{i}^{th}$ item is $\boldsymbol{w}_i$ and its profit is $\boldsymbol{p}_i$. 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 $w_i > 0$wi>0
- Profit for **i**th item $p_i > 0$pi>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 $\boldsymbol{x_i}$ of **i**th item.

$$0 \leqslant x_i \leqslant 1$$ 0⩽xi⩽1

The **i**th item contributes the weight $x_i.w_i$xi.wi to the total weight in the knapsack and profit $x_i.p_i$xi.pi to the total profit.

Hence, the objective of this algorithm is to

$$\text{maximize} \sum_{n=1}^{n} (x_i.p_i)$$ maximize∑n=1n(xi.pi)

subject to constraint,

$$\sum_{n=1}^{n} (x_i.w_i) \leqslant W$$ ∑n=1n(xi.wi)⩽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

$$\sum_{n=1}^{n} (x_i.w_i) = W$$ ∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$piwi, so that $\frac{p_{i+1}}{w_{i+1}}$pi+1wi+1 ≤ $\frac{p_i}{w_i}$piwi . 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] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of $p_i w_i$piwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

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 ($p_i w_i$)(piwi)	7	10	6	5

As the provided items are not sorted based on $p_i w_i$piwi. 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 ($p_i w_i$)(piwi)	10	7	6	5

Solution

After sorting all the items according to $p_i w_i$piwi. 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.

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 number 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 **{f_1, f_2, f_3, …, 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);
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

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

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 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

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

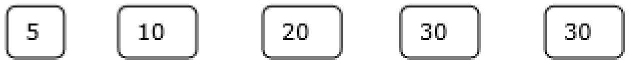
Therefore, the total number of operations is

15 + 35 + 65 + 95 = 210

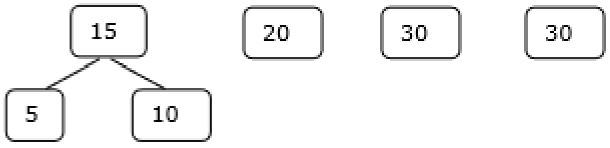
Obviously, this is better than the previous one.

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

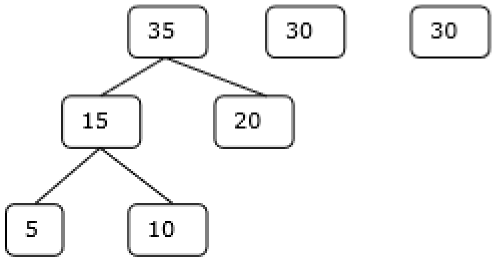
Initial Set



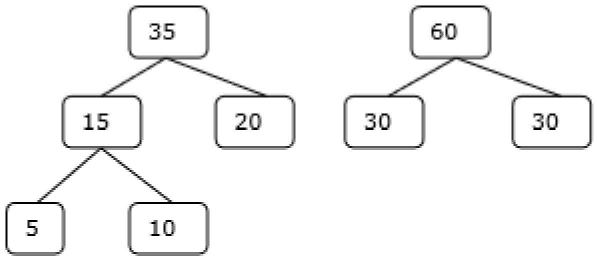
Step-1



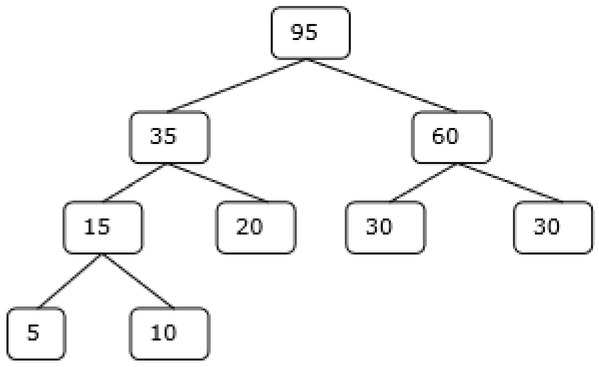
Step-2



Step-3



Step-4



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

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.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

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.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

we have discussed Fractional Knapsack problem using Greedy approach. We have shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem and its analysis.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	24	18	18	10

Weight	24	10	10	7

Without considering the profit per unit weight (***p_i/w_i***), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio *p_i/w_i*. Let us consider that the capacity of the knapsack is *W* = 60 and the items are as shown in the following table.

Item	**A**	**B**	**C**
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a max$_i$mal weight of ***W*** into his knapsack. There are ***n*** items and weight of **i**th item is ***w_i*** and the profit of selecting this item is ***p_i***. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then ***S' = S - {i}*** is an optimal solution for ***W - w_i*** dollars and the value to the solution ***S*** is ***V_i*** plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maxᵢmum weight **w**.

The algorithm takes the following inputs

- The maxᵢmum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if $w_i \leq w$ then
         if $v_i$ + c[i-1, $w-w_i$] then
            c[i, w] = $v_i$ + c[i-1, $w-w_i$]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

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

Subsequence

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

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

Common Subsequence

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

Longest Common Subsequence

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

The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

Naïve Method

Let ***X*** be a sequence of length ***m*** and ***Y*** a sequence of length ***n***. Check for every subsequence of ***X*** whether it is a subsequence of ***Y***, and return the longest common subsequence found.

There are ***2^m*** subsequences of ***X***. Testing sequences whether or not it is a subsequence of ***Y*** takes ***O(n)*** time. Thus, the naïve algorithm would take ***$O(n2^m)$*** time.

Dynamic Programming

Let $X = < x_1, x_2, x_3, \dots, x_m >$ and $Y = < y_1, y_2, y_3, \dots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used.

In this procedure, table ***C[m, n]*** is computed in row major order and another table ***B[m,n]*** is computed to construct optimal solution.

Algorithm: LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
   C[i, 0] := 0
for j = 1 to n do
   C[0, j] := 0
for i = 1 to m do
   for j = 1 to n do
      if $x_i = y_j$
         C[i, j] := C[i - 1, j - 1] + 1
         B[i, j] := 'D'
      else
         if C[i -1, j] ≥ C[i, j -1]
            C[i, j] := C[i - 1, j] + 1
            B[i, j] := 'U'
         else
         C[i, j] := C[i, j - 1]
         B[i, j] := 'L'
return C and B
```

Algorithm: Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
   return
if B[i, j] = 'D'
```

```
   Print-LCS(B, X, i-1, j-1)
   Print(x_i)
else if B[i, j] = 'U'
   Print-LCS(B, X, i-1, j)
else
   Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of **X** and **Y**.

Analysis

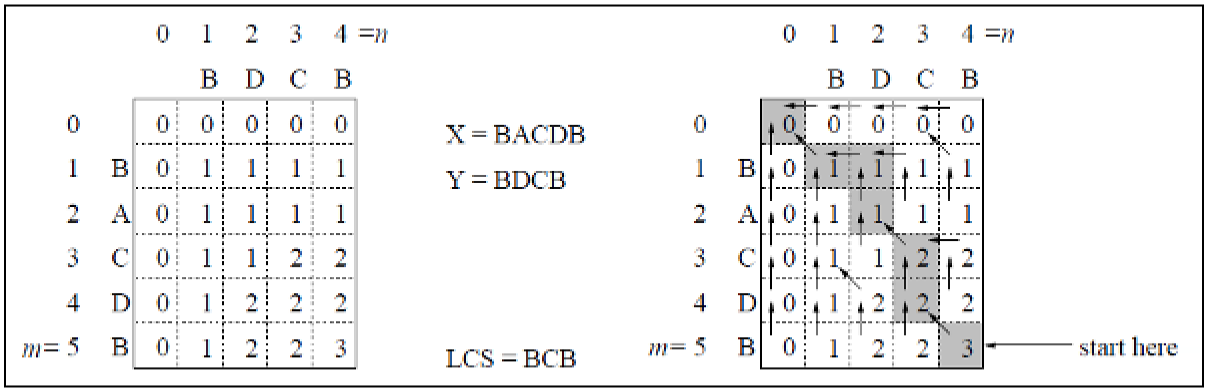
To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times. Hence, the complexity of the algorithm is *O(m, n)*, where ***m*** and ***n*** are the length of two strings.

Example

In this example, we have two strings ***X = BACDB*** and ***Y = BDCB*** to find the longest common subsequence.

Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).

In table B, instead of 'D', 'L' and 'U', we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



		0	1	2	3	4 =*n*
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m = 5	B	0	1	2	2	3

A **spanning tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

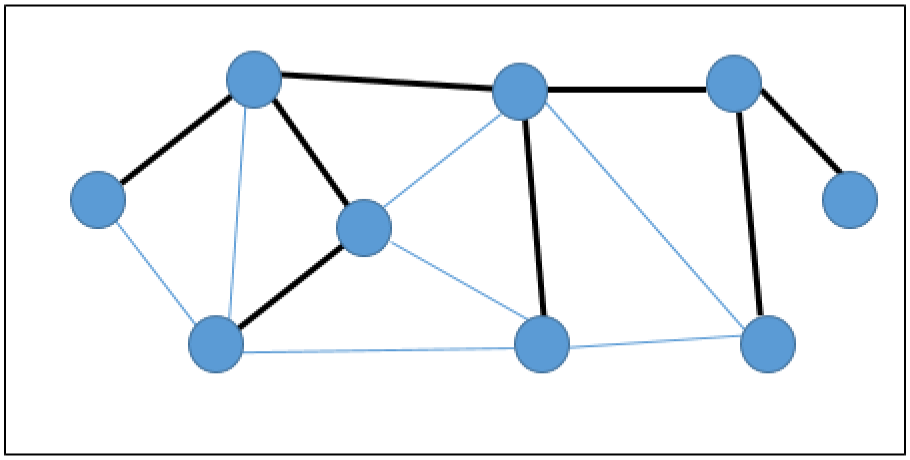
If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

Example

In the following graph, the highlighted edges form a spanning tree.

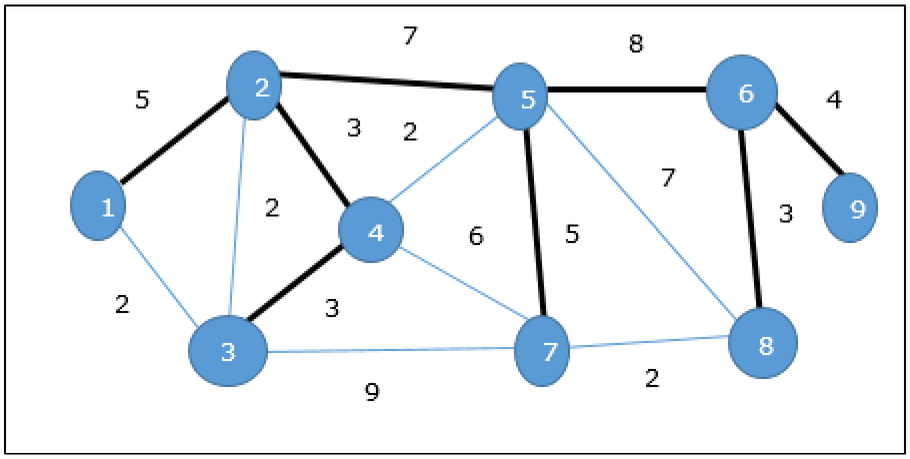


Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are **n** number of vertices, the spanning tree should have ***n* - 1** number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.



In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is (5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38.

We will use Prim's algorithm to find the minimum spanning tree.

Prim's Algorithm

Prim's algorithm is a greedy approach to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.

