Algorithmic Analysis of TCP/IP Network Model

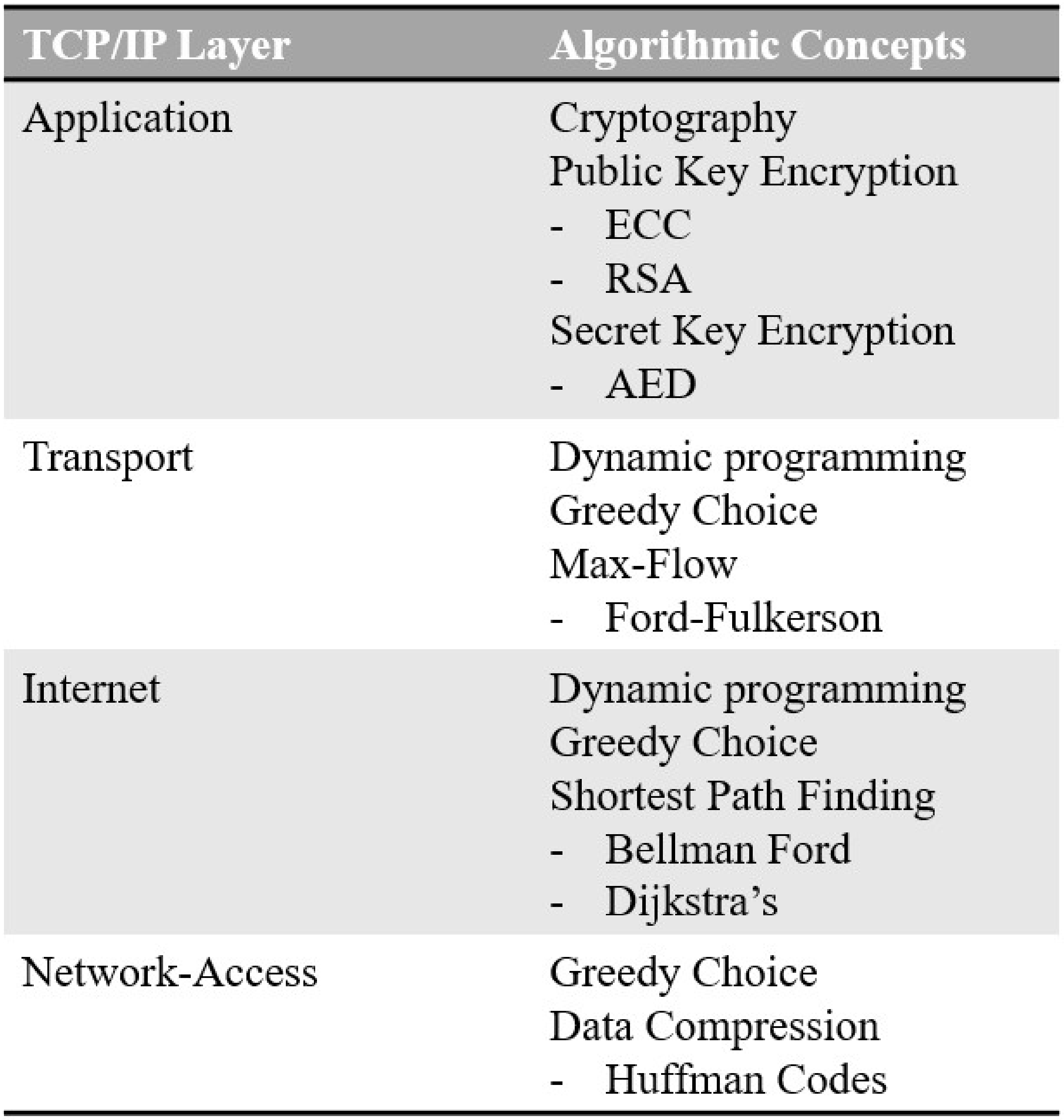
Technical Writing

for M21.CS1.301 – Algorithm Analysis and Design

by Pratham Gupta (2020101080)

Abstract

The TCP/IP model is the default method of data communication on the Internet. It breaks messages into packets to avoid having to resend the entire message in case it encounters a problem during transmission. TCP/IP divides communication tasks into layers that keep the process standardized through a set of discretely defined protocols, without hardware and software providers doing the management themselves. The data packets must pass through four layers before they are received by the destination device, then TCP/IP goes through the layers in reverse order to reassemble the packets, i.e., put the message back into its original format.
The protocols followed in the network model layers are, in fact, defined abstractions of particular computational algorithms that are employed to achieve/secure their respective functionalities. This is an extensive and intensive analysis of these algorithms, to develop a critical understanding of the TCP/IP model from the viewpoint of computational and algorithmic design.



TCP/IP Layer	Algorithmic Concepts
Application	Cryptography Public Key Encryption - ECC - RSA Secret Key Encryption - AED
Transport	Dynamic programming Greedy Choice Max-Flow - Ford-Fulkerson
Internet	Dynamic programming Greedy Choice Shortest Path Finding - Bellman Ford - Dijkstra's
Network-Access	Greedy Choice Data Compression - Huffman Codes

Index

Network-Access Layer

The Network Access layer of the TCP/IP model corresponds with the Data Link and Physical layers of the OSI reference model. It defines the protocols and hardware required to connect a host to a physical network and to deliver data across it. Packets from the Internet layer are sent down the Network Access layer for delivery within the physical network. The destination can be another host in the network, itself, or a router for further forwarding. So the Internet layer has a view of the entire Internetwork whereas the Network Access layer is limited to the physical layer boundary that is often defined by a layer 3 device such as a router.

The Network Access layer consists of a large number of protocols. When the physical network is a LAN, Ethernet at its many variations are the most common protocols used. On the other hand when the physical network is a WAN, protocols such as the Point-to-Point Protocol (PPP) and Frame Relay are common.

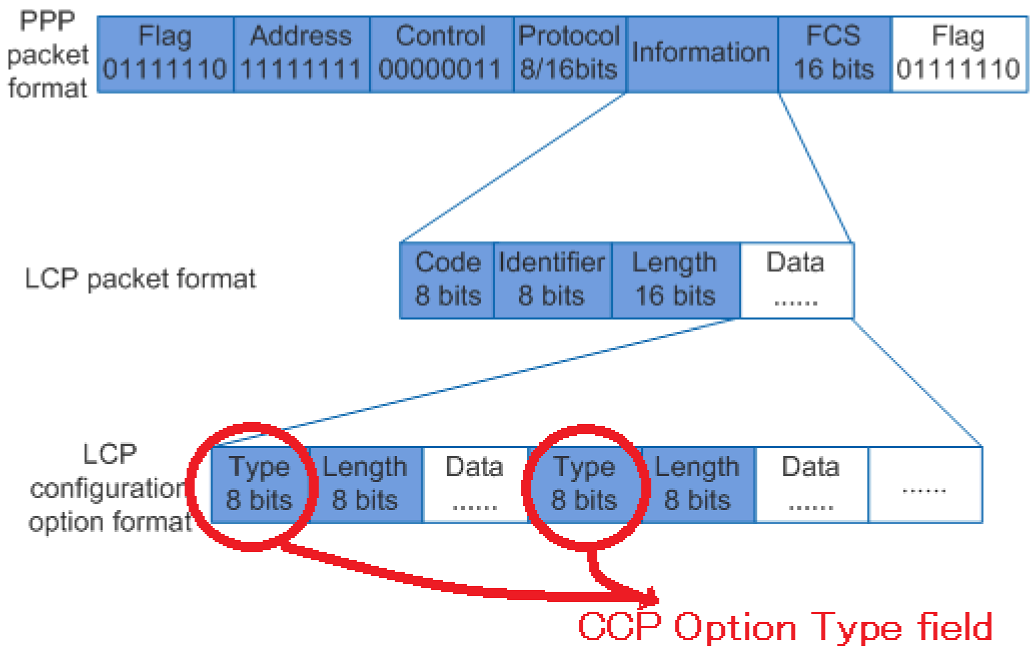
`PPP` is, of course, primarily used to provide data link layer connectivity to physical serial links. One of the biggest problems with serial links compared to many other types of layer one connections is that they are relatively slow. Consider that while 10 Mbps regular Ethernet is considered sluggish by modern LAN standards, it is actually much faster than most serial lines used for WAN connectivity, which can be 10, 100 or even 1000 times slower.

One way to improve performance over serial links is to use compression on the data sent over the line. Depending on the data transferred, this can double the performance compared to uncompressed transmissions, and can in some cases do even better than that. For this reason, many hardware devices include the ability to compress the data stream at the physical layer. The best example of this is probably the set of compression protocols used on analog modems.

`PPP Compression Control Protocol (CCP)` is responsible for negotiating and managing the use of compression on a PPP link.

The Compression Control Protocol (CCP) is responsible for configuring, enabling, and disabling data compression algorithms on both ends of the point-to-point link. It is also used to signal a failure of the compression/decompression mechanism in a reliable manner. A wide variety of compression methods may be negotiated, although typically only one method is used in each direction of the link.

A different compression algorithm may be negotiated in each direction, for speed, cost, memory or other considerations, or only one direction may be compressed.

> Several of the **PPP Compression Algorithm** are defined in `Internet standards (RFCs)`. In addition, it is possible for two devices to negotiate the use of a proprietary compression method if they want to use one not defined by a public standard.
>
> CCP Configuration Options allow negotiation of compression algorithms and their parameters. CCP uses the same Configuration Option format defined for LCP , with a separate set of Options.
>
> Configuration Options, in this protocol, indicate algorithms that the receiver is willing or able to use to decompress data sent by the sender. As a result, it is to be expected that systems will offer to accept several algorithms, and negotiate a single one that will be used.
>
> 
>
>
> > These `CCP Option Type bytes` specify the algorithms that the node is willing to operate on. They are stated directly in the most recent "Assigned Numbers" RFC.
> >
> > Some unassigned values are intended to be assigned to other freely available compression algorithms that have no license fees.
> >
> > The `OUI` value configuration option provides a way to negotiate the use of a proprietary compression protocol.

Some of the public standard Data Compression algorithms are:

1. `Huffman Coding`
2. Lempel-Ziv range of algorithms viz `LV77`, `LZMA`
3. Huffman Coding - Lempel Ziv hybrids viz `Deflate`

Huffman Coding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. The variable-length codes assigned to input characters are `Prefix Codes`.

> **Prefix code** is a type of code system (typically a variable-length code) distinguished by its possession of the `“prefix property”`,i.e. the codes(bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream. Consider the counter example, say there are four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”

Motivation:

- Transmitting the characters with the higher frequencies with smaller bit sequences shall be an optimal compression strategy.

Given:

- Input array of unique characters along with their frequency of occurrences

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

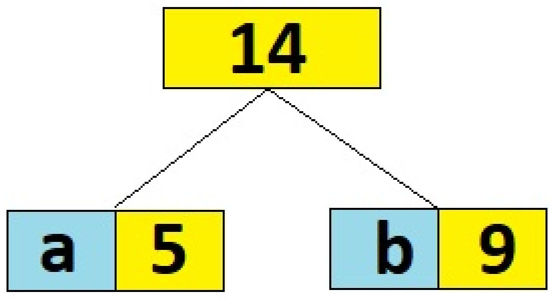
Algorithm:

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.

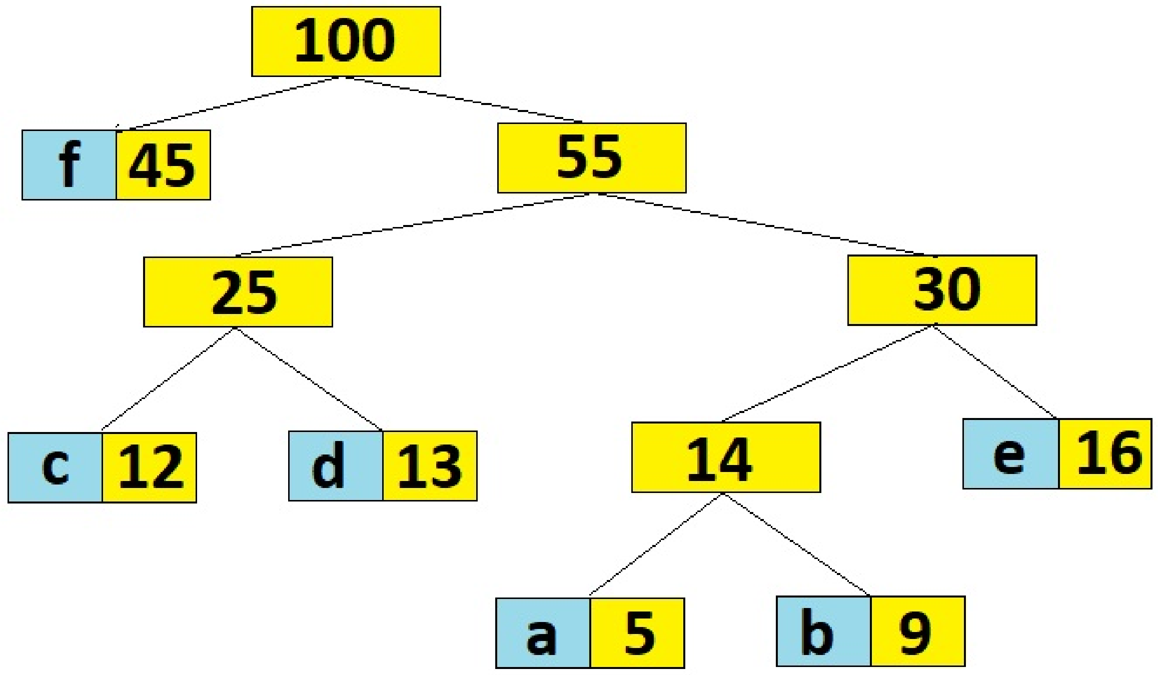
   > This is a **Greedy Choice** since making the two symbols with least frequency have the maximum code length implicitly ensures reduced cost of transmission.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

> This provides an **Optimal Substructure** as the next greedy choice can be made normally from the `n-2+1 = n-1` nodes.