```
Algorithm: MST-Prim's (G, w, r)
for each u є G.V
   u.key = ∞
   u.∏ = NIL
r.key = 0
Q = G.V
while Q ≠ Φ
   u = Extract-Min (Q)
   for each v є G.adj[u]
      if each v є Q and w(u, v) < v.key
         v.∏ = u
         v.key = w(u, v)
```

The function Extract-Min returns the vertex with minimum edge cost. This function works on min-heap.

Example

Using Prim's algorithm, we can start from any vertex, let us start from vertex **1**.

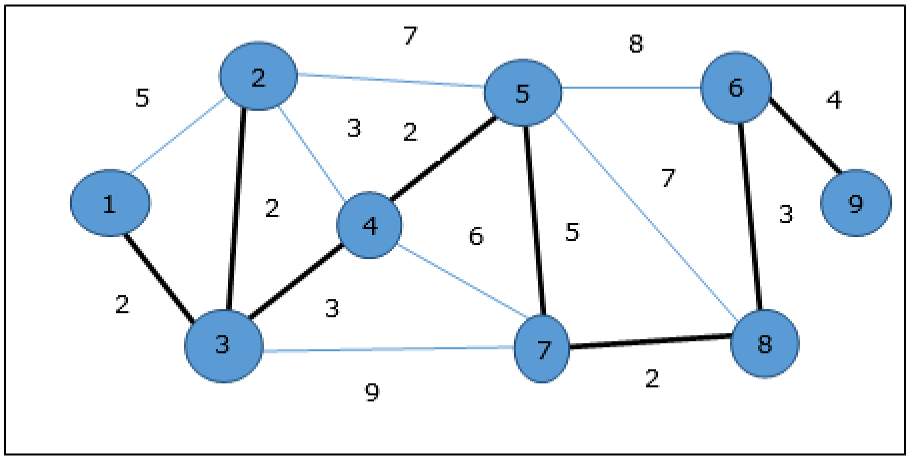
Vertex **3** is connected to vertex **1** with minimum edge cost, hence edge **(1, 2)** is added to the spanning tree.

Next, edge **(2, 3)** is considered as this is the minimum among edges **{(1, 2), (2, 3), (3, 4), (3, 7)}**.

In the next step, we get edge **(3, 4)** and **(2, 4)** with minimum cost. Edge **(3, 4)** is selected at random.

In a similar way, edges **(4, 5), (5, 7), (7, 8), (6, 8)** and **(6, 9)** are selected. As all the vertices are visited, now the algorithm stops.

The cost of the spanning tree is (2 + 2 + 3 + 2 + 5 + 2 + 3 + 4) = 23. There is no more spanning tree in this graph with cost less than **23**.



Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., *w(u, v)* ≥ 0 for each edge *(u, v) Є E*).

In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

```
Algorithm: Dijkstra's-Algorithm (G, w, s)
for each vertex v Є G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v Є G.adj[u]
      if v.d > u.d + w(u, v)
         v.d := u.d + w(u, v)
         v.∏ := u
```

Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is ***O(V^2 + E)***.

In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.

Example

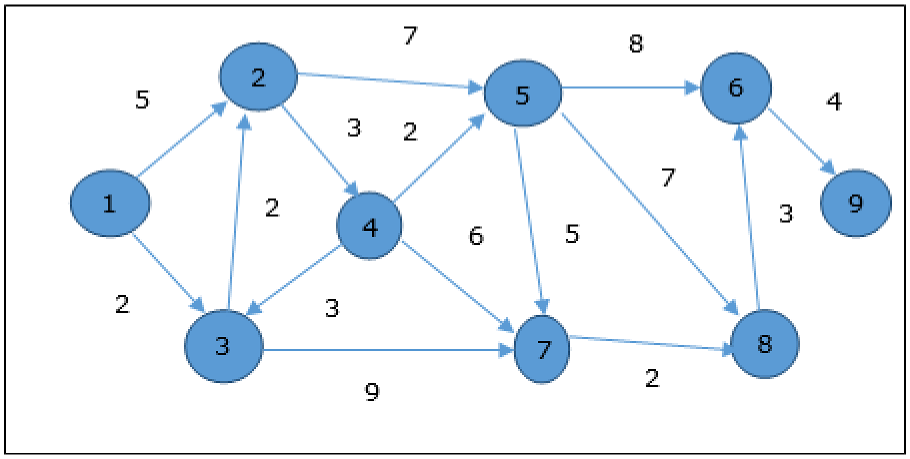
Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



Bellman Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph ***G* = (*V*, *E*)** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

```
Algorithm: Bellman-Ford-Algorithm (G, w, s)
for each vertex v Є G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
   for each edge (u, v) Є G.E
      if v.d > u.d + w(u, v)
         v.d := u.d +w(u, v)
         v.∏ := u
for each edge (u, v) Є G.E
   if v.d > u.d + w(u, v)
      return FALSE
return TRUE
```

Analysis

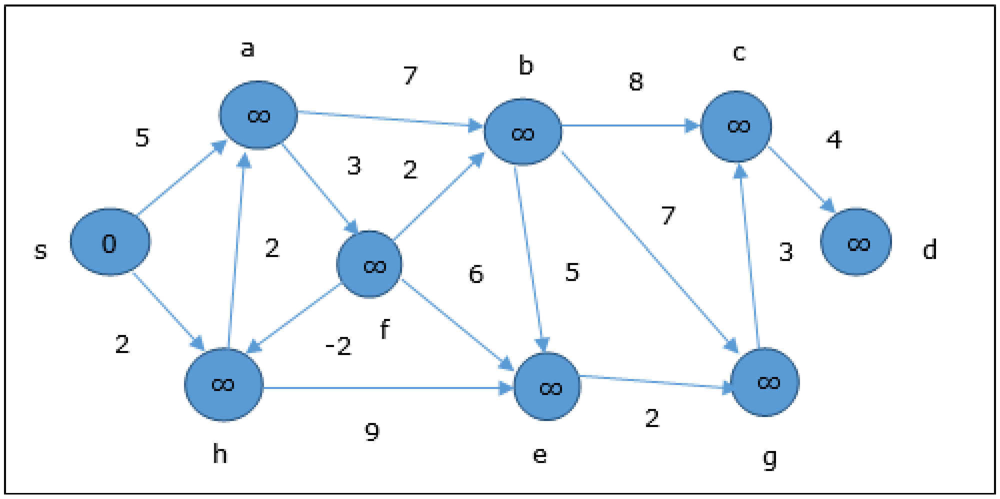
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs **|*V* - 1|** passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

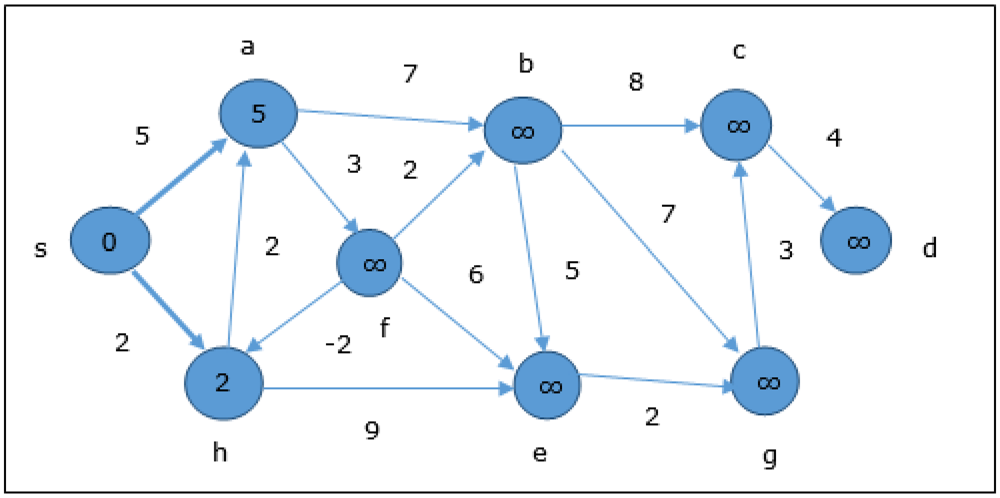
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

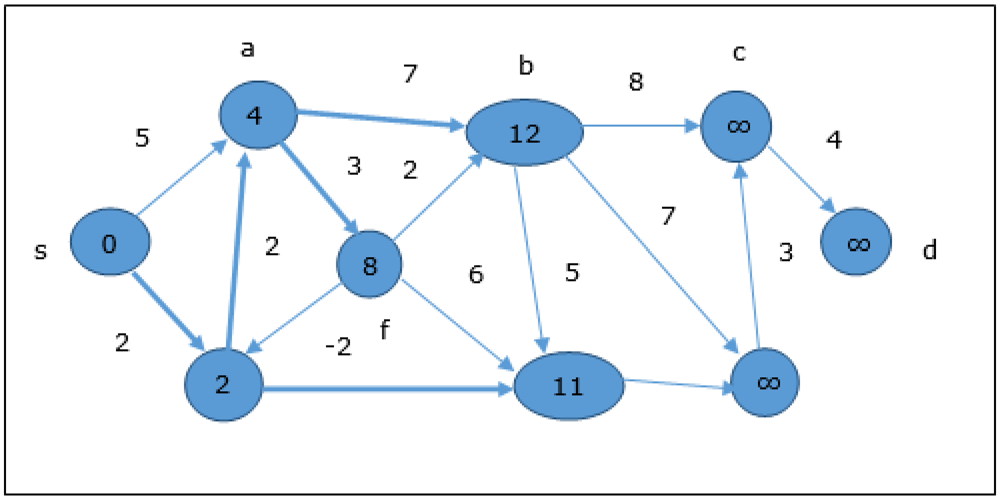
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



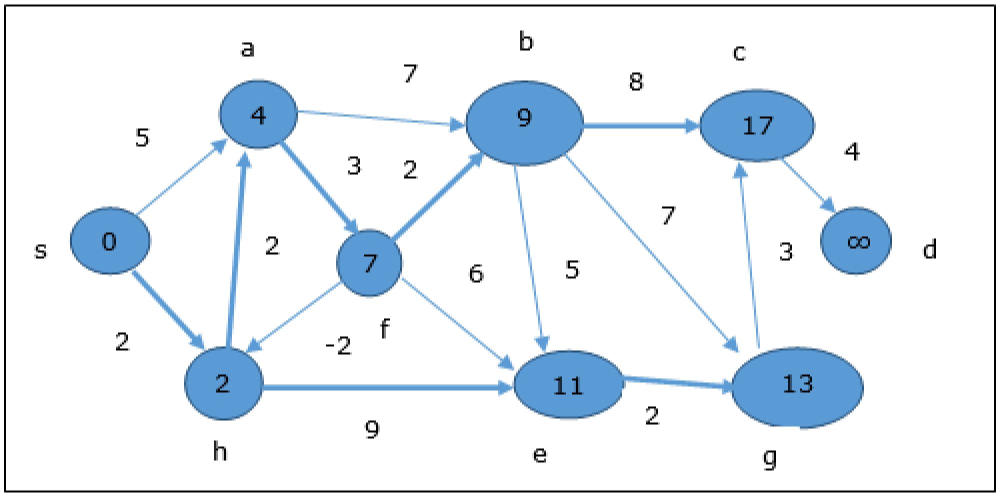
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



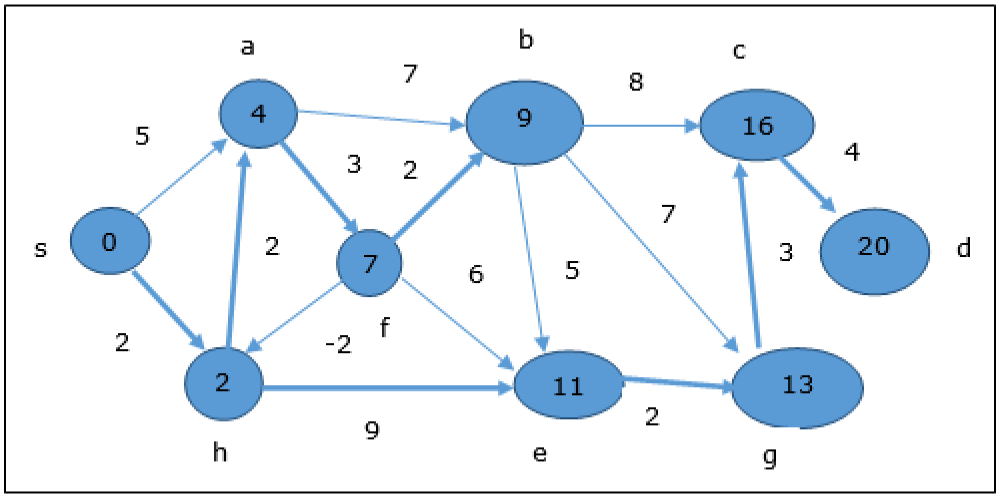
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



Hence, the minimum distance between vertex **s** and vertex **d** is **20**.

Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

Dijkstra’s Algorithm

Dijkstra’s algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., *w(u, v)* ≥ 0 for each edge *(u, v) Є E*).

In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

```
Algorithm: Dijkstra's-Algorithm (G, w, s)
for each vertex v Є G.V
   v.d := ∞
   v.∏ := NIL
```

```
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v Є G.adj[u]
      if v.d > u.d + w(u, v)
         v.d := u.d + w(u, v)
         v.∏ := u
```

Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is ***O(V^2 + E)***.

In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.

Example

Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

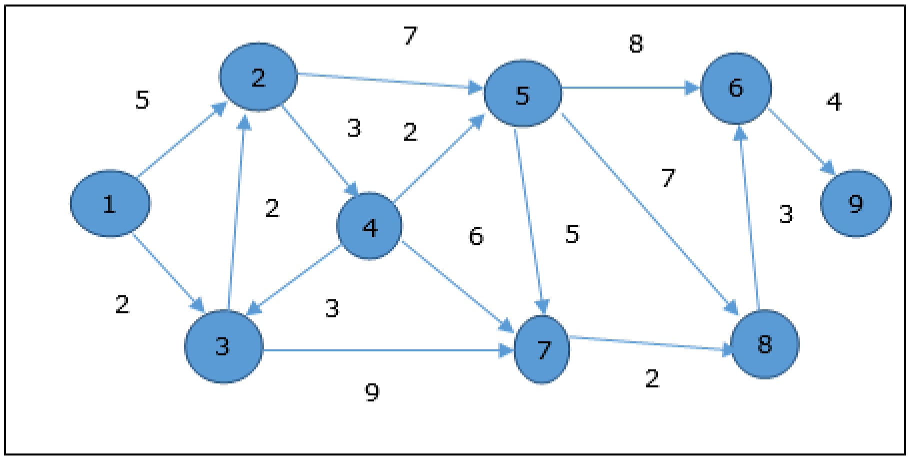
Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11

8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



Bellman Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph ***G = (V, E)*** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