1. Repeat steps #2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.



1. Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

```
character   code-word   bits
 f          0          1
 c          100        3
 d          101        3
 e          111        3
 a          1100       4
 b          1101       4
```

In [3]:

```python
from heapq import *
from collections import defaultdict
def encode(frequency):
    heap = [[weight, [symbol, '']] for symbol, weight in frequency.items()]
    heapify(heap)
    while len(heap) > 1:
        low = heappop(heap)

        high = heappop(heap)

        for value in low[1:]:

            value[1] = '0' + value[1]

        for value in high[1:]:

            value[1] = '1' + value[1]

        heappush(heap, [low[0] + high[0]] + low[1:] + high[1:])

    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

'''
string=input("Enter the string to be encoded: ")

frequency = defaultdict(int)

for character in string:

    frequency[character] += 1
'''
frequency ={
    'a':5,
    'b':9,
    'c':12,
    'd':13,
    'e':16,
    'f':45
}

huff = encode(frequency)
print("\n" + "Character".ljust(10),"Frequency".ljust(10), "Huffman-Code")

for i in huff:
    print(i[0].ljust(10), str(frequency[i[0]]).ljust(10), i[1])
```

```
Character  Frequency  Huffman-Code
f          45         0
c          12         100
d          13         101
e          16         111
a          5          1100
b          9          1101
```

Time complexity: O(nlogn) where n is the number of unique characters.

- If there are n nodes, `heappop()` is called 2*(n – 1) times, which in turn calls for `heapification` of the heap. General implementations of the heapify function takes `O(logn)`. So, overall complexity is O(nlogn).

> In information theory, **Shannon's source coding theorem** (or noiseless coding theorem) is able to establish the limits to possible data compression, and the operational meaning of the Shannon entropy. The source coding theorem displays that (in the limit, as the length of a stream of independent and identically-distributed random variable (i.i.d.) data tends to infinity) it is not possible to compress the data such that the code rate (average number of bits per symbol) is smaller than the Shannon entropy of the source, without it being virtually certain that information will be lost. However, it is possible to obtain the code rate arbitrarily close to the Shannon entropy, with negligible probability of loss.
>
>> Entropy can be computed for a random variable X with k in K discrete states as $H(X) = \sum_{k \in K}(p(k) * log(1/p(k)))$
>
> The base of the log function in the context of Huffman Coding is `2` since the encoding is binary. Information entropy is defined as the average rate at which information is produced by a stochastic source of data. Arithmetic coding and Huffman coding produce equivalent results — achieving entropy — when every symbol has a probability of the form 1/2k. In other circumstances, arithmetic coding can offer better compression than Huffman coding because — intuitively — its "code words" can have effectively non-integer bit lengths, whereas code words in prefix codes such as Huffman codes can only have an integer number of bits. Therefore, a code word of length k only optimally matches a symbol of probability 1/2k and other probabilities are not represented optimally; whereas the code word length in arithmetic coding can be made to exactly match the true probability of the symbol. This difference is especially striking for small alphabet sizes.
>
>> The lowest entropy is computed for a random variable that has a single event with a probability of 1.0, a certainty. The largest entropy for a random variable will be possible if all events are performed equally likely. Hence **data with larger gradient in frequency of symbols has lower entropy, i.e. higher level of optimization by data compression**.

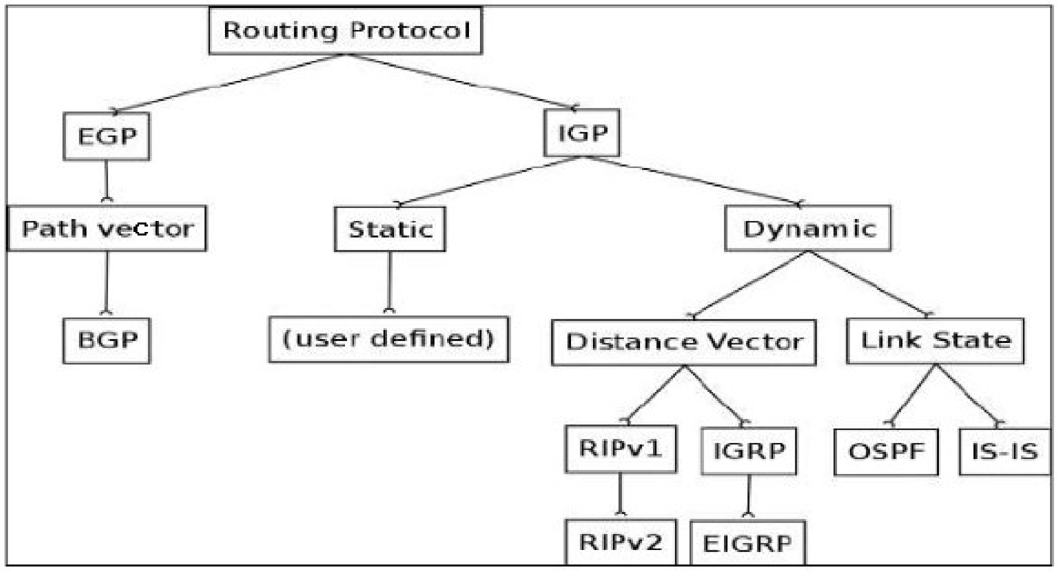
Internet Layer

The network/internet layer is responsible for packet forwarding including routing through intermediate routers. It defines the protocols which are responsible for logical transmission of data over the entire network. Some of these protocols extensively employ `Shortest Path algorithms` that aim to find the optimal paths between the network nodes so that routing cost is minimized.

> Devices responsible for routing messages between networks are called **gateways** or **routers** in TCP/IP terminology. The main protocol at this layer is the **Internet protocol (IP)**. The IP frame format includes data fields for checksums and network

> addressing, also known as IP addresses, similar to those defined in the OSI network layer. IP addresses are crucial to the protocol. They are used throughout the Internet to identify every single node in the world connected to it. The IP protocol also includes data fields for fragmentation and block re-ordering.

`Routing` is the process of connecting two or more different networks. A device that connects two different networks is called `router`. Each internet network requires protocol to process data communication, called it's `routing protocol`. A routing protocol serves to regulate data traffic in a network and can search for the best path for each of the data points in the network. Each routing protocol has an `entry route` i.e. the database storage route. Entry route lies in the routing table that lists ip addresses of all of the nodes in the network that it can traverse.

> > **Interior Gateway Protocol (IGP)** is a Routing Protocol which is used to find network path information within an Autonomous System. Known Interior Gateway Protocol (IGP) Routing Protocols are **Routing Information Protocol (RIP)**, **Interior Gateway Routing Protocol (IGRP)**, **Open Shortest Path First (OSPF)** and **Intermediate System to Intermediate System (IS-IS).**
> >
> > **Exterior Gateway Protocol (EGP)** is a Routing Protocol which is used to find network path information between different Autonomous Systems. Exterior Gateway Protocol (EGP) is commonly used in the Internet to exchange routing table information. There is only one Exterior Gateway Protocol (EGP) exists now and it is **Border Gateway Protocol (BGP)**.
>
> 
>

A Routing Protocol may be either `Static` or `Dynamic`

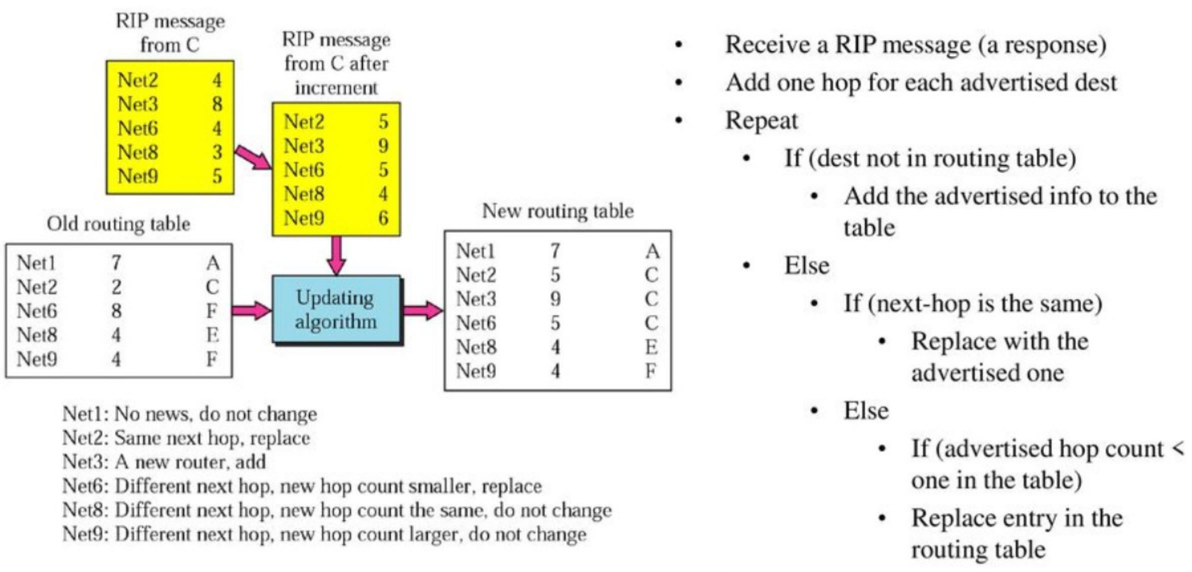
- **Static routing** is the process of connecting two or more different networks with entry routes manually input

- **Dynamic routing** connects two or more different networks where entry routes are automatically determined based off a computational graph algorithm, in the routing table.

`RIP (Routing Information Protocol)` and `OSPF (Open Shortest Path First) Protocol` are the most popular dynamic routing algorithms.

Routing Information Protocol (RIP) is a dynamic routing protocol that uses hop count as a routing metric to find the best path between the source and the destination network. It is the oldest `distance-vector routing protocol` that employs `Bellman-Ford Algorithm` to determine the shortest path from the entry node to the destination node.

> **Distance vector** is much simpler than Dijkstra because it only needs local information from the direct neighbors, whereas Dijkstra requires knowledge of the topology of the entire network.
>
> **Hop count** is the number of routers occurring in between the source and destination network. The path with the lowest hop count is considered as the best route to reach a network and therefore placed in the routing table. RIP prevents routing loops by limiting the number of hops allowed in a path from source and destination. The maximum hop count allowed for RIP is 15 and a hop count of 16 is considered as network unreachable.



RIP message from C

Net2	4
Net3	8
Net6	4
Net8	3
Net9	5

RIP message from C after increment

Net2	5
Net3	9
Net6	5
Net8	4
Net9	6

Old routing table

Net1	7	A
Net2	2	C
Net6	8	F
Net8	4	E
Net9	4	F

New routing table

Net1	7	A
Net2	5	C
Net3	9	C
Net6	5	C
Net8	4	E
Net9	4	F

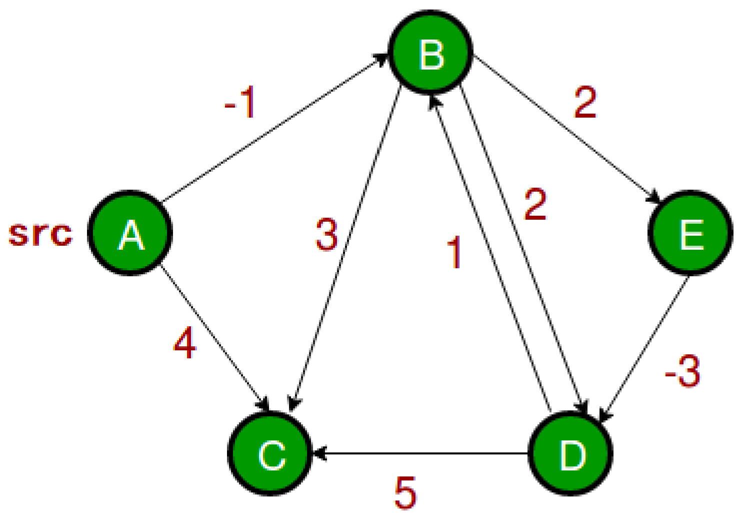
Net1: No news, do not change
Net2: Same next hop, replace
Net3: A new router, add
Net6: Different next hop, new hop count smaller, replace
Net8: Different next hop, new hop count the same, do not change
Net9: Different next hop, new hop count larger, do not change

- Receive a RIP message (a response)
- Add one hop for each advertised dest
- Repeat
  - If (dest not in routing table)
    - Add the advertised info to the table
  - Else
    - If (next-hop is the same)
      - Replace with the advertised one
    - Else
      - If (advertised hop count < one in the table)
      - Replace entry in the routing table

Bellman–Ford Algorithm

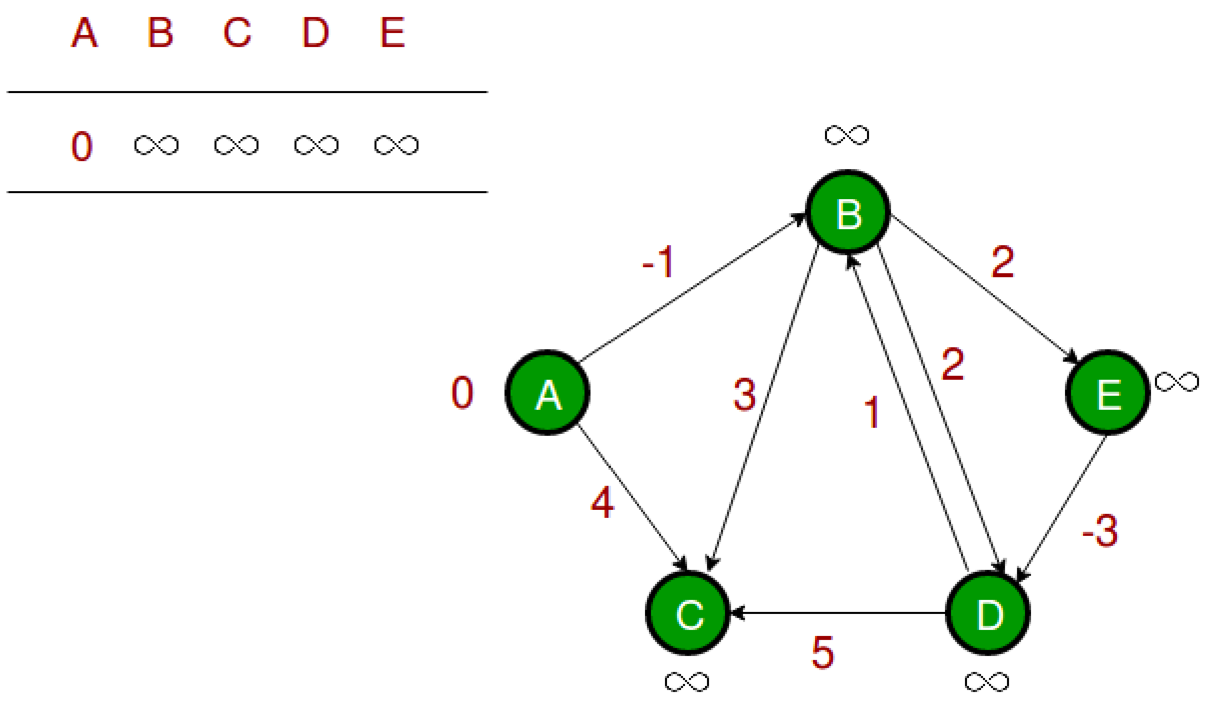
Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

Given: A graph (that may contain edges with negative weights) and a source vertex src in graph. Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.