```
Algorithm: Bellman-Ford-Algorithm (G, w, s)
for each vertex v Є G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
   for each edge (u, v) Є G.E
      if v.d > u.d + w(u, v)
         v.d := u.d +w(u, v)
         v.∏ := u
for each edge (u, v) Є G.E
   if v.d > u.d + w(u, v)
      return FALSE
return TRUE
```

Analysis

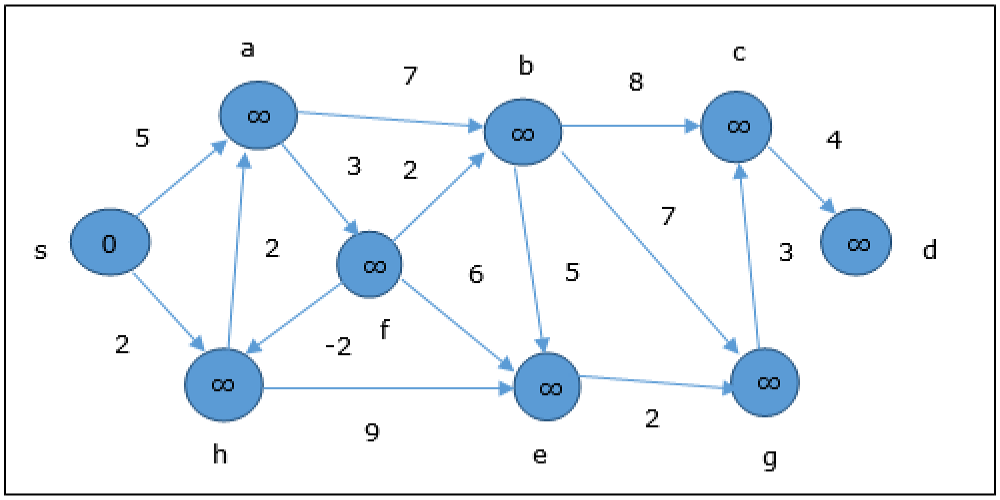
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs **|*V* - *1*|** passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

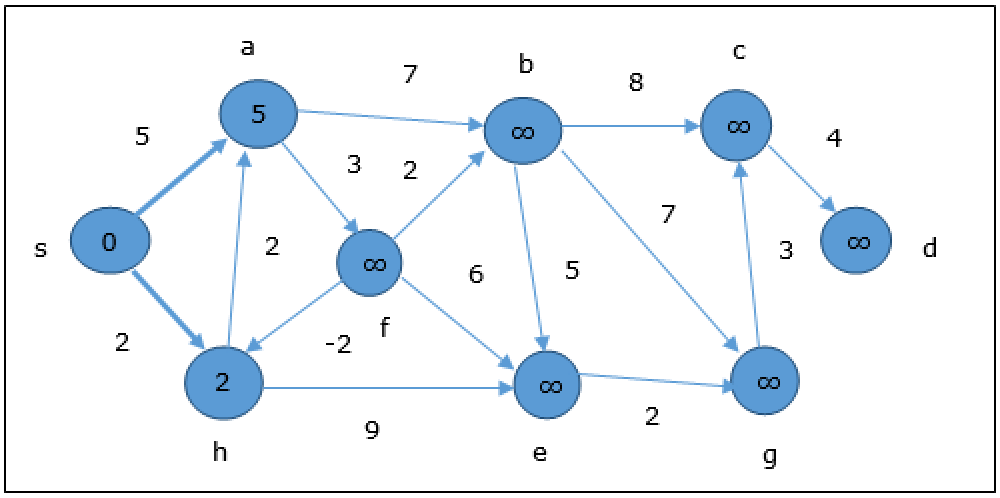
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

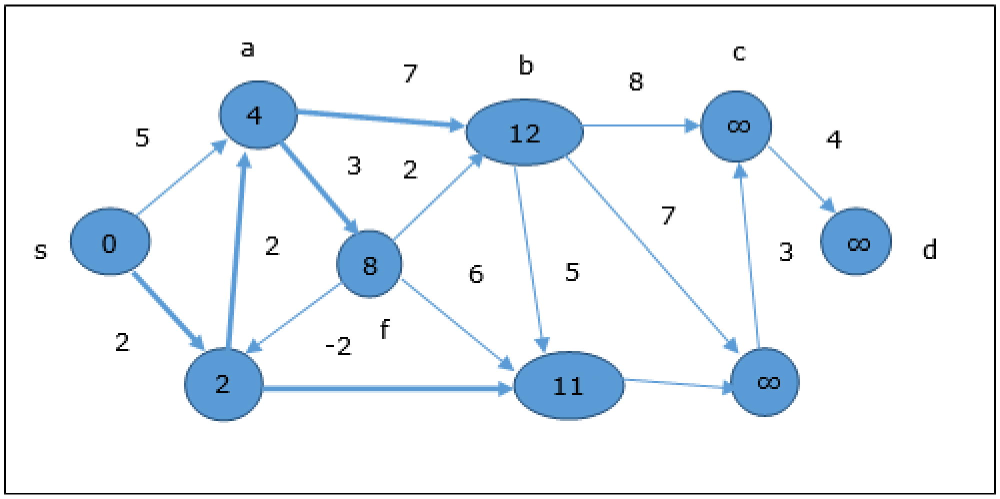
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



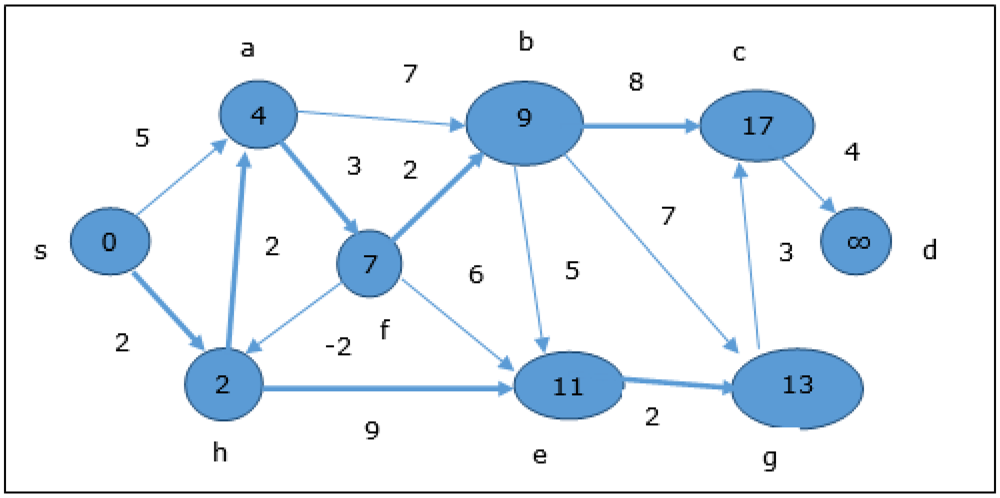
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



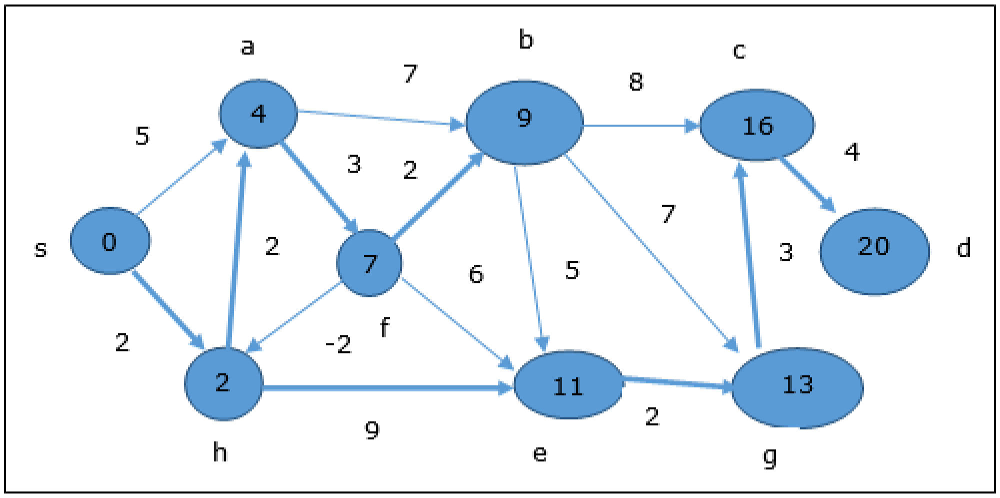
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



Hence, the minimum distance between vertex **s** and vertex **d** is **20**.

Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

TRAVELLING SALESMAN PROBLEM

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G* = (*V*, *E*)**, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S* Є {1, 2, 3, ... , *n*}** that includes ***1***, and ***j* Є *S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

C(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi ≠jC(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠j

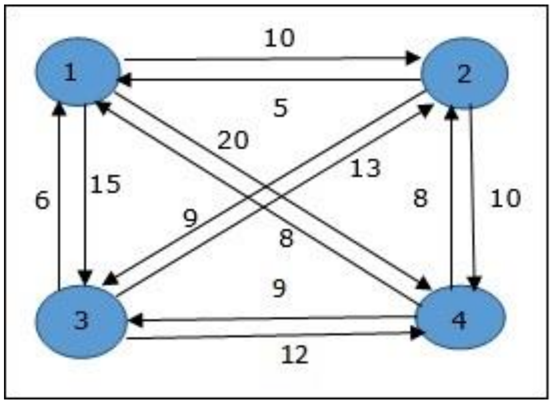
```
Algorithm: Traveling-Salesman-Problem
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
   for all j Є S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most $2^n.n$2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2^n.n^2)$O(2n.n2).

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8

S = 1

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18cost(2,{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18cost(2,{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3,{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3,{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3,{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3,{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13

S = 2

Cost(2,{3,4},1)
=d[2,3]+Cost(3,{4},1)=9+20=29
d[2,4]+Cost(4,{3},1)=10+15=25=25
{d[2,3]+cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25

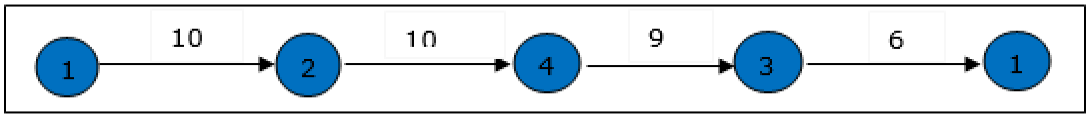
Cost(3,{2,4},1)=d[3,2]+Cost(2,{4},1)=13+18=31
d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1){d[3,2]+cost(2,{4},1)=13+18=31
\d[3,4]+Cost(4,{2},1)=12+13=25=25

Cost(4,{2,3},1)=⎧⎩ d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1){d[4,2]+cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



Branch and bound •

Technique for solving mixed (or pure) integer programming problems, based on tree search – Yes/no or 0/1 decision variables, designated xi – Problem may have continuous, usually linear, variables – O(2n) complexity

• Relies on upper and lower bounds to limit the number of combinations examined while looking for a solution

• Dominance at a distance – Solutions in one part of tree can dominate other parts of tree – DP only has local dominance: dominance: states in same stage dominate dominate

• Handles master/subproblem framework better than DP

• Same problem size as dynamic programming, perhaps a little larger: data specific, a few hundred 0/1 variables – Branch-and-cut is a more sophisticated, related method

• May solve problems with a few thousand 0/1 variables

• Its code and math are complex

• If you need branch-and-cut, use a commercial solver

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution

example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

bin packing problem:, items of different volumes must be packed into a finite number of bins or containers each of a fixed given volume in a way that minimizes the number of bins used. In computational complexity theory, it is a combinatorial NP-hard problem.[1] The decision problem (deciding if items will fit into a specified number of bins) is NP-complete.[2]

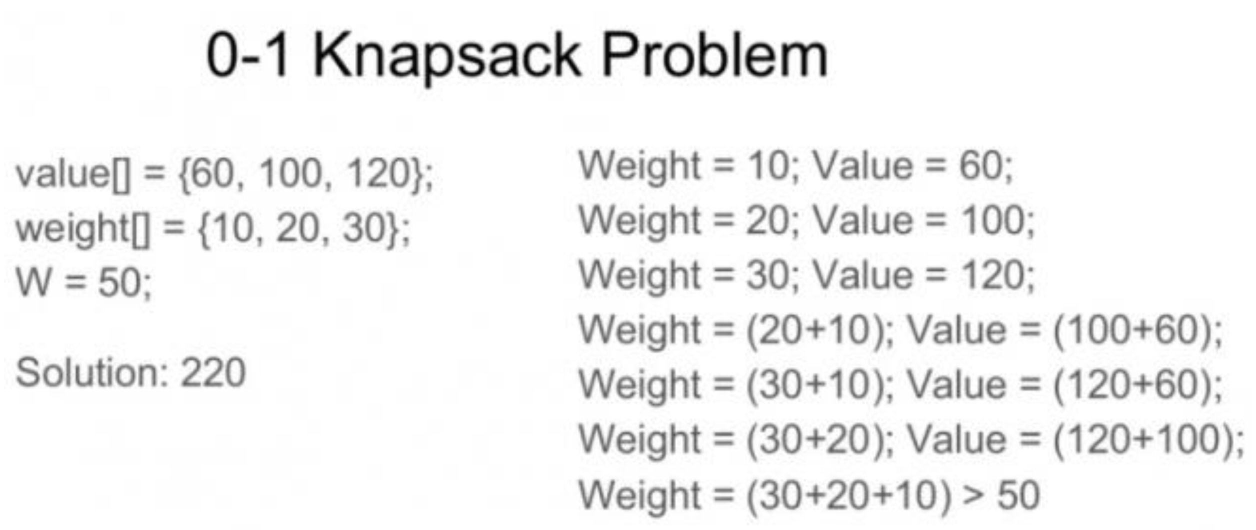
There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and so on. They have many applications, such as filling up containers, loading trucks with weight capacity constraints, creating file backups in media and technology mapping in field-programmable gate array semiconductor chip design.

The bin packing problem can also be seen as a special case of the cutting stock problem. When the number of bins is restricted to 1 and each item is characterised by both a volume and a value, the problem of maximising the value of items that can fit in the bin is known as the knapsack problem.

knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item 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. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint,

0-1 Knapsack Problem | DP-10

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).


0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms often times used to solve NP-complete problems, a class of decision problems. In these problems, there is no known efficient way to find a solution quickly and accurately although solutions can be verified when given. Heuristics can produce a solution individually or be used to provide a good baseline and are supplemented with optimization algorithms. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive.