Algorithm:

1. This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array `dist[]` of size `|V|` with all values as `infinite` except `dist[src]` where src is source vertex.



A	B	C	D	E
0	∞	∞	∞	∞

1. This step calculates shortest distances, essentially `dynamically` using already known values. Do following |V|-1 times where |V| is the number of vertices in given graph. Do following for each edge `u-v`

```
If dist[v] > dist[u] + weight of edge u-v, then update dist[v] as
    dist[v] = dist[u] + weight of edge u-v
```

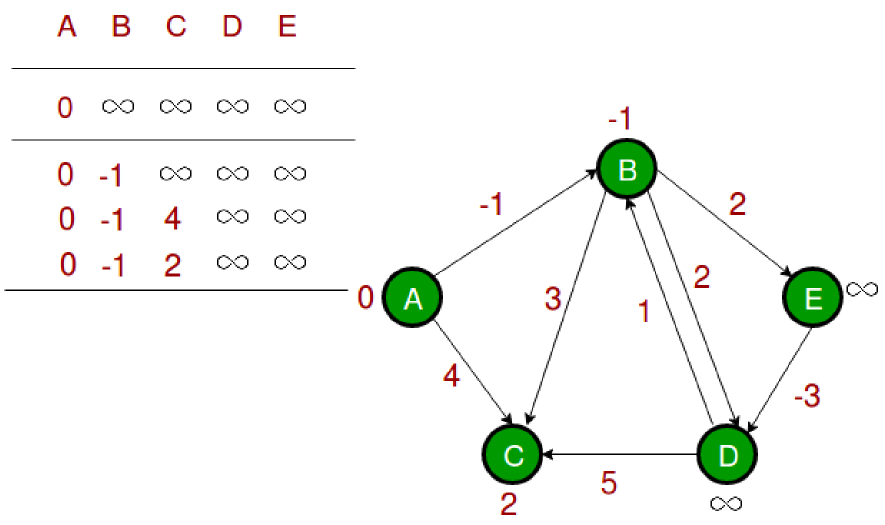
2. This step reports if there is a negative weight cycle in graph. Do following for each edge `u-v`

```
 If dist[v] > dist[u] + weight of edge u-v, then “Graph contains negative
weight cycle”
```

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

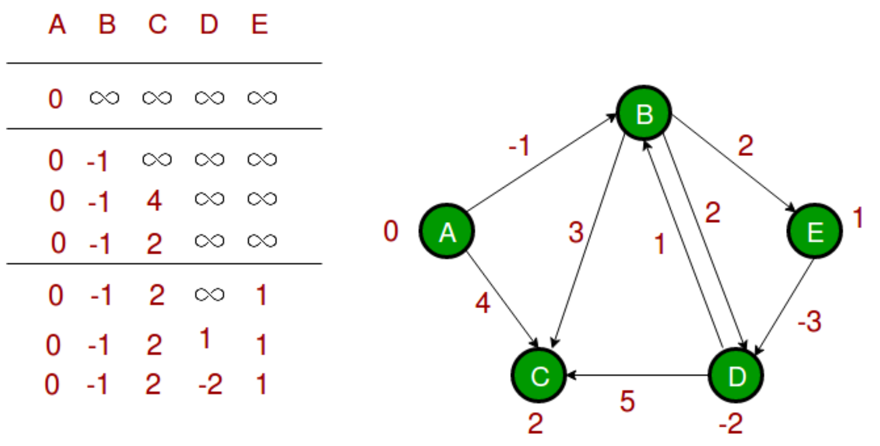
>

Pass 1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

Pass 2



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

Understanding:

Like other `Dynamic Programming` Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

In [5]:

```
# Python3 program for Bellman-Ford's
# single source shortest path algorithm.
from sys import maxsize

# The main function that finds shortest
# distances from src to all other vertices
# using Bellman-Ford algorithm. The function
# also detects negative weight cycle
# The row graph[i] represents i-th edge with
# three values u, v and w.
def BellmanFord(graph, V, E, src):

    # Initialize distance of all vertices as infinite.
    dis = [maxsize] * V

    # initialize distance of source as 0
    dis[src] = 0

    # Relax all edges |V| - 1 times. A simple
    # shortest path from src to any other
    # vertex can have at-most |V| - 1 edges
    for i in range(V - 1):
        for j in range(E):
            if dis[graph[j][0]] + graph[j][2] < dis[graph[j][1]]:
                dis[graph[j][1]] = dis[graph[j][0]] + graph[j][2]

    # check for negative-weight cycles.
    # The above step guarantees shortest
    # distances if graph doesn't contain
    # negative weight cycle. If we get a
    # shorter path, then there is a cycle.
    for i in range(E):
        x = graph[i][0]
        y = graph[i][1]
        weight = graph[i][2]
        if dis[x] != maxsize and dis[x] + weight < dis[y]:
            print("Graph contains negative weight cycle")

    print("Vertex".ljust(10), "Distance from Source".ljust(1))
    for i in range(V):
        print(str(i).ljust(10), str(dis[i]).ljust(10))

V = 5 # Number of vertices in graph
E = 8 # Number of edges in graph

# Every edge has three values (u, v, w) where
# the edge is from vertex u to v. And weight
# of the edge is w.
graph = [[0, 1, -1],
         [0, 2, 4],
         [1, 2, 3],
         [1, 3, 2],
         [1, 4, 2],
          [3, 2, 5],
          [3, 1, 2],
          [4, 3, -3]]
BellmanFord(graph, V, E, 0)
```

```
Vertex     Distance from Source
0          0
1          -1
2          2
3          -2
4          1
```

Time complexity: O(EV) where E is the number of edges and V is the number of vertices.

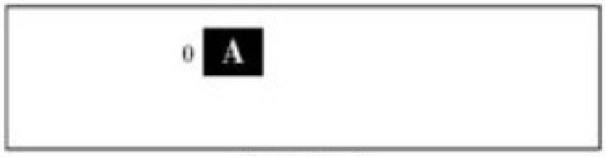
- This is because of the first loop that runs `|V|-1` times and the nested loop that runs `E` times.

`Open Shortest Path First` is a link-state routing protocol. You can think of a link as an interface on a router. The state of the link is a description of that interface and of its relationship to its neighboring routers. A description of the interface would include, for example, the IP address of the interface, the subnet mask, the type of network to which it is connected, the routers that are connected to that network, and so on. The collection of all of these link states forms a `link-state database`.

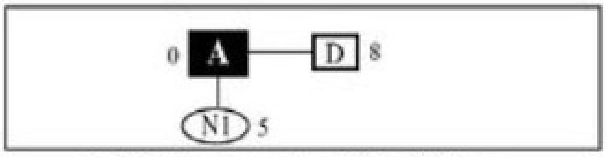
> A router sends **link-state advertisement (LSA)** packets to advertise its state periodically (every 30 minutes) and immediately when the router state changes. Information about attached interfaces, metrics used, and other variables is included in OSPF LSAs. As OSPF routers accumulate linkstate information, they use the shortest path first (SPF) algorithm to calculate the shortest path to each node.
>
> A **topological (link-state) database** is, essentially, an overall picture of networks in relation to routers. The topological database contains the collection of LSAs received from all routers in the same area. Because routers within the same area share the same information, they have identical topological databases.

Each router has its own view of the topology, even though all of the routers build a shortest-path tree using the same link-state database. When the LS database is synchronized within an area, the Dijkstra algorithm is run against it.

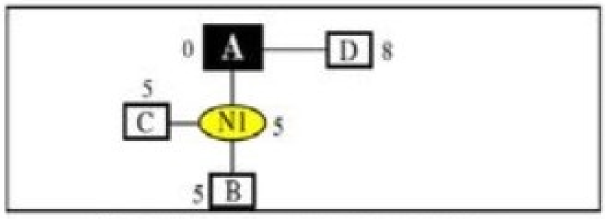
The SPF algorithm places each router at the root of a tree and calculates the shortest path to each node, using Dijkstra's algorithm, based on the cumulative cost that is required to reach that destination. LSAs are flooded throughout the area using a reliable algorithm, which ensures that all routers in an area have the same topological database. Each router uses the information in its topological database to calculate a shortest path tree, with itself as the root. The router then uses this tree to route network traffic.