Understanding Heuristics

Digital technology has disrupted all industries including finance, retail, media, and transportation. Suddenly, once typical daily activities have become outdated. Checks are deposited to bank accounts without visiting a local branch, products and services are purchased online and take-out food is delivered by food service delivery apps. Technology is creating data, which is increasingly shared across multiple industries and sectors, and a professional in any industry may find themselves working with mounds of complex data to solve a problem. Heuristic methods can help with data complexity given limited time and resources.

Why Use Heuristics?

Heuristics facilitate timely decisions. Analysts in every industry use rules of thumb such as intelligent guesswork, trial and error, process of elimination, past formulas and the analysis of historical data to solve a problem. Heuristic methods make decision making simpler and faster through short cuts and good-enough calculations.

The Disadvantages of Using Heuristics

There are trade-offs with the use of heuristics that render the approach prone to bias and errors in judgment. The user's final decision may not be the optimal or best solution, the decision made may be inaccurate and the data selected might be insufficient leading to an imprecise solution to a problem. For example, copycat investors often imitate the investment pattern of successful investment managers to avoid researching securities and the associated quantitative and qualitative information on their own.

By using a heuristic approach underlying past performance, copycat investors hope that the formulas used by these managers will continually earn them profits, but this is not always the case. For example, the crash of Valeant Pharmaceutical International was a shock to investors when the company saw its stock plunge 90% from 2015 to 2016. Valeant was a stock held in the portfolios of many hedge fund managers and the investors copying them.

Representativeness Heuristics

A popular shortcut method in problem-solving is Representativeness Heuristics. Representativeness uses mental shortcuts to make decisions based on past events or traits that are representative of or similar to the current situation. Say, for example, Fast Food ABC expanded its operations to India and its stock price soared. An analyst noted that India is a profitable venture for all fast food chains. Therefore, when Fast Food XYZ announced its plan to explore the Indian market the following year, the analyst wasted no time in giving XYZ a “buy” recommendation.

Although his shortcut approach saved reviewing data for both companies, it may not have been the best decision. XYZ may have food that is not appealing to Indian consumers, which research would have revealed. Other prevalent heuristic approaches for decision-making and problem-solving include Availability Bias, Anchoring and Adjustment, Familiarity Heuristic, Hindsight Bias and Naïve Diversification.

Module-3

Graph and tree are the non-linear data structure which is used to solve various complex problems. A **graph** is a group of vertices and edges where an edge connects a pair of vertices whereas a **tree** is considered as a minimally connected **graph** which must be connected and free from loops.

Difference between graph and tree

Graph :
A graph is collection of two sets V and E where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

- Vertices are nothing but the nodes in the graph.
- Two adjacent vertices are joined by edges.
- Any graph is denoted as G = {V, E}.

For Example:

G = {{V_1, V_2, V_3, V_4, V_5, V_6}, {E_1, E_2, E_3, E_4, E_5, E_6, E_7}}

Tree :

A tree is a finite set of one or more nodes such that –

1. There is a specially designated node called root.
2. The remaining nodes are partitioned into n>=0 disjoint sets T_1, T_2, T_3, …, T_n where T_1, T_2, T_3, …, T_n is called the subtrees of the root.

The concept of tree is represented by following Fig.

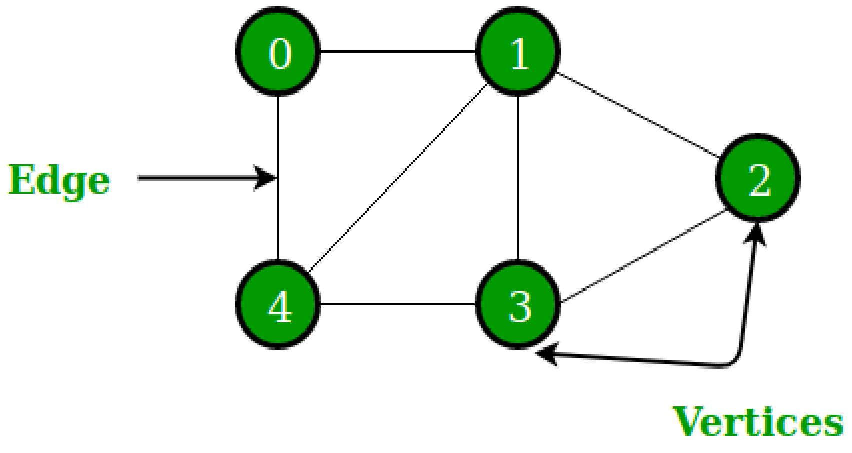
Difference between graph and tree

Graph :

A graph is collection of two sets V and E where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

- Vertices are nothing but the nodes in the graph.
- Two adjacent vertices are joined by edges.
- Any graph is denoted as G = {V, E}.

For Example:

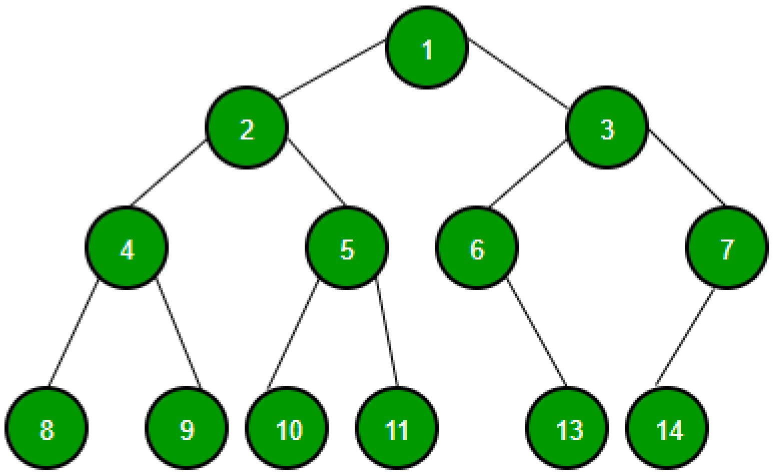


G = {{V_1, V_2, V_3, V_4, V_5, V_6}, {E_1, E_2, E_3, E_4, E_5, E_6, E_7}}

Tree :
A tree is a finite set of one or more nodes such that –

1. There is a specially designated node called root.
2. The remaining nodes are partitioned into n>=0 disjoint sets T_1, T_2, T_3, …, T_n where T_1, T_2, T_3, …, T_n is called the subtrees of the root.

The concept of tree is represented by following Fig.



Trees are graphs that do not contain even a single cycle. They represent hierarchical structure in a graphical form. Trees belong to the simplest class of graphs. Despite their simplicity, they have a rich structure.

Trees provide a range of useful applications as simple as a family tree to as complex as trees in data structures of computer science.

Tree

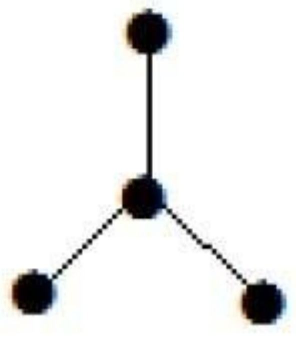
A **connected acyclic graph** is called a tree. In other words, a connected graph with no cycles is called a tree.

The edges of a tree are known as **branches**. Elements of trees are called their **nodes**. The nodes without child nodes are called **leaf nodes**.

A tree with ‘n’ vertices has ‘n-1’ edges. If it has one more edge extra than ‘n-1’, then the extra edge should obviously has to pair up with two vertices which leads to form a cycle. Then, it becomes a cyclic graph which is a violation for the tree graph.

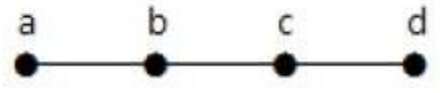
Example 1

The graph shown here is a tree because it has no cycles and it is connected. It has four vertices and three edges, i.e., for ‘n’ vertices ‘n-1’ edges as mentioned in the definition.



Note − Every tree has at least two vertices of degree one.

Example 2



In the above example, the vertices ‘a’ and ‘d’ has degree one. And the other two vertices ‘b’ and ‘c’ has degree two. This is possible because for not forming a cycle, there should be at least two single edges anywhere in the graph. It is nothing but two edges with a degree of one.

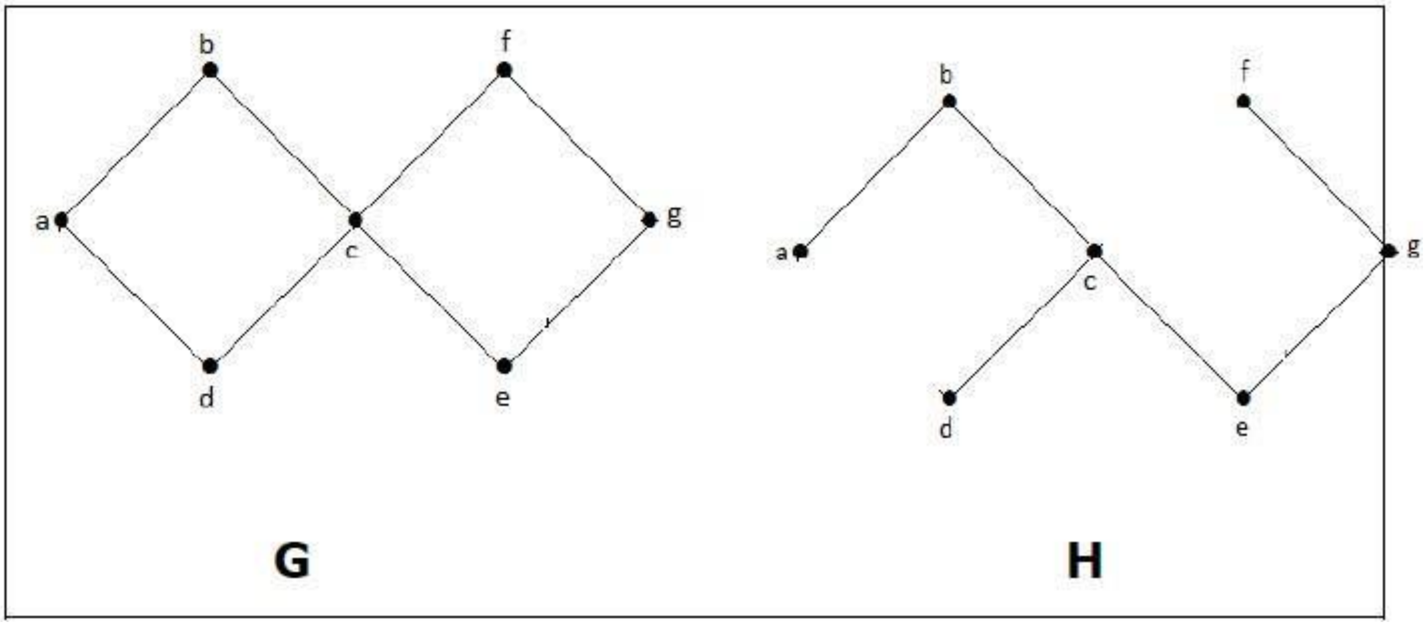
Spanning Trees

Let G be a connected graph, then the sub-graph H of G is called a spanning tree of G if –

- H is a tree
- H contains all vertices of G.