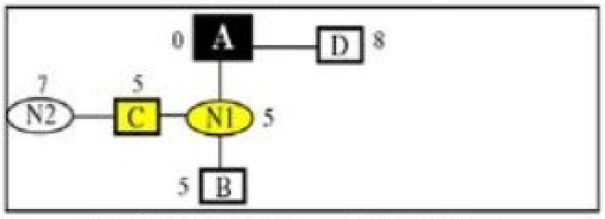
a. Start with A



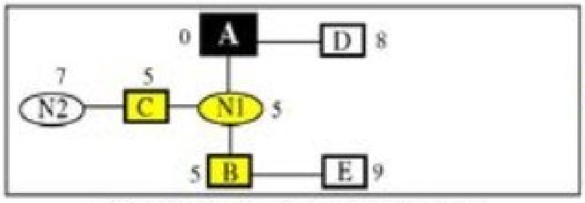
b. Make A permanent, add its neighbors



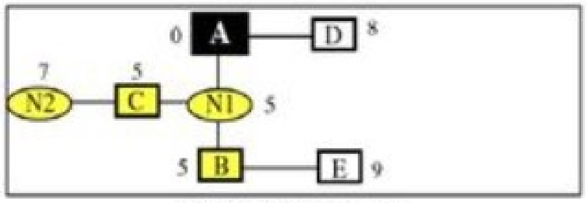
c. Make N1 permanent, add its neighbors



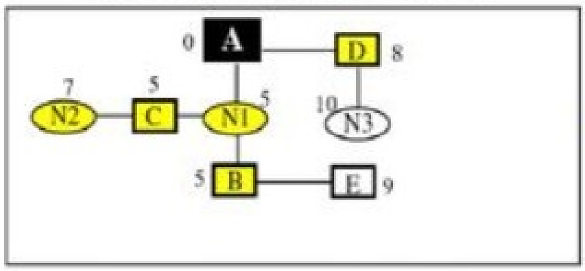
d. Make C permanent, add its neighbors



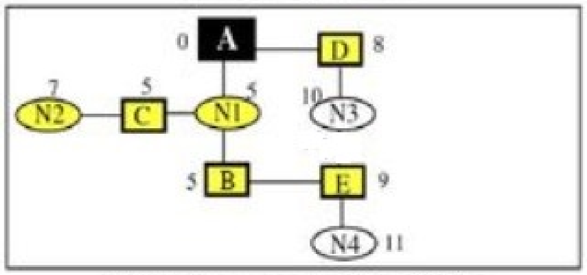
e. Make B permanent, add its neighbors



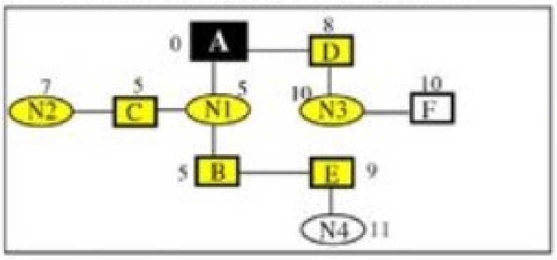
f. Make N2 permanent



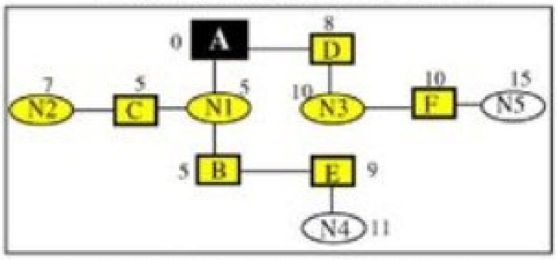
g. Make D permanent, add its neighbors



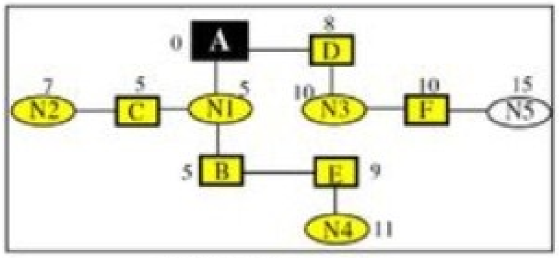
h. Make E permanent, add its neighbors



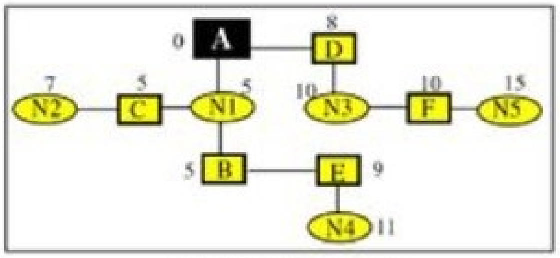
i. Make N3 permanent, add its neighbors



j. Make F permanent, add its neighbors



k. Make N4 permanent



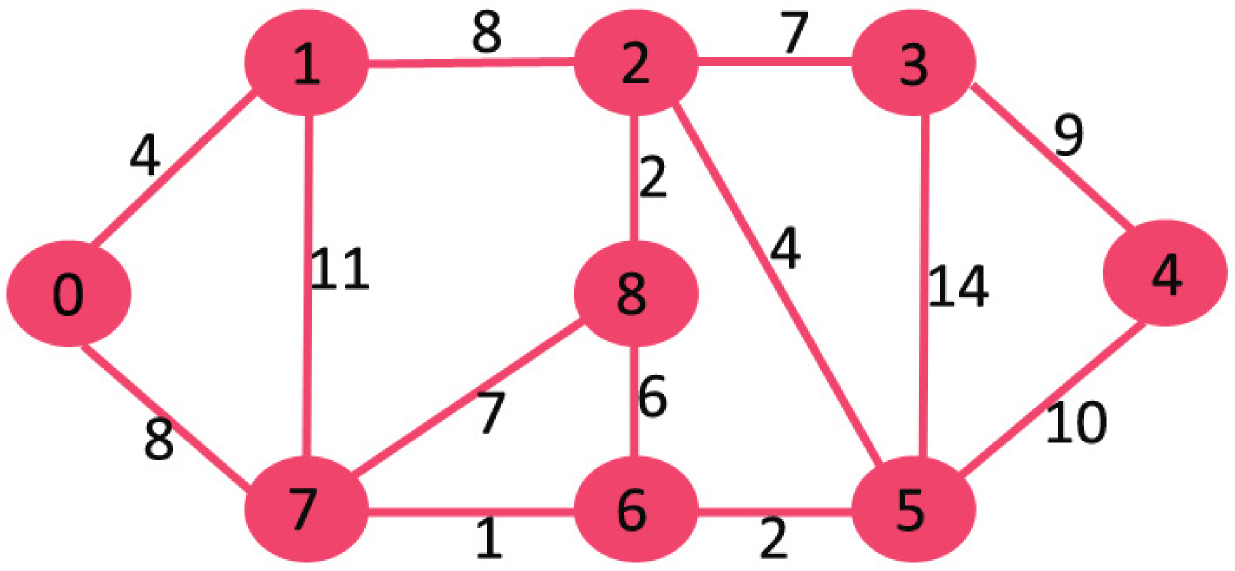
l. Make N5 permanent

Dijkstra’s shortest path algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. Dijkstra's Algorithm works on the basis that any subpath `B -> D` of the shortest path `A -> D` between vertices A and D is also the shortest path between vertices B and D. Djikstra used this property in the opposite direction i.e. we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Given: A graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.



Algorithm:

1. Create a set `sptSet` (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as `INFINITE`. Assign distance value as 0 for the source vertex so that it is picked first. Let this array be called `dist`.

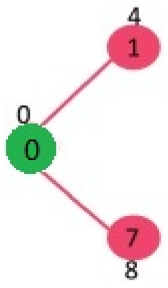
The set sptSet is initially empty and distances assigned to vertices are `dist{0, INF, INF, INF, INF, INF, INF, INF}` where INF indicates infinite.

1. While sptSet doesn't include all vertices

   A. Pick a vertex `u` which is not there in `sptSet` and has a minimum distance value.
   B. Include `u` to sptSet.
   C. Update distance value of all adjacent vertices of `u`. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v,

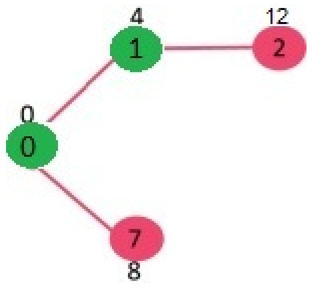
   ```
    if (dist[u] + weight of edge u-v < dist[v], then update the distance
   value of v as
        dist[v] = dist[u] + weight of edge u-v
   ```

Pass 1 The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including `0` to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown.

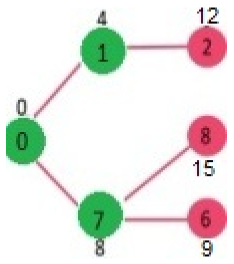


The vertices included in SPT are shown in green colour.

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

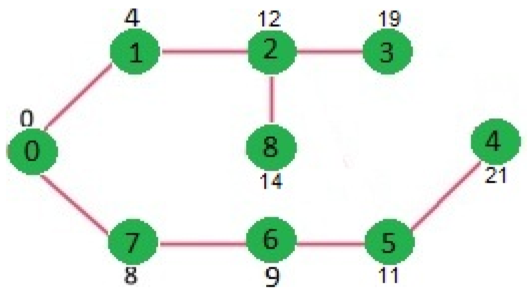


Pass 3 Vertex 7 is picked. So sptSet now becomes {0, 1, 7}.



Pass 4 Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}.

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



In [3]:

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
from sys import maxsize

class Graph():

        def __init__(self, vertices):
                self.V = vertices
                self.graph = [[0 for column in range(vertices)]
                                        for row in range(vertices)]

        def printSolution(self, dist):
                print ("Vertex".ljust(10),"Distance from Source".ljust(10))
                for node in range(self.V):
                        print (str(node).ljust(10), str(dist[node]).ljust(10))

        # A utility function to find the vertex with
        # minimum distance value, from the set of vertices
        # not yet included in shortest path tree
        def minDistance(self, dist, sptSet):

                # Initialize minimum distance for next node
                min = maxsize

                # Search not nearest vertex not in the
                # shortest path tree
                for u in range(self.V):
                        if dist[u] < min and sptSet[u] == False:
                                min = dist[u]
                                min_index = u

                return min_index

        # Function that implements Dijkstra's single source
        # shortest path algorithm for a graph represented
        # using adjacency matrix representation
        def dijkstra(self, src):

                dist = [maxsize] * self.V
                dist[src] = 0
                sptSet = [False] * self.V

                for cout in range(self.V):

                        # Pick the minimum distance vertex from
                        # the set of vertices not yet processed.
                        # x is always equal to src in first iteration
                        x = self.minDistance(dist, sptSet)

                        # Put the minimum distance vertex in the
                        # shortest path tree
                        sptSet[x] = True

                        # Update dist value of the adjacent vertices
                        # of the picked vertex only if the current
                        # distance is greater than new distance and
                        # the vertex in not in the shortest path tree
                        for y in range(self.V):
                                if self.graph[x][y] > 0 and sptSet[y] == False and dist
                                        dist[y] = dist[x] + self.graph[x][y]

                self.printSolution(dist)

# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
                [4, 0, 8, 0, 0, 0, 0, 11, 0],
                [0, 8, 0, 7, 0, 4, 0, 0, 2],
                [0, 0, 7, 0, 9, 14, 0, 0, 0],
                [0, 0, 0, 9, 0, 10, 0, 0, 0],
                [0, 0, 4, 14, 10, 0, 2, 0, 0],
                [0, 0, 0, 0, 0, 2, 0, 1, 6],
                [8, 11, 0, 0, 0, 0, 1, 0, 7],
                [0, 0, 2, 0, 0, 0, 6, 7, 0]
                ];

g.dijkstra(0);
```

```
Vertex      Distance from Source
0           0
1           4
2           12
3           19
4           21
5           11
6           9
7           8
8           14
```

Time Complexity: O(V^2) -> O($(E+V)\log V$) -> O($E + V\log V$)

The time complexity of the above code/algorithm looks `O(V^2)` as there are two nested while loops. If we use `adjacency list` instead of the matrix, we will need to run the statements in inner loop `k` times, where `k` is the number of edges on the selected vertex.

Further if we use a `heap` to store the nodes whose distance hasn't been finalized, the outer loop runs `V` times only but updates to the heap will cost `O(logV)` time. Also additional `O(VLogV)` time for heap creation. So overall time complexity is

`O(VLogV) + O(k1 +k2 +k3 + ..... +kv) * O(LogV)` which is `O((E*LogV) + (V*LogV))`

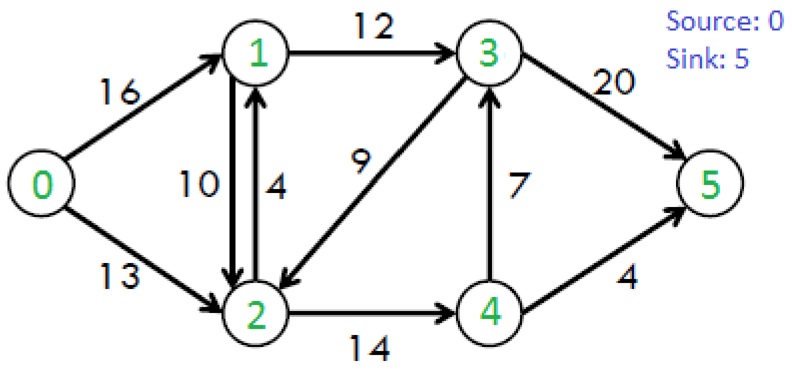
Further if we switch from Binary Heap for `Priority Queue`, Time complexity can be reduced to `O(E + VLogV)`. The reason is, Fibonacci Heap takes `O(1)` time for update operations while

Binary Heap takes `O(Logn)` time.

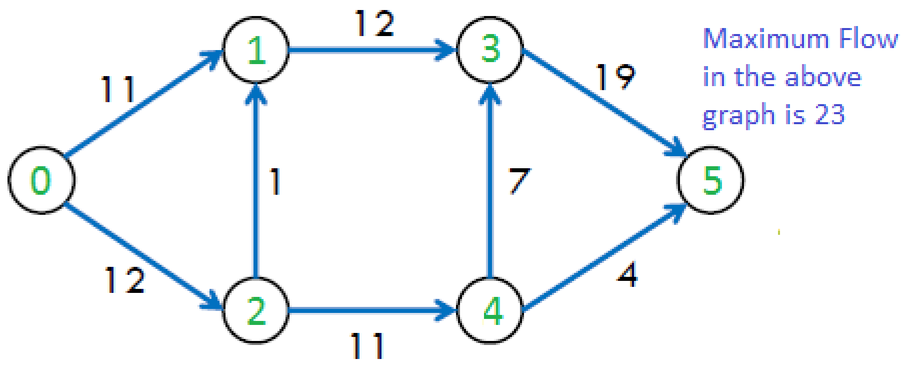
Transport Layer

It is responsible for end-to-end communication and error-free delivery of data. It shields the upper-layer applications from the complexities of data. The main protocol at this layer, the TCP performs sequencing and segmentation of data. It also has acknowledgment feature and controls the flow of the data through flow control mechanism which ensure the sender is not overwhelming the receiver with more data than it can handle. It also uses a network congestion-avoidance algorithm that includes various aspects of an `additive increase/multiplicative decrease (AIMD) scheme`, along with other schemes including slow start and congestion window, to achieve congestion avoidance. The TCP congestion-avoidance algorithm is the primary basis for congestion control in the Internet. To illustrate the functioning congestion control, we pose the problem of `maximum-flow` possible through a network, and look at an algorithm that describes an optimal solution.

Maximum flow problems involve finding a feasible flow through a single-source, single-sink flow network that is maximum. Let's take an image to explain how the above definition wants to say.



Each edge is labeled with capacity, the maximum amount of stuff that it can carry. The goal is to figure out how much stuff can be pushed from the vertex s(source) to the vertex t(sink).



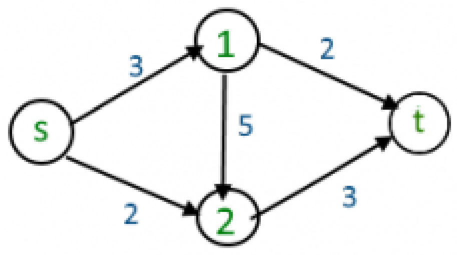
maximum flow possible is : 23

Following are different approaches to solve the problem :

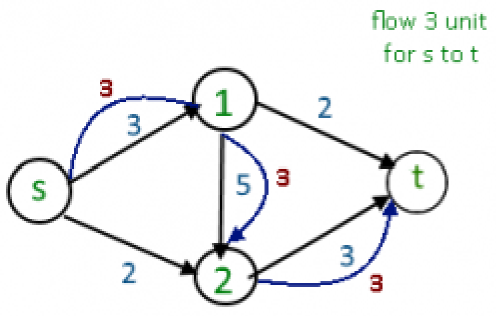
1. Naive Greedy Algorithm Approach (May not produce an optimal or correct result) Greedy approach to the maximum flow problem is to start with the all-zero flow and greedily produce flows with ever-higher value. The natural way to proceed from one to the next is to send more flow on some path from s to t How Greedy approach work to find the maximum flow :

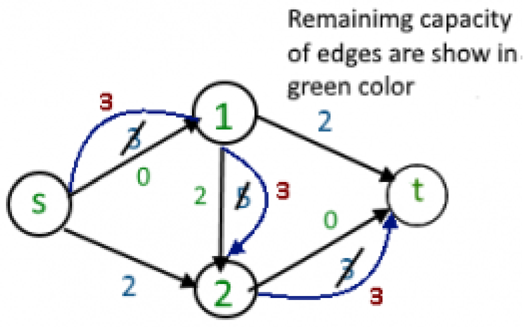
```
E number of edge
f(e) flow of edge
C(e) capacity of edge

1) Initialize : max_flow = 0
                f(e) = 0 for every edge 'e' in E

2) Repeat search for an s-t path P while it exists.
   a) Find if there is a path from s to t using BFS
      or DFS. A path exists if f(e) < C(e) for
      every edge e on the path.
   b) If no path found, return max_flow.
   c) Else find minimum edge value for path P

      // Our flow is limited by least remaining
      // capacity edge on path P.
      (i) flow = min(C(e)- f(e)) for path P ]
             max_flow += flow
      (ii) For all edge e of path increment flow
             f(e) += flow
```

3) Return max_flow Note that the path search just needs to determine whether or not there is an s-t path in the subgraph of edges e with f(e) < C(e). This is easily done in linear time using BFS or DFS.

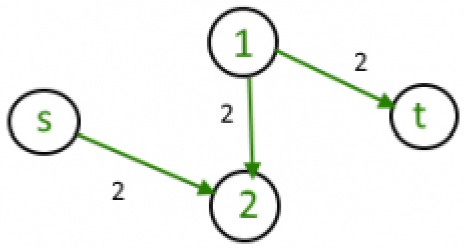


There is a path from source (s) to sink(t) `[s -> 1 -> 2 -> t]` with maximum flow 3 unit (path show in blue color)



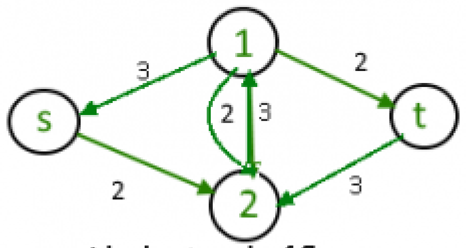


After removing all useless edge from graph it's look like



For above graph there is no path from source to sink so maximum flow : 3 unit But maximum flow is 5 unit. to over come form this issue we use residual Graph.

1. Residual Graphs The idea is to extend the naive greedy algorithm by allowing "undo" operations. For example, from the point where this algorithm gets stuck in above image, we'd like to route two more units of flow along the edge (s, 2), then backward along the edge (1, 2), undoing 2 of the 3 units we routed the previous iteration, and finally along the edge (1,t) maximum `backward edge : ( f(e) )` and `forward edge : ( C(e) - f(e) )`



residual network of flow

We need a way of formally specifying the allowable `“undo”` operations. This motivates the following simple but important definition, of a residual network. The idea is that, given a graph G and a flow f in it, we form a new flow network Gf that has the same vertex set of G and that has two edges for each edge of G. An edge e = `(1,2)` of G that carries flow f(e) and has capacity C(e) (for above image) spawns a “forward edge” of Gf with capacity `C(e)-f(e)` (the room remaining) and a `“backward edge” (2,1)` of Gf with capacity f(e) (the amount of previously routed flow that can be undone). source(s)- sink(t) paths with `f(e) < C(e)` for all edges, as searched for by the naive greedy algorithm, corresponding to the special case of s-t paths of Gf that comprise only forward edges.

Ford-Fulkerson Algorithm for Maximum Flow Problem

Given: a graph which represents a flow network where every edge has a capacity. Also given two vertices source ‘s’ and sink ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:

1. Flow on an edge doesn’t exceed the given capacity of the edge.
2. Incoming flow is equal to outgoing flow for every vertex except s and t.

Algorithm:

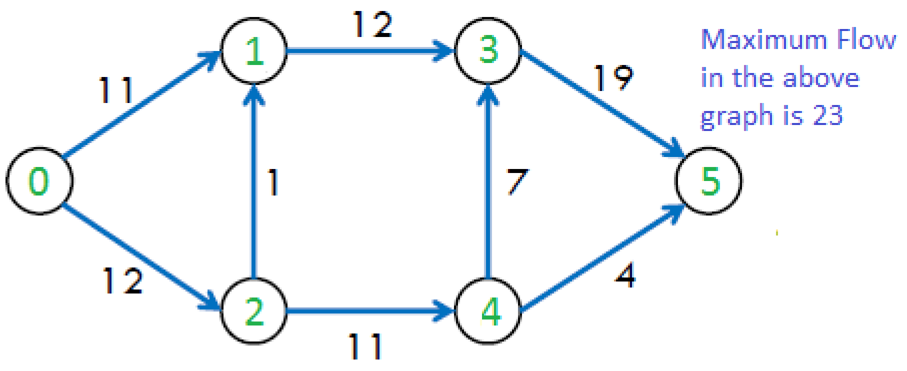
1. Start with initial flow as 0.
2. While there is a augmenting path from source to sink. Add this path-flow to flow.
3. Return flow.

> **Residual Graph** of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called residual capacity which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.

Implementation details:

1. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity.
2. To find an augmenting path, we can either do a `BFS` or `DFS` of the residual graph. We have used BFS in below implementation.
3. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent array. Using the parent array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.
4. The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We

need to add path flow along reverse edges because may later need to send flow in reverse direction



In [4]:

```python
# Python program for implementation
# of Ford Fulkerson algorithm

from collections import defaultdict

# This class represents a directed graph
# using adjacency matrix representation
class Graph:

        def __init__(self, graph):
                self.graph = graph # residual graph
                self. ROW = len(graph)
                # self.COL = len(gr[0])

        '''Returns true if there is a path from source 's' to sink 't' in
        residual graph. Also fills parent[] to store the path '''

        def BFS(self, s, t, parent):

                # Mark all the vertices as not visited
                visited = [False]*(self.ROW)

                # Create a queue for BFS
                queue = []

                # Mark the source node as visited and enqueue it
                queue.append(s)
                visited[s] = True

                # Standard BFS Loop
                while queue:

                        # Dequeue a vertex from queue and print it
                        u = queue.pop(0)

                        # Get all adjacent vertices of the dequeued vertex u
                        # If a adjacent has not been visited, then mark it
                        # visited and enqueue it
                        for ind, val in enumerate(self.graph[u]):
                                if visited[ind] == False and val > 0:
                                        # If we find a connection to the sink node,
                                        # then there is no point in BFS anymore
                                        # We just have to set its parent and can return
                                        queue.append(ind)
                                        visited[ind] = True
                                        parent[ind] = u
                                        if ind == t:
                                                return True

                # We didn't reach sink in BFS starting
                # from source, so return false
                return False


        # Returns tne maximum flow from s to t in the given graph
        def FordFulkerson(self, source, sink):

                # This array is filled by BFS and to store path
                parent = [-1]*(self.ROW)

                max_flow = 0 # There is no flow initially

                # Augment the flow while there is path from source to sink
                while self.BFS(source, sink, parent) :

                        # Find minimum residual capacity of the edges along the
                        # path filled by BFS. Or we can say find the maximum flow
                        # through the path found.
                        path_flow = float("Inf")
                        s = sink
                        while(s != source):
                                path_flow = min (path_flow, self.graph[parent[s]][s])
                                s = parent[s]

                        # Add path flow to overall flow
                        max_flow += path_flow

                        # update residual capacities of the edges and reverse edges
                        # along the path
                        v = sink
                        while(v != source):
                                u = parent[v]
                                self.graph[u][v] -= path_flow
                                self.graph[v][u] += path_flow
                                v = parent[v]

                return max_flow


# Create a graph given in the above diagram

graph = [[0, 16, 13, 0, 0, 0],
                [0, 0, 10, 12, 0, 0],
                [0, 4, 0, 0, 14, 0],
                [0, 0, 9, 0, 0, 20],
                [0, 0, 0, 7, 0, 4],
                [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0; sink = 5