A spanning tree T of an undirected graph G is a subgraph that includes all of the vertices of G.

Example



In the above example, G is a connected graph and H is a sub-graph of G.

Clearly, the graph H has no cycles, it is a tree with six edges which is one less than the total number of vertices. Hence H is the Spanning tree of

Graph theory connectivity:-

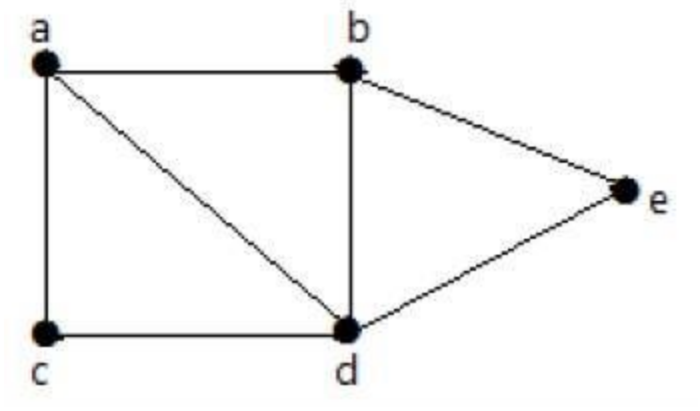
Whether it is possible to traverse a graph from one vertex to another is determined by how a graph is connected. Connectivity is a basic concept in Graph Theory. Connectivity defines whether a graph is connected or disconnected. It has subtopics based on edge and vertex, known as edge connectivity and vertex connectivity. Let us discuss them in detail.

Connectivity

A graph is said to be **connected if there is a path between every pair of vertex**. From every vertex to any other vertex, there should be some path to traverse. That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.

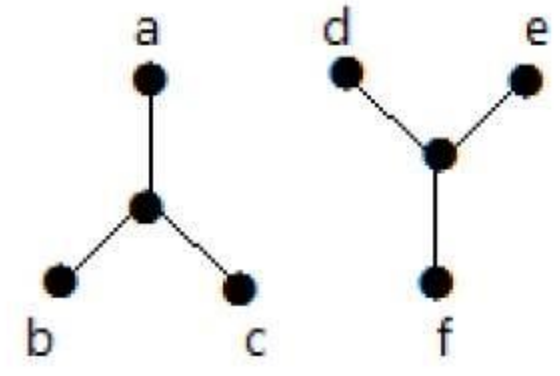
Example 1

In the following graph, it is possible to travel from one vertex to any other vertex. For example, one can traverse from vertex ‘a’ to vertex ‘e’ using the path ‘a-b-e’.



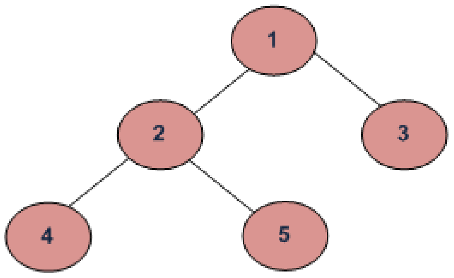
Example 2

In the following example, traversing from vertex ‘a’ to vertex ‘f’ is not possible because there is no path between them directly or indirectly. Hence it is a disconnected graph.



Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5
Please see this post for Breadth First Traversal.

Inorder Traversal (Practice):

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

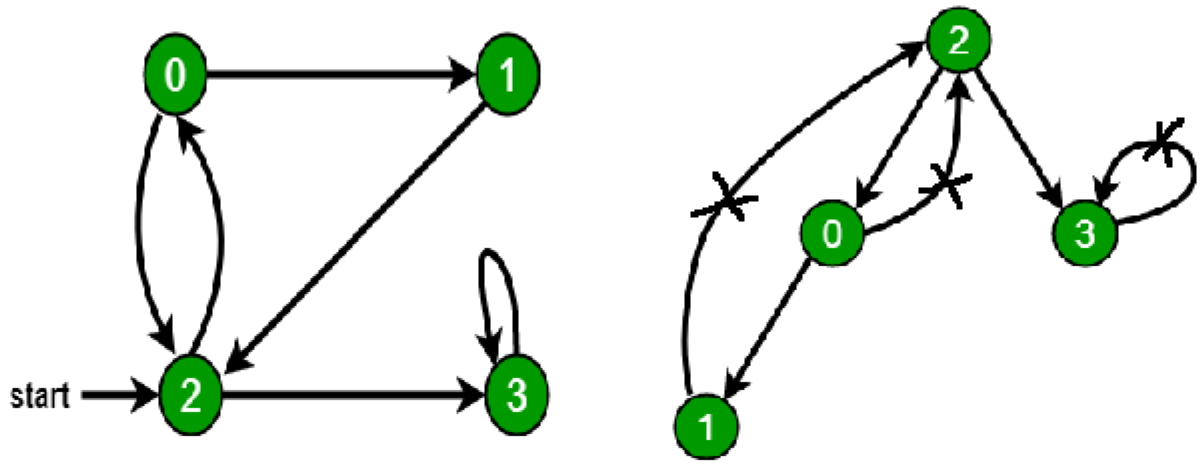
Preorder Traversal (Practice):

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Tree Traversal Algorithms can be classified broadly in the following two categories by the order in which the nodes are visited:

- **Depth-First Search (DFS) Algorithm:** It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.

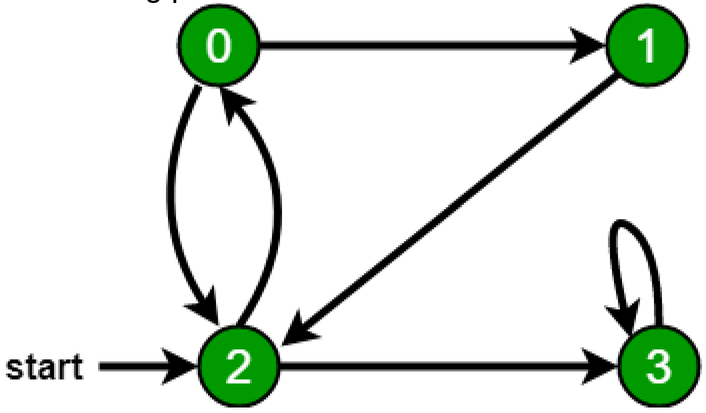
  - Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.
  - For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



- **Breadth-First Search (BFS) Algorithm:** It also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree. We will cover one algorithm of BFS type in the upcoming section.

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all

vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.

Bellman Ford's Algorithm:

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n−1 edges, because the shortest path couldn't have a cycle.

Algorithm Steps:

- The outer loop traverses from 0 : n−1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Time Complexity of Bellman Ford algorithm is relatively high O(V·E), in case E=V2, O(E3).
Let's discuss an optimized algorithm.

Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Time Complexity of Dijkstra's Algorithm is O(V2) but with min-priority queue it drops down to O(V+ElogV).

Floyd–Warshall's Algorithm

Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in O(V3), where V is the number of vertices in a graph.

The Algorithm Steps:

For a graph with N vertices:

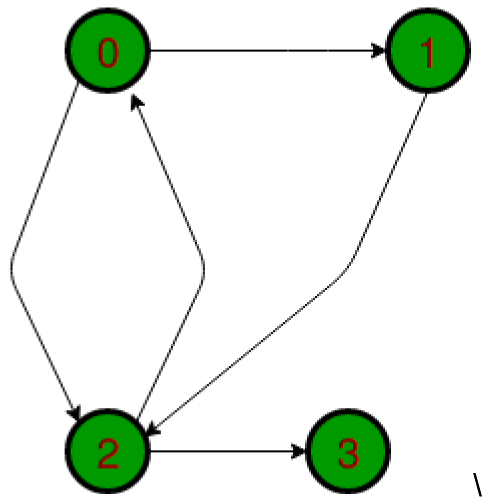
- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

Transitive closure of a graph

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called transitive closure of a graph.

```
For example, consider below graph
```



Minimum Spanning Tree

What is a Spanning Tree?

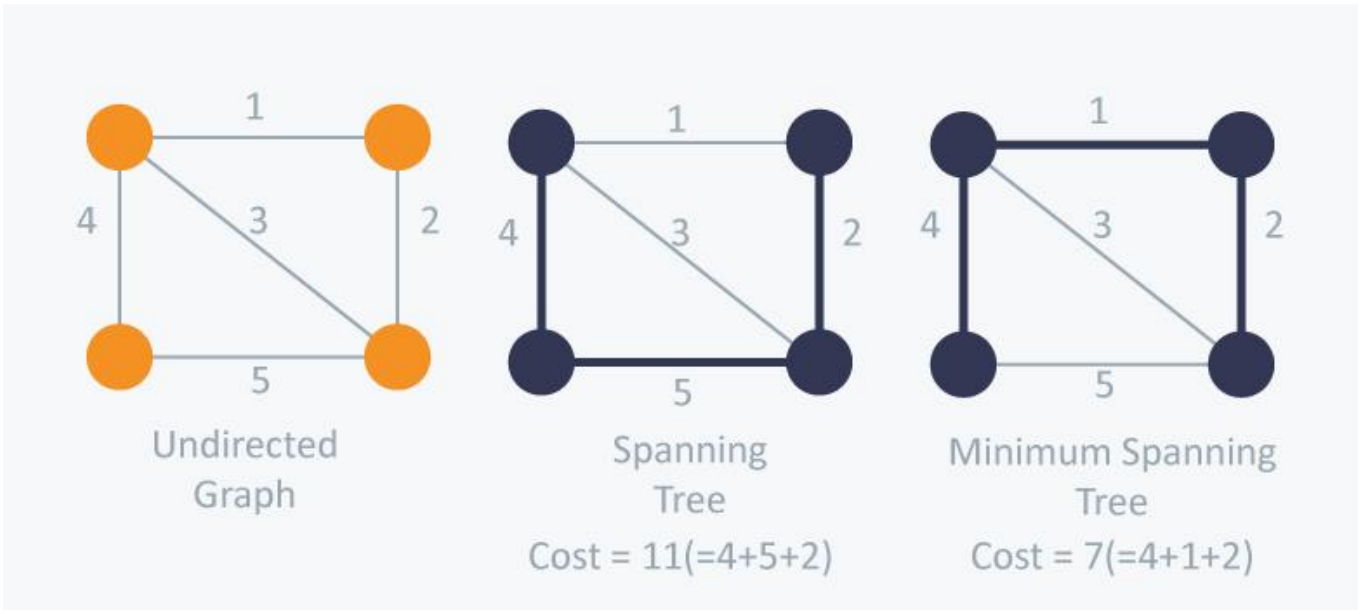
Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal’s Algorithm