print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))
```

```
The maximum possible flow is 23
```

Time Complexity: O(max_flow * E)

- We run a loop while there is an augmenting path. Finding a path using DFS/BFS takes O(E) for a graph with E edges. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow * E).

Application Layer

Comprised of the Application, Presentation and Session Layer, it is responsible for node-to-node communication and controls user-interface specifications While using a service from any server application, the client and server exchange a lot of information on the underlying intranet or Internet. Since, these information transactions are vulnerable to various attacks, a major role of the protocols at this layer is to ensure network security, which entails securing data against attacks while it is in transit on a network. Protocols like `SSH (Secure Shell)` and `SSL/TLS (Secure Sockets Layer/ Transport Layer Security` are used as protection layersover protocols like the HTTP that employ encryption schemes like `RSA, ECC, 3DES, AES`.

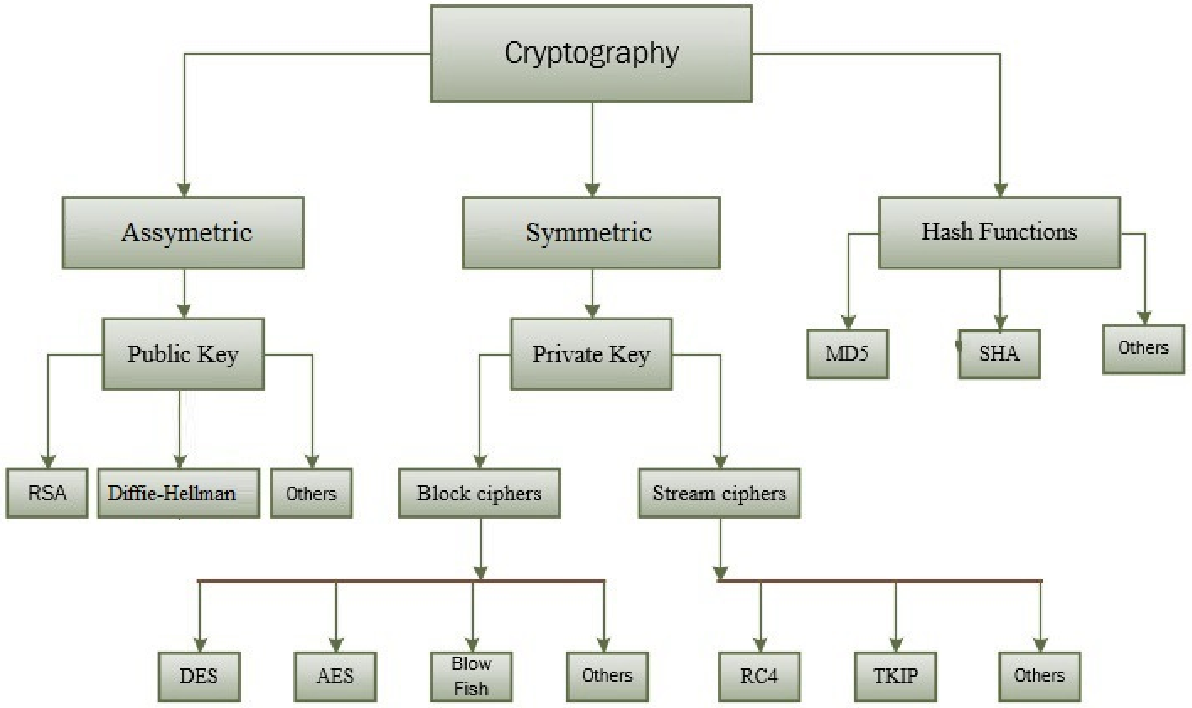
> **SSL** stands for Secure Sockets Layer and, in short, it's the standard technology for keeping an internet connection secure and safeguarding any sensitive data that is being sent between two systems, preventing criminals from reading and modifying any information transferred, including potential personal details. The two systems can be a server and a client (for example, a shopping website and browser) or server to server (for example, an application with personal identifiable information or with payroll information). It does this by making sure that any data transferred between users and sites, or between two systems remain impossible to read.
>
> It uses encryption algorithms to scramble data in transit, preventing hackers from reading it as it is sent over the connection. This information could be anything sensitive or personal which can include credit card numbers and other financial information, names and addresses.
>
> **TLS** (Transport Layer Security) is just an updated, more secure, version of SSL. The most up to date TLS certificates have options of **ECC, RSA or DSA** encryption. **HTTPS (Hyper Text Transfer Protocol Secure)** appears in the URL when a website is secured by an SSL certificate. The details of the certificate, including the issuing authority and the corporate name of the website owner, can be viewed by clicking on the lock symbol on the browser bar.

In the following section, we’ll look at the application and process of cryptography in order to develop a basic understanding of the concepts that lie under the particular algorithms and how they are categorized.

Cryptography

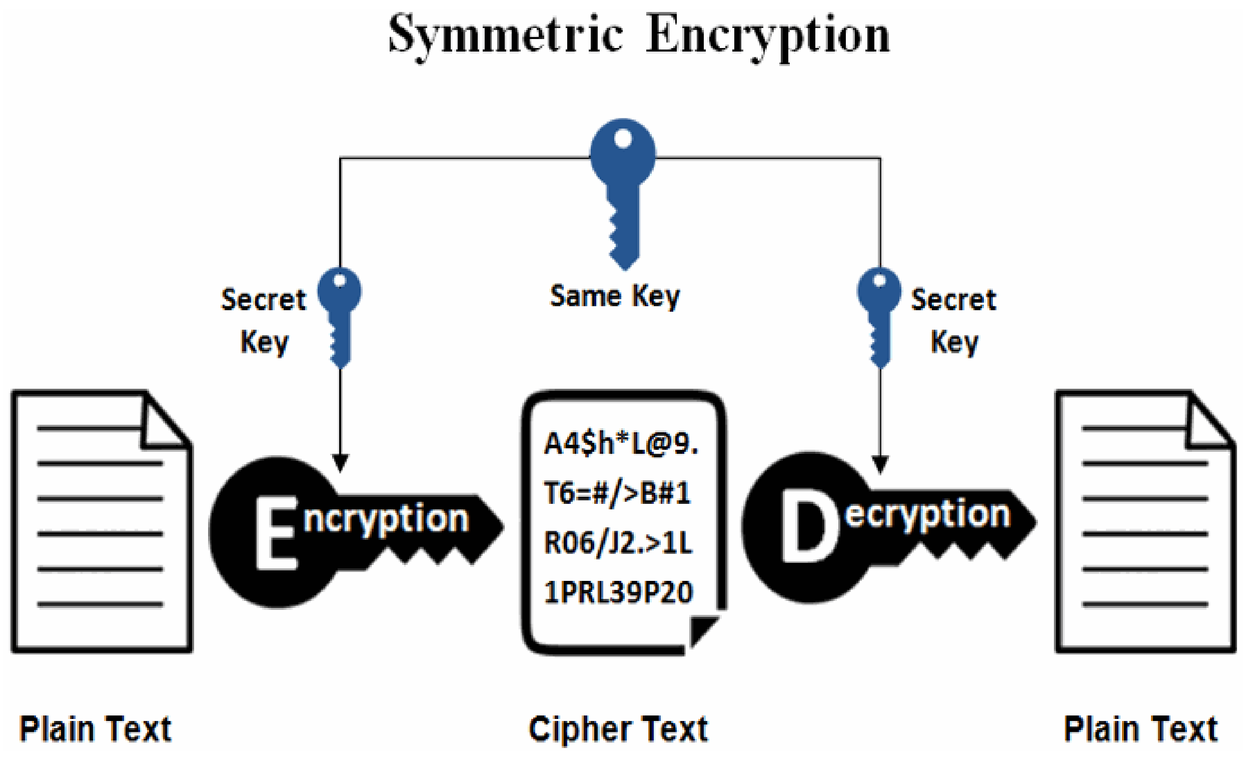
Encryption is a method of converting data into an undecipherable format so that only the authorized parties can access the information.

Cryptographic keys, in conjunction with encryption algorithms, are what makes the encryption process possible. And, based on the way these keys are applied, there are two major types of encryption schemes that are widely used: `symmetric encryption` (where a **single secret key** is used to encrypt and decrypt data) and `asymmetric encryption` (where a **public key cryptosystem** is used and encryption and decryption is done using a pair of public and corresponding private key). Both of these methods use different mathematical algorithms (i.e., those encryption algorithms we mentioned moments ago) to scramble the data.



Symmetric Encryption

Symmetric encryption schemes use the same symmetric key (or password) to encrypt data and decrypt the encrypted data back to its original form:



Symmetric encryption usually combines several crypto algorithms into an symmetric encryption scheme, **e.g. AES-256-CTR-HMAC-SHA256**. The encryption scheme (cipher construction) may include:

1. password to key derivation algorithm (with certain parameters)
2. symmetric cipher algorithm (with certain parameters)
3. cipher block mode algorithm + message authentication (MAC) algorithm.

This means that the above shown diagram is simplified and does not fully represent the process.

> **Secret Keys** The secret key used to cipher (encrypt) and decipher (decrypt) data is typically of size `128, 192 or 256 bits` and is sometimes referred as "encryption key" or "shared key", because both sending and receiving parties should know it. Most applications use a password-to-key-derivation scheme to extract a secret key from certain password, because users tend to remember passwords easier than binary data. Additionally, message authentication is often incorporated along with the encryption to provide integrity and authenticity (this encryption approach is known as "authenticated encryption").
>
> In many systems (e.g. public blockchains, PGP, OpenSSL and others) secret keys are encoded as `base58` or `base64` for shorter string representation.
>
> > Consider a 256-bit secret key, encoded as hex string: `02c324648931b89e3e8a0fc42c96e8e3be2e42812986573a40d46563bceaf75110`
> >
> > The above key looks like this in base58: `pbPRqYDxnKZfs8j4KKiqYmx6nzipAjTJf1oCD1WKgy99`

What Makes Symmetric Encryption a Great Technique? The most outstanding feature of symmetric encryption is the simplicity of its process. This simplicity of this type of encryption lies in the use of a single key for both encryption as well as decryption. As a result, symmetric encryption algorithms:

1. Are significantly `faster` than their asymmetric encryption counterparts (which we'll discuss shortly),
2. Require `less computational power`, and
3. `Don't dampen` internet speed.

This means that when there's a large chunk of data to be encrypted, symmetric encryption proves to be a great option. We now look at one of the most common Symmetric Encryption algorithm `AES`

Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data, widely used today as it is much stronger than DES and triple DES despite being harder to implement.

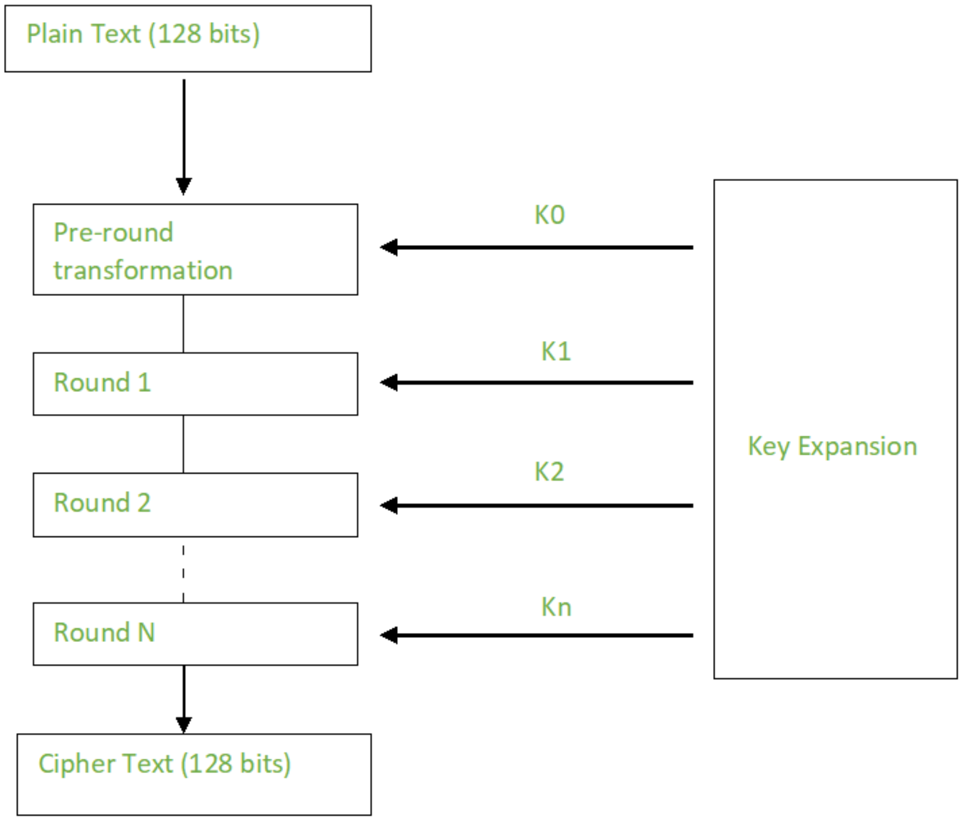
- AES is a block cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of 128 bits each. That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text as output. AES relies on substitution-permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data.

Working of the cipher: AES performs operations on bytes of data rather than in bits. Since the block size is 128 bits, the cipher processes 128 bits (or 16 bytes) of the input data at a time.

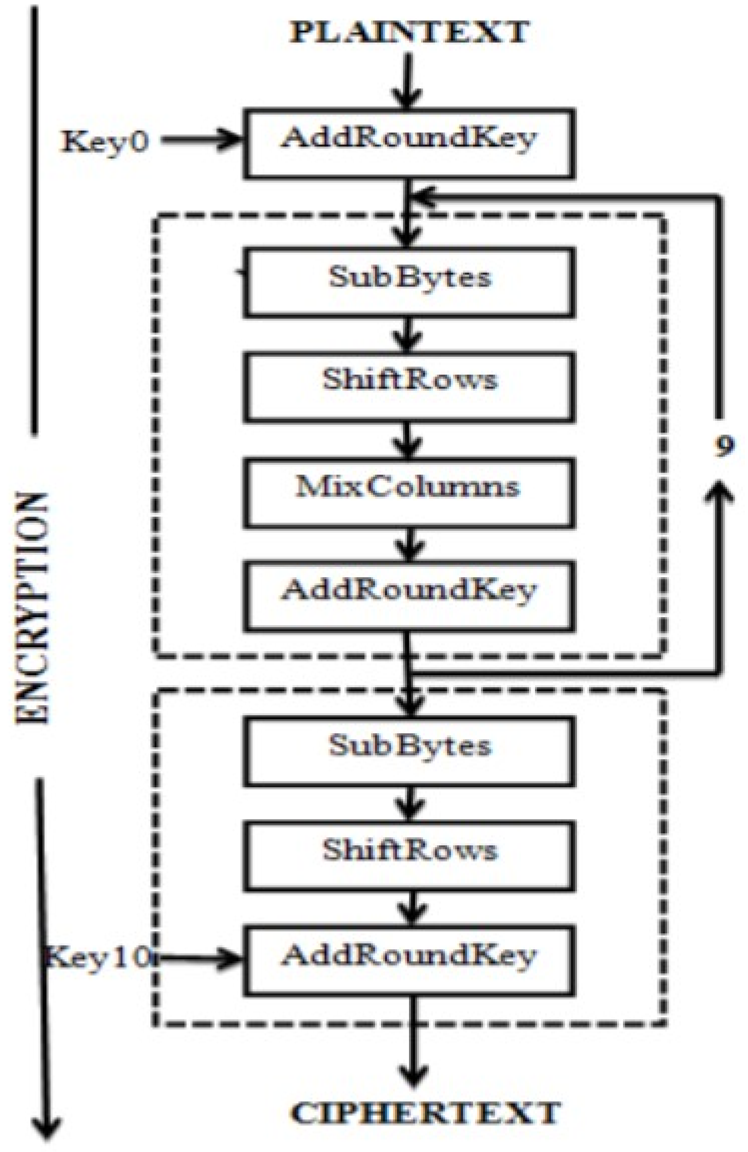
The number of rounds depends on the key length as follows :

- 128 bit key – 10 rounds
- 192 bit key – 12 rounds
- 256 bit key – 14 rounds

Creation of Round keys: A Key Schedule algorithm is used to calculate all the round keys from the key. So the initial key is used to create many different round keys which will be used in the corresponding round of the encryption.


Plain Text (128 bits)
Pre-round transformation
Round 1
Round 2
Round N
Cipher Text (128 bits)
K0
K1
K2
Kn
Key Expansion

Encryption:



`AES` considers each 128 bit block as a `(4 byte x 4 byte) = 16 byte grid` in a column major arrangement.

```
[ b0 | b4 | b8 | b12 ]
| b1 | b5 | b9 | b13 |
| b2 | b6 | b10| b14 |
[ b3 | b7 | b11| b15 ]
```

Each round comprises of 4 steps :

1. SubBytes
2. ShiftRows
3. MixColumns
4. Add Round Key The last round doesn't have the MixColumns round.

The SubBytes does the substitution and ShiftRows and MixColumns performs the permutation in the algorithm.

SubBytes: This step implements the substitution.

In this step each byte is substituted by another byte. It is performed using a lookup table also called the S-box. This substitution is done in a way that a byte is never substituted by itself and also not substituted by another byte which is a compliment of the current byte.The result of this step is a 16 byte (4 x 4) matrix like before.

The next two steps implement the permutation.

ShiftRows: This step is just as it sounds. Each row is shifted a particular number of times.

- The first row is not shifted
- The second row is shifted once to the left.
- The third row is shifted twice to the left.
- The fourth row is shifted thrice to the left. (A left circular shift is performed.)

```
[ b0  | b1  | b2  | b3  ]         [ b0  | b1  | b2  | b3  ]
| b4  | b5  | b6  | b7  |    ->   | b5  | b6  | b7  | b4  |
| b8  | b9  | b10 | b11 |         | b10 | b11 | b8  | b9  |
[ b12 | b13 | b14 | b15 ]         [ b15 | b12 | b13 | b14 ]
```

MixColumns: This step is basically matrix multiplication. Each column is multiplied with a specific matrix and thus the position of each byte in the column is changed as a result.

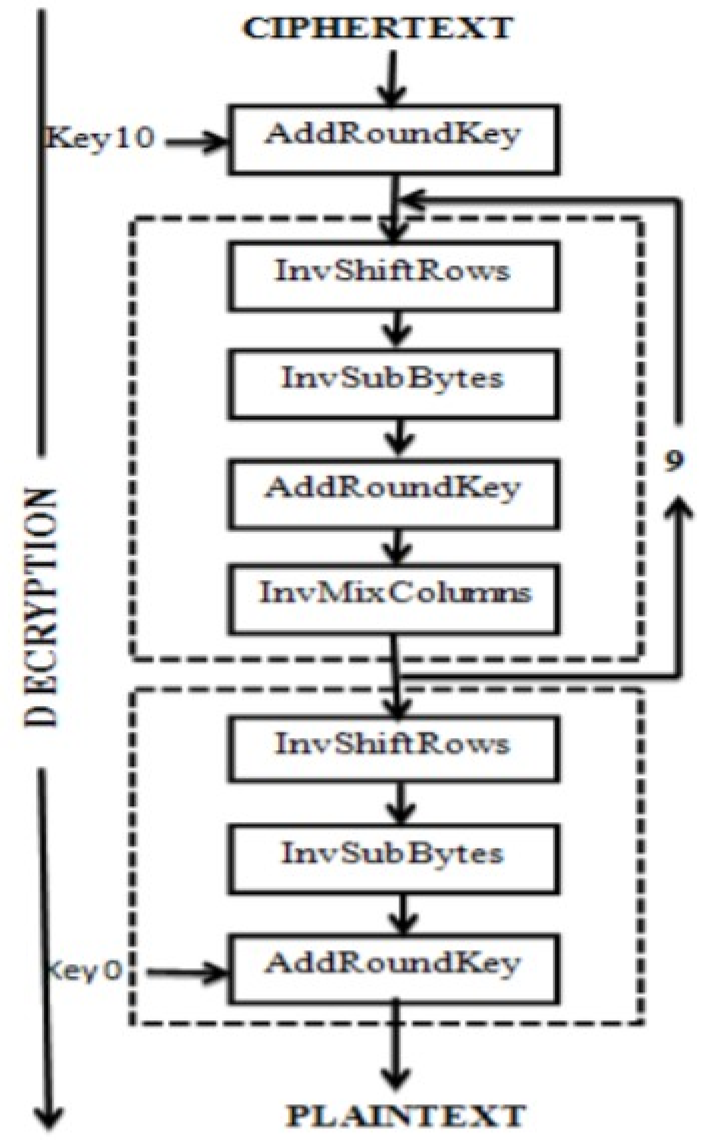
This step is skipped in the last round.

```
[ c0 ]         [ 2  3  1  1 ]     [ b0 ]
| c1 |  =      | 1  2  3  1 |     | b1 |
| c2 |         | 1  1  2  3 |     | b2 |
[ c3 ]         [ 3  1  1  2 ]     [ b3 ]
```

Add Round Keys: Now the resultant output of the previous stage is XOR-ed with the corresponding round key. Here, the 16 bytes is not considered as a grid but just as 128 bits of data.

After all these rounds 128 bits of encrypted data is given back as output. This process is repeated until all the data to be encrypted undergoes this process.

Decryption:



The stages in the rounds can be easily undone as these stages have an opposite to it which when performed reverts the changes. Each block goes through the 10, 12 or 14 rounds depending on the key size.

The stages of each round in decryption is as follows :

1. Add round key
2. Inverse MixColumns
3. ShiftRows
4. Inverse SubByte The decryption process is the encryption process done in reverse so I will explain the steps with notable differences.

Inverse MixColumns: This step is similar to the MixColumns step in encryption, but differs in the matrix used to carry out the operation.

```
[ b0 ]        [ 14  11  13  9  ]     [ c0 ]
| b1 |  =     | 9   14  11  13 |     | c1 |
| b2 |          | 13  9   14  11 |      | c2 |
[ b3 ]          [ 11  13  9   14 ]      [ c3 ]
```

Inverse SubBytes: Inverse S-box is used as a lookup table and using which the bytes are substituted during decryption.

Summary:

`AES` instruction set is now integrated into the CPU (offers throughput of several GB/s)to improve the speed and security of applications that use AES for encryption and decryption. Even though it's been 20 years since its introduction, we have failed to break the AES algorithm as it is infeasible even with the current technology. Till date the only vulnerability remains in the implementation of the algorithm.

In [5]:

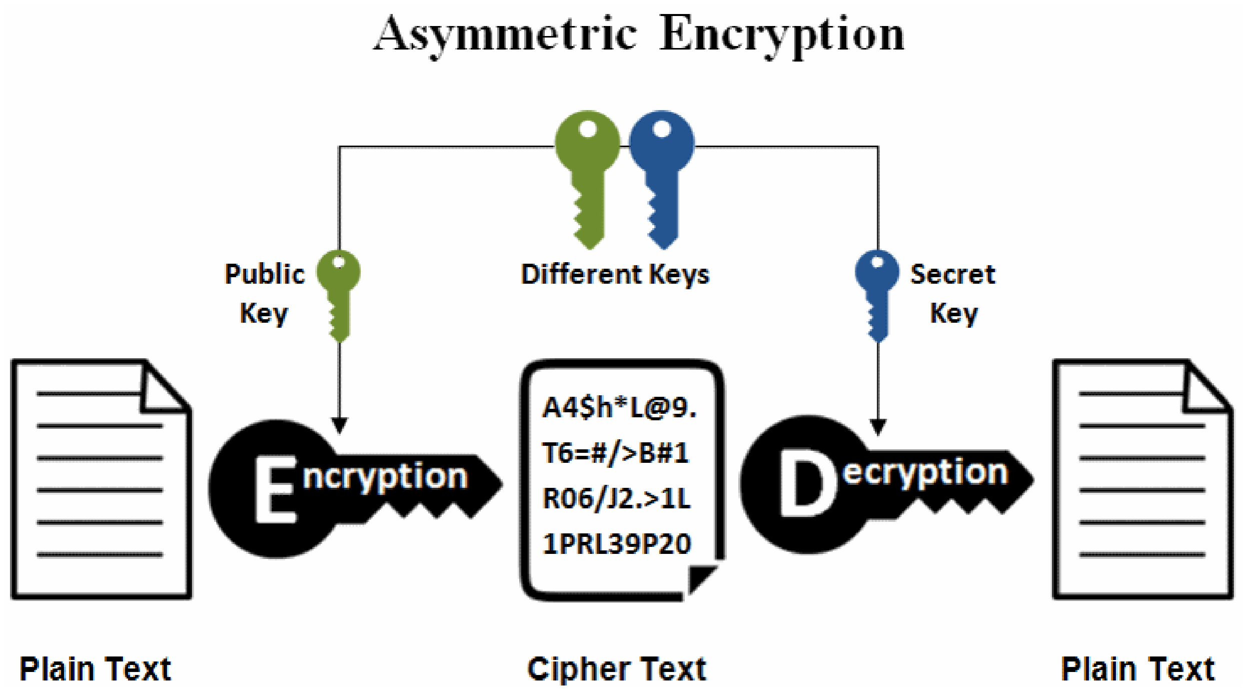
```python
import hashlib
from Crypto import Random
from Crypto.Cipher import AES
from base64 import b64encode, b64decode

class AESCipher(object):
    def __init__(self, key):
        self.block_size = AES.block_size
        self.key = hashlib.sha256(key.encode()).digest()

    def encrypt(self, plain_text):
        plain_text = self.__pad(plain_text)
        iv = Random.new().read(self.block_size)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        encrypted_text = cipher.encrypt(plain_text.encode())
        return b64encode(iv + encrypted_text).decode("utf-8")

    def decrypt(self, encrypted_text):
        encrypted_text = b64decode(encrypted_text)
        iv = encrypted_text[:self.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        plain_text = cipher.decrypt(encrypted_text[self.block_size:]).decode("utf-8")
        return self.__unpad(plain_text)

    def __pad(self, plain_text):
        number_of_bytes_to_pad = self.block_size - len(plain_text) % self.block_size
        ascii_string = chr(number_of_bytes_to_pad)
        padding_str = number_of_bytes_to_pad * ascii_string
        padded_plain_text = plain_text + padding_str
        return padded_plain_text

    @staticmethod
    def __unpad(plain_text):
        last_character = plain_text[len(plain_text) - 1:]
        return plain_text[:-ord(last_character)]


AED_key="password"
plain_text = "hello, world"
obj = AESCipher(AED_key);
encrypted_text = obj.encrypt(plain_text)
```

```
decrypted_text = obj.decrypt(encrypted_text)
print("Plain text:", plain_text)
print("AED key:", AED_key)
print("Encrypted text:", encrypted_text)
print("Decrypted text:", decrypted_text)
```

```
Plain text: hello, world
AED key: password
Encrypted text: ahI14DsuWd7eKG6QrrMV2FTViQ7JlKqZZVsvIwaCS/A=
Decrypted text: hello, world
```

Asymmetric Encryption

Asymmetric encryption schemes use a pair of cryptographically related public and private keys to encrypt the data (by the public key) and decrypt the encrypted data back to its original forms (by the private key). Data encrypted by a public key is decrypted by the corresponding private key:



Public key encryption can work also in the opposite scenario: encrypt data by a private key and decrypt it by the public key. Thus someone can prove that he is owner of certain private key, while revealing only its corresponding public key. This approach is used by some `digital signature` schemes.

> **Signatures: Asymmetric Signing / Verification** `Digital signature` schemes typically use a public-key cryptosystem `(such as RSA or ECC)` and use a public / private key pairs. A message is signed by a private key and the signature is verified by the corresponding public key:
>
> > **Messages are signed by the sender using a private key (signing key)**.
>
> Typically the input message is hashed and then the signature is calculated by the signing algorithm. Most signature algorithms perform some calculation with the

`message hash + the signing key` in a way that the result cannot be calculated without the signing key. The result from message signing is the digital signature (one or more integers):

`signMsg(msg, privKey) → signature`

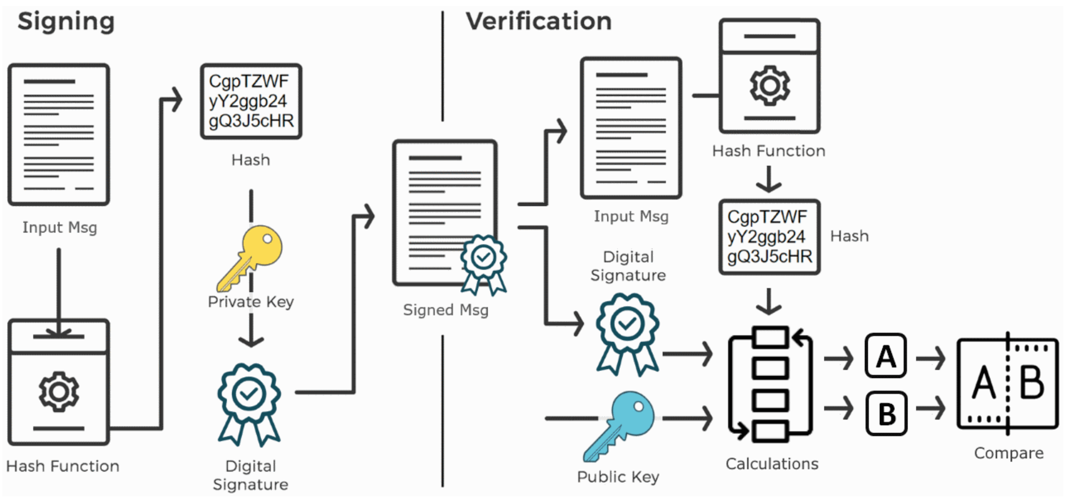
> **Message signatures are verified by the corresponding public key (verification key)**.

Typically the signed message is hashed and some calculation is performed by the signature algorithm using the `message hash + the public key`. The result from signing is a boolean value (valid or invalid signature):

`verifyMsgSignature(msg, signature, pubKey) → valid / invalid`

A message signature mathematically guarantees that certain message was signed by certain (secret) private key, which corresponds to certain (non-secret) public key. After a message is signed, the message and the signature cannot be modified and thus message authentication and integrity is guaranteed. Anyone, who knows the public key of the message signer, can verify the signature. After signing the signature author cannot reject the act of signing (this is known as non-repudiation).

Most signature schemes work like it is shown at the following diagram:



At signing, the input message is hashed (either alone, or together with the public key and other input parameters), then some computation (based on elliptic curves, discrete logarithms or other cryptographic primitive) calculates the digital signature. The produced signed message consists of the original message + the calculated signature.

At signature verification, the message for verification is hashed (either alone or together with the public key) and some computations are performed between the message hash, the digital signature and the public key, and finally a comparison decides whether the signature is valid or not.

> **Private Keys**: Message encryption and signing is done by a private key. The private keys are always kept secret by their owner, just like passwords. In the server infrastructure, private key usually stay in an encrypted and protected keystore. In the blockchain systems the private keys usually stay in specific software or hardware apps or devices called **"crypto wallets"**, which store securely a set of private keys.
>
> **Public keys**: Message decryption and signature verification is done by the public key. Public keys are by design public information (not a secret). It is mathematically infeasible to calculate the private key from its corresponding public key. In many systems the public key is encapsulated in a digital certificate, which binds certain identity (e.g. person or Internet domain name) to certain public key. In blockchain systems public keys are usually published as parts of the blockchain transactions to help identify who has signed each transaction. In systems like `PGP` and `SSH` the public key is downloaded from the server once (after manual user verification) and is remembered for further use.

What Makes Symmetric Encryption a Great Technique?

1. The first (and most obvious) advantage of this type of encryption is the `security` it provides. In this method, the public key — which is publicly available — is used to encrypt the data, while the decryption of the data is done using the private key, which needs to be stored securely. This ensures that the data remains protected against man-in-the-middle `(MiTM) attacks`. For web/email servers that connect to hundreds of thousands of clients ever minute, asymmetric encryption is nothing less than a boon as they only need to manage and protect a single key. Another key point is that public key cryptography allows creating an encrypted connection without having to meet offline to exchange keys first.
2. The second crucial feature that asymmetric encryption offers is `authentication`. As we saw, the data encrypted by a public key can only be decrypted using the private key related to it. Therefore, it makes sure that the data is only seen and decrypted by the entity that's supposed to receive it. In simpler terms, it verifies that you're talking to the person or organization that you think you are.

We now look at some common Symmetric Encryption algorithms viz `RSA` and `D

Rivest, Shamir, Adleman (RSA)

The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024 bit keys could be broken in the near future. But till now it seems to be an infeasible task.

Generating Public Key = `{n, e}`:

1. Select two prime no's. Suppose P = 53 and Q = 59.
2. Now First part of the Public key : n = P*Q = 3127.
3. We also need a small exponent say e, where must be:
    - An integer.
    - Not be a factor of n.
    - 1 < e < Φ(n) = (P-1)(Q-1), Let us now consider it to be equal to 3.

Generating Private Key = `{d, k}` :

1. We need to calculate Φ(n) such that Φ(n) = (P-1)(Q-1),
   so, Φ(n) = 3016

2. Now calculate d : d = (k*Φ(n) + 1) / e for some integer k For k = 2, value of d is 2011. We are ready with our – `Public Key ( n = 3127 and e = 3)` and `Private Key(d = 2011)`

Now we will encrypt `“HI”` :

- Convert letters to numbers : H = 8 and I = 9
- Thus Encrypted Data c = 89e mod n.
- Thus our Encrypted Data comes out to be 1394

Now we will decrypt `1394` :

- Decrypted Data = cd mod n.
- Thus our Encrypted Data comes out to be 89
- 8 = H and I = 9 i.e. "HI".

To encrypt : $\boxed{x}$

Public Key : N, e

where $N = P \times Q$ (P, Q are prime)

$\gcd(e, N) = 1$

Encryption : generates $\boxed{c}$

where $c \equiv x^e \pmod N$

Private Key : d, K

where $d = (K(P-1)(Q-1) + 1)/e$

Decryption : $x \equiv c^d \pmod N$

Proof :

$$
\begin{aligned}
& c^d \pmod N \\
= \; & (x^e \pmod N)^d \pmod N \\
= \; & x^{ed} \pmod N \\
= \; & x^{K(P-1)(Q-1)+1} \pmod N \\
= \; & x \cdot \left[\left(\left(x^{P-1}\right)^{Q-1}\right)^K \bmod N\right] \\
= \; & x \cdot 1 \\
= \; & x
\end{aligned}
$$

$\because$ Fermat's Little Theorem

.

Fermat's Little Theorem: If p is prime, then $\forall\; 1 \leq a < p$, $a^{p-1} \equiv 1(\mod p)$

- The numbers $a.i \mod p$ are distinct $\because$ if $a.i \equiv a.j(\mod p)$ $\implies i \equiv j(\mod p)$
- The numbers are non-zero $\because a.i \equiv 0 \implies i \equiv 0$
- Hence:
  - $\{1, 2, 3, \ldots p-1\} = \{a.1 \mod p, a.2 \mod p, \ldots a.(p-1) \mod p\}$
  - $(p-1)! \equiv a^{p-1}.(p-1)!(\mod p)$

In [6]:

```python
# Returns gcd of a and b
def gcd(a, h):
    while (True):
        temp = a%h
        if temp == 0:
          return h
        a = h
        h = temp


# Code to demonstrate RSA algorithm
# Two random prime numbers
p = 53
q = 59

# First part of public key:
n = p*q

# Finding other part of public key.
# e stands for encrypt
e = 3
phi = (p-1)*(q-1)
while (e < phi):
    # e must be co-prime to phi and
    # smaller than phi.
    if (gcd(e, phi)==1):
        break
    else:
        e+=1

# Private key (d stands for decrypt)
# choosing d such that it satisfies
# d*e = 1 + k * totient
k = 2  # A constant value
d = (1 + (k*phi))//e

# Message to be encrypted
msg = 89

print("Message data = ", msg)

# Encryption c = (msg ^ e) % n
c = pow(msg, e, n)
print("Encrypted data = ", c)

# Decryption m = (c ^ d) % n
m = pow(c, d, n)
print("Original Message Sent = ", m)
```

```
Message data =  89
Encrypted data =  1394
Original Message Sent =  89
```

Diffie-Hellman Algorithm - ECC

`Elliptic Curve Cryptography (ECC)` is an approach to public-key cryptography, based on the algebraic structure of elliptic curves over finite fields. ECC requires a smaller key as compared to non-ECC cryptography to provide equivalent security (a 256-bit ECC security has equivalent security attained by 3072-bit RSA cryptography).

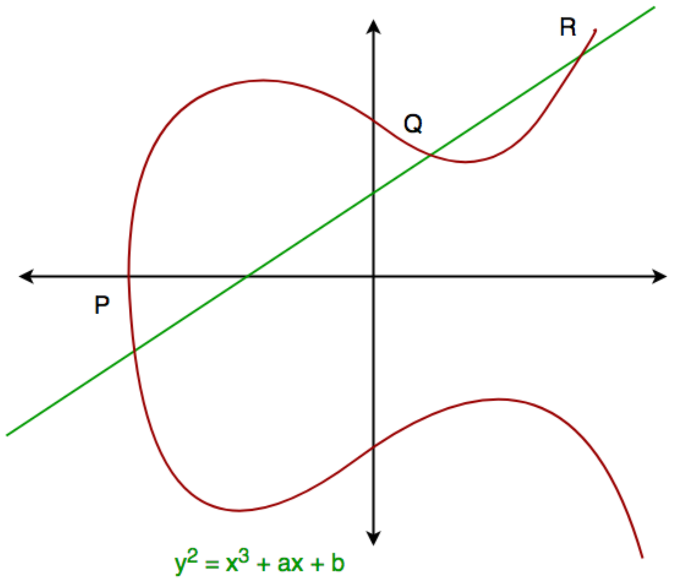
For a better understanding of Elliptic Curve Cryptography, it is very important to understand the basics of the Elliptic Curve. An elliptic curve is a planar algebraic curve defined by an equation of the form

$$y^2 = x^3 + ax + b$$

Where `‘a’` is the co-efficient of `x` and `‘b’` is the constant of the equation

The curve is non-singular; that is, its graph has no cusps or self-intersections (when the characteristic of the Co-efficient field is equal to 2 or 3).

In general, an elliptic curve looks like as shown below. Elliptic curves can intersect atmost 3 points when a straight line is drawn intersecting the curve. As we can see, the elliptic curve is symmetric about the x-axis. This property plays a key role in the algorithm.



This approach uses six tuple {P, a, b, G, n, h}

P = Field that the curve is define over

G = Generator point

a, b = Values define the curve

h = Co- factor

n = Prime order of G

The `Diffie-Hellman algorithm` is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime `P` and `G (a primitive root of P)` and two private values a and b.

> The **primitive root** of a prime number `n` is an integer `r` between `[1, n-1]` such that the values of $r^x(mod n)$ where $x \in [0, n-2]$ are all different. This is an **O(nlogn) complexity** operation by the following steps:
>
> 1. Euler Totient Function phi = n-1 (Assuming n is prime)
> 2. Find all prime factors of phi.
> 3. Calculate all powers to be calculated further using (ϕ/prime-factors) one by one.
> 4. Check for all numbers for all powers from i=2 to n-1 i.e. (i^ powers) modulo n.
> 5. If it is 1 then 'i' is not a primitive root of n.

> 6. If it is never 1 then return i;.

`P` and `G` are both publicly available numbers. Users (say Alice and Bob) pick private values `a` and `b` and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Algorithm:

```
___________Alice_______________________|_____________Bob_______________________

Public Keys available = P, G           | Public Keys available = P, G
Private Key Selected = a               | Private Key Selected = b
Key generated = x = G^a mod P          | Key generated = y = G^b mod P
Sends key to Bob                       | Sends key to Alice
Key received = y                       | key received = x
Generated Secret Key = k_a = y^a mod P | Generated Secret Key = k_b = x^b
mod P
```

Algebraically, it can be shown that `k_a = k_b` Users now have a symmetric secret key to encrypt

Example:

- Step 1: Alice and Bob get public numbers P = 23, G = 9
- Step 2: private key selection

```
- Alice selected a private key a = 4 and
- Bob selected a private key b = 3
```

- Step 3: Alice and Bob compute public values

```
- Alice:  x = (9^4 mod 23) = (6561 mod 23) = 6
- Bob:    y = (9^3 mod 23) = (729 mod 23) = 16
```

- Step 4: Alice and Bob exchange public numbers
- Step 5: key reception

```
- Alice receives public key y =16 and
- Bob receives public key x = 6
```

- Step 6: Alice and Bob compute symmetric keys

```
- Alice:  ka = y^a mod p = 65536 mod 23 = 9
- Bob:    kb = x^b mod p = 216 mod 23 = 9
```

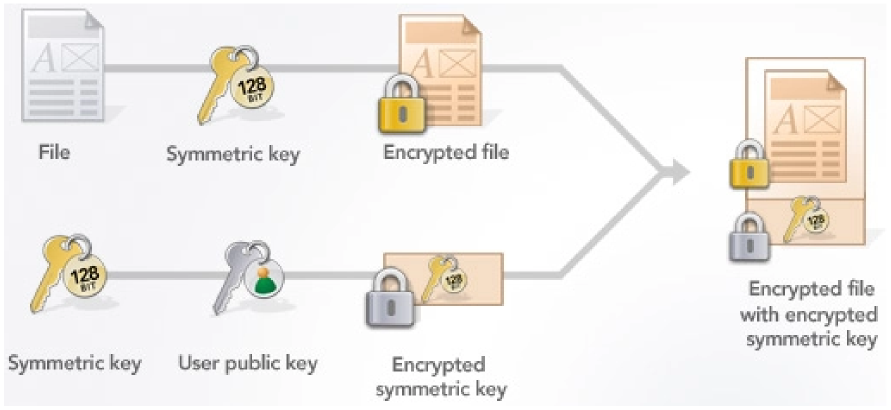
- Step 7: 9 is the shared secret.

In [7]:

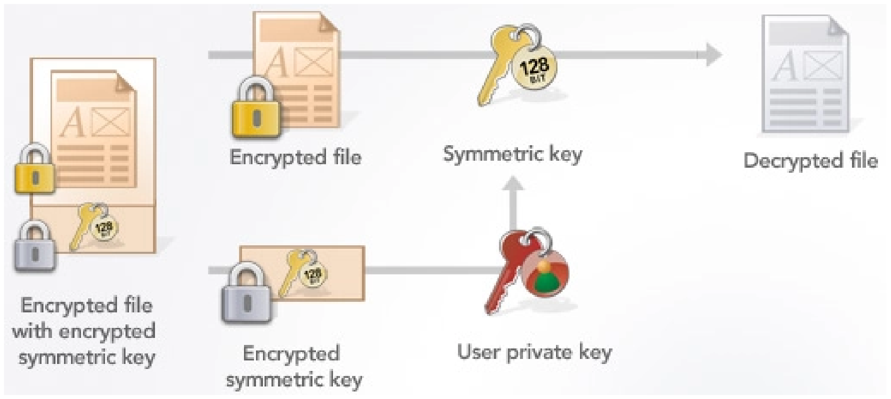
```python
from random import randint

if __name__ == '__main__':

        # Both the persons will be agreed upon the
        # public keys G and P
        # A prime number P is taken
        P = 23

        # A primitive root for P, G is taken
        G = 9


        print('The Value of P is :%d'%(P))
        print('The Value of G is :%d'%(G))

        # Alice will choose the private key a
        a = 4
        print('The Private Key a for Alice is :%d'%(a))

        # gets the generated key
        x = int(pow(G,a,P))

        # Bob will choose the private key b
        b = 3
        print('The Private Key b for Bob is :%d'%(b))

        # gets the generated key
        y = int(pow(G,b,P))


        # Secret key for Alice
        ka = int(pow(y,a,P))

        # Secret key for Bob
        kb = int(pow(x,b,P))

        print('Secret key for the Alice is : %d'%(ka))
        print('Secret Key for the Bob is : %d'%(kb))
```

```
The Value of P is :23
The Value of G is :9
The Private Key a for Alice is :4
The Private Key b for Bob is :3
Secret key for the Alice is : 9
Secret Key for the Bob is : 9
```

> > The **Elliptic curve cryptography (ECC)** does not directly provide encryption method. Instead, we can design a hybrid encryption scheme by using the ECDH (Elliptic Curve Diffie–Hellman) key exchange scheme to derive a shared secret key for symmetric data encryption and decryption.
>
> Typically, `public-key cryptosystems` can encrypt messages of limited length only and are slower than symmetric ciphers. For encrypting longer messages (e.g. PDF documents) usually a public-key encryption scheme (also known as **hybrid encryption scheme**) is used, which combines symmetric and asymmetric encryption like this:
>
> > For the **encryption** a random symmetric key `sk` is generated, the message is symmetrically encrypted by `sk`, then `sk` is asymmetrically encrypted using the recipient's public key.



For **decryption**, first the `sk` key is asymmetrically decrypted using the recipient's private key, then the ciphertext is decrypted symmetrically using `sk`.



The above process is known as **`Key encapsulation mechanism (KEM)`**: encapsulate an asymmetrically-encrypted random (ephemeral) symmetric key and use symmetric algorithm for the data encryption.

References

- RFC 1962 document - https://datatracker.ietf.org/doc/html/rfc1962#section-2.1
- TCP/IP guide - http://www.tcpipguide.com/free/t_PPPCompressionControlProtocolCCPandCompressionAlgo.htm
- Huffman Codes and Entropy in data Structures - https://www.tutorialspoint.com/huffman-codes-and-entropy-in-data-structure
- Bellman Ford algorithm - in Routing Information Protocol (RIP) by Krianto Sulaiman, Oris ; Mahmud Siregar, Amir ; Nasution, Khairuddin ; Haramaini, Tasliyah, published in Journal of Physics: Conference Series, Volume 1007, Issue 1, article id. 012009 (2018) - https://iopscience.iop.org/article/10.1088/1742-6596/1007/1/012009/pdf
- Dynamic routing published by Scott Waters - https://slideplayer.com/slide/11930475/
- Practical Cryptography for Developers by Svetlin Nakov - https://cryptobook.nakov.com/
- Types of Encryption by Jay Thakkar - https://www.thesslstore.com/blog/types-of-encryption-encryption-algorithms-how-to-choose-the-right-one/

- AES Implementation in Python by Pablo T. Campos - https://medium.com/quick-code/aes-implementation-in-python-a82f582f51c2
- Geeks for Geeks pages for various algorithms - https://www.geeksforgeeks.org/