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

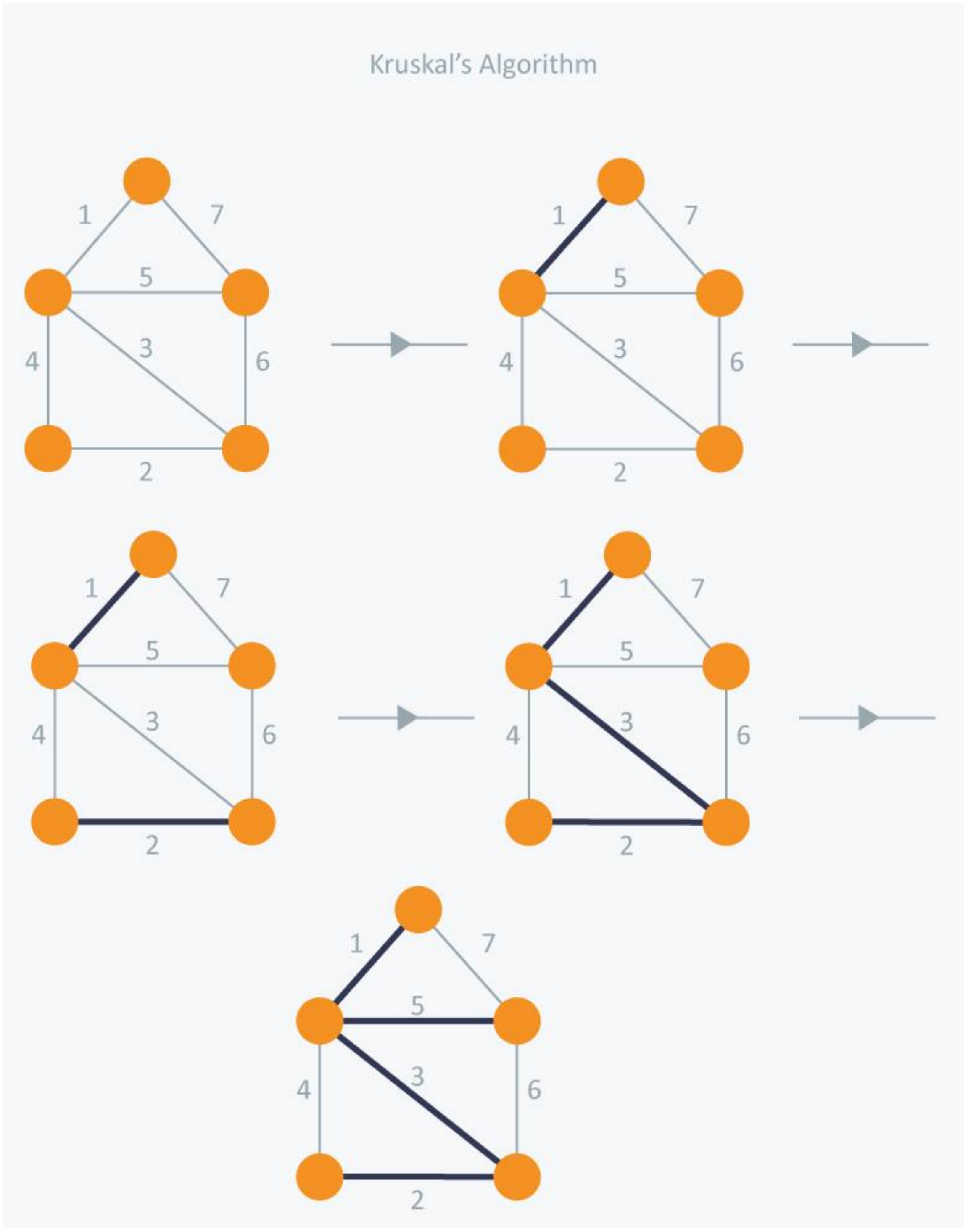
Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **"Disjoint Sets":**
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:



In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Implementation:

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
    for(int i = 0;i < MAX;++i)
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        // Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
    }
    return minimumCost;
}

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity:
In Kruskal’s algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

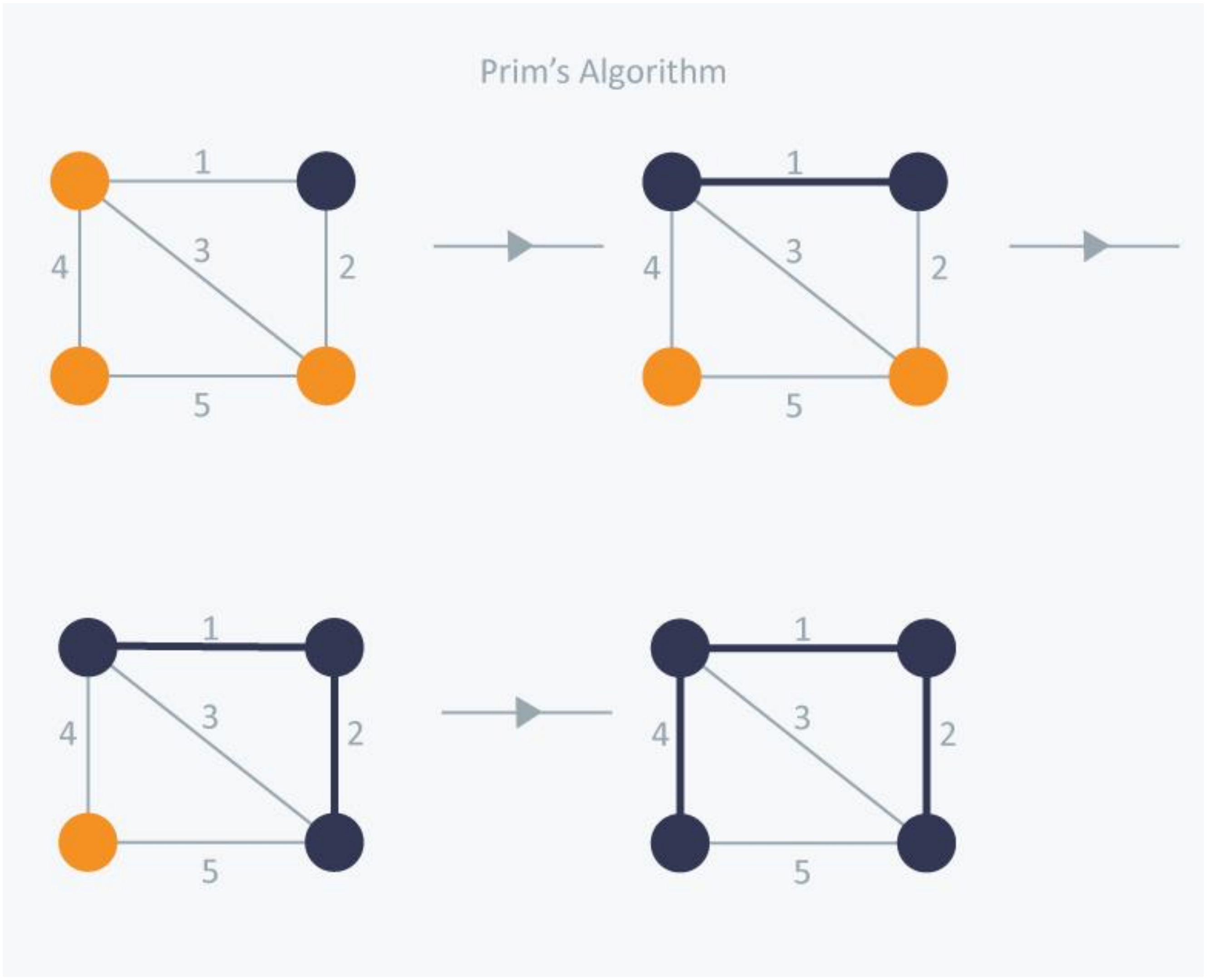
Prim’s Algorithm

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Implementation:

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>

using namespace std;
```

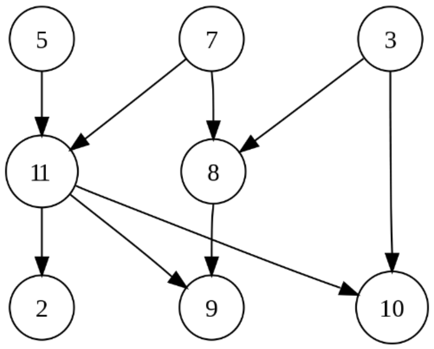
```
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    int y;
    long long minimumCost = 0;
    PII p;
    Q.push(make_pair(0, x));
    while(!Q.empty())
    {
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}

int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity:
The time complexity of the Prim’s Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

Topological sorting

topological sort or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Example



The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Module 4

Computable and non-computable problem

Computable Problems –
You are familiar with many problems (or functions) that are computable (or decidable), meaning there exists some algorithm that computes an answer (or output) to any instance of the problem (or for any input to the function) in a finite number of simple steps.A simple example is the integer increment operation:

```
f(x) = x + 1
```

It should be intuitive that given any integer x, we can compute x + 1 in a finite number of steps. Since x is finite, it may be represented by a finite string of digits. Using the addition method (or algorithm) we all learned in school, we can clearly compute another string of digits representing the integer equivalent to x + 1.

Yet there are also problems and functions that that are non-computable (or undecidable or uncomputable), meaning that there exists no algorithm that can compute an answer or output for all inputs in a finite number of simple steps. (Undecidable simply means non-computable in the context of a decision problem, whose answer (or output) is either “true” or “false”).

Non-Computable Problems –
A non-computable is a problem for which there is no algorithm that can be used to solve it. Most famous example of a non-computablity (or undecidability) is the **Halting Problem**. Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes).
The alternative is that it runs forever without halting. The halting problem is about seeing if a machine will ever come to a halt when a certain input is given to it or if it will finish running. This input itself can be something that keeps calling itself forever which means that it will cause the program to run forever.

Other example of an uncomputable problem is: determining whether a computer program loops forever on some input. You can replace “computer program” by “Turing machine or algorithm”if you know about Turing machine.

P, NP-Complete, NP, and NP-Hard

NP problems have their own significance in programming, but the discussion becomes quite hot when we deal with differences between NP, P , NP-Complete and NP-hard.

P and NP- Many of us know the difference between them.

P- Polynomial time **solving** . Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3). Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time.

NP- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP(travelling salesman problem) or An easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?.

but NP problems are **checkable** in polynomial time means that given a solution of a problem , we can check that whether the solution is correct or not in polynomial time.

So till now you have got what is NP and what is P.

Now we will discuss about NP-Complete and NP-hard.

but first we need to know what is **reducibility** .

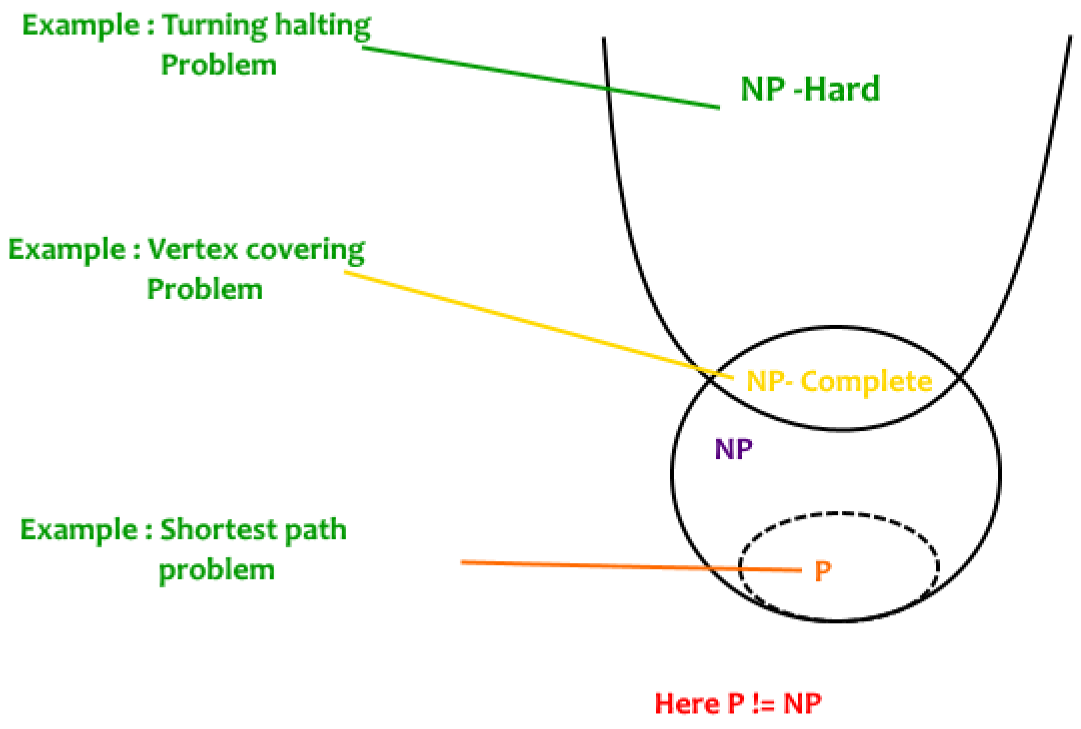
Take two problems A and B both are NP problems.

Reducibility- If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B.

NP-hard-- Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

NP-Complete -- The group of problems which are both in NP and NP-hard are known as NP-Complete problem.

Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. therefore Q will also be at least NP-hard , it may be NP-complete also.


Example : Turning halting Problem
NP -Hard
Example : Vertex covering Problem
NP- Complete
NP
Example : Shortest path problem
P
Here P != NP

Decision vs Optimization Problems

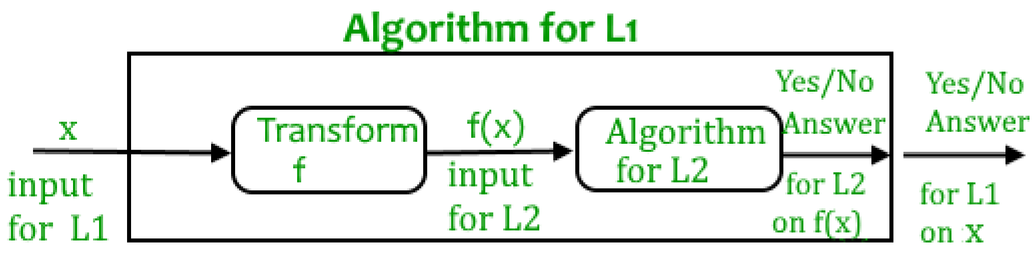
NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding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

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

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



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

How to prove that a given problem is NP complete?

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

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

Cook's Theorem

Stephen Cook presented four theorems in his paper "The Complexity of Theorem Proving Procedures". These theorems are stated below. We do understand that many unknown terms are being used in this chapter, but we don't have any scope to discuss everything in detail.

Following are the four theorems by Stephen Cook −

Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

- For any $T_Q(k)$ of type **Q**, TQ(k)k√(logk)2TQ(k)k(logk)2 is unbounded
- There is a $T_Q(k)$ of type **Q** such that TQ(k)⩽2k(logk)2TQ(k)⩽2k(logk)2

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type **Q**, then there is a constant **K**, so **S** can be recognized by a deterministic machine within time $T_Q(K8^n)$.

- First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.
- Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.
- Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.
- Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

1. **clique problem** is the computational **problem** of finding **cliques** (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph. It has several different formulations depending on which **cliques**, and what information about the **cliques**, should be found.

MODULE-5

next →← prev

Approximate Algorithms

Introduction:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Performance Ratios

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum. Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio P (n) for an input size n, where

Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.

Travelling Salesman Problem

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S Є {1, 2, 3, ... , n}*** that includes ***1***, and ***j Є S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$$C(S,j)=minC(S-\{j\},i)+d(i,j)wherei\in Sandi\neq jc(S,j)=minC(s-\{j\},i)+d(i,j)wherei\in Sandi\neq j$$

Analysis

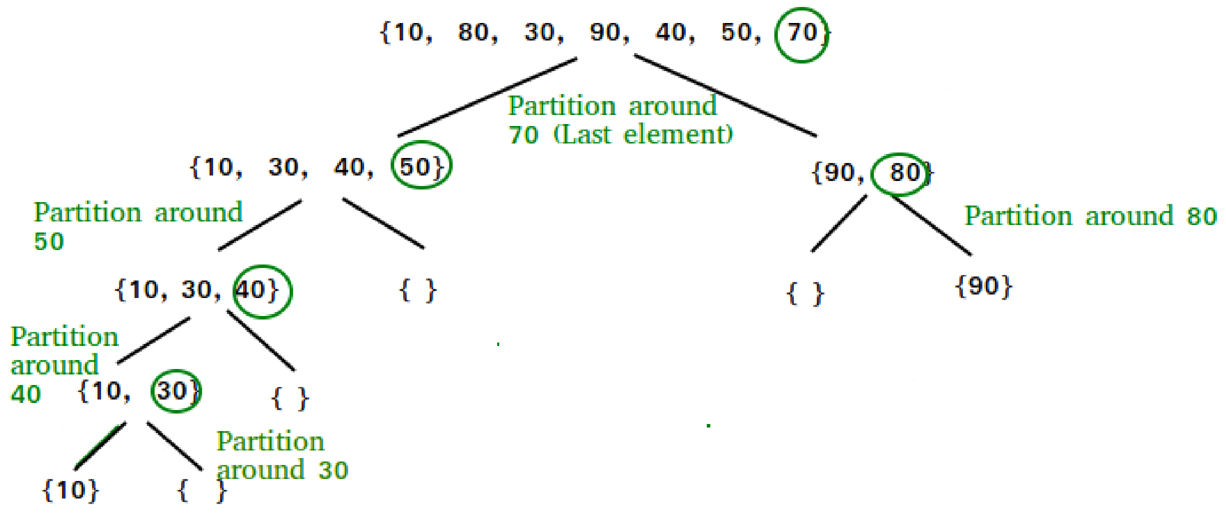
There are at the most $2^n.n$2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2^n.n^2)$O(2n.n2).

Randomized algorithms:

Quick sort: Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n^2), respectively.

•



-

N Queen Problem

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.

The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way.

A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

Input and Output

```
Input:
The size of a chess board. Generally, it is 8. as (8 x 8 is the size of a
normal chess board.)
Output:
The matrix that represents in which row and column the N Queens can be
placed.
If the solution does not exist, it will return false.

1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0
0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1
0  1  0  0  0  0  0  0
0  0  0  1  0  0  0  0
0  0  0  0  0  1  0  0
0  0  1  0  0  0  0  0

In this output, the value 1 indicates the correct place for the queens.
The 0 denotes the blank spaces on the chess board.
```

Minimum cut

In graph theory, a **minimum cut** or **min-cut** of a graph is a cut (a partition of the vertices of a graph into two disjoint subsets) that is minimal in some sense.

Variations of the minimum cut problem consider weighted graphs, directed graphs, terminals, and partitioning the vertices into more than two sets.

PSPACE-Complete

PSPACE. Decision problems solvable in polynomial space.

PSPACE-Complete. Problem Y is PSPACE-complete if (i) Y is in PSPACE and (ii) for every problem X in PSPACE, X ≤P Y. Theorem. [Stockmeyer-Meyer 1973] QSAT is PSPACE-complete.

Theorem. PSPACE ⊆ EXPTIME.

Pf. Previous algorithm solves QSAT in exponential time, and QSAT is PSPACE-complete.

▪ Summary. P ⊆ NP ⊆ PSPACE ⊆ EXPTIME. it is known that P ≠ EXPTIME, but unknown which inclusion is strict; conjectured that all areInput