**BFS (Breadth First Search)**

* Uninformed/blind search technique i.e., the cost of finding or the existence of the desired node is unknown.
* Level-order searching technique.
* Optimal.
* Uses queue.
* Time complexity: O(V+E)
* Space complexity: O(V)

**DFS (Depth First Search)**

* Uninformed/blind search technique.
* Solution may or may not exist.
* Non-optimal.
* Uses stack.
* Time complexity: O(V+E)
* Space complexity: O(V)

**TOPOLOGICAL SORT**

* Can be implemented using both BFS (Kahn’s Algorithm) and DFS. The complexity in both cases stay the same i.e., O(V+E) for time and O(V) for space.
* **Steps (Kahn’s Algorithm):**
  + Maintain indegree array (no. of possible sources of each vertex).
  + Push vertices with zero indegree into the queue.
  + Pop from the front, print it, mark as visited, and check its adjacent vertices, reduce their indegree.
  + If indegree becomes zero push into the queue.
  + REPEAT!!!
* There may exist more than one topological sequence for a graph.
* Time complexity: O(V+E)
* Space complexity: O(V)

**CYCLE IN AN UNDIRECTED GRAPH (USING DFS)**

* Implement normal DFS along with maintaining a parent array.
* For each neighbour of a vertex, check if it is visited or not.
* If not visited, carry on.
* If visited, then check if it is the parent vertex.
* If not the parent vertex, a cycle has been detected.
* Time complexity: O(V+E)
* Space complexity: O(V)

**CYCLE DETECTION IN A DIRECTED GRAPH (USING DFS)**

**STRONGLY CONNECTED COMPONENTS**

* The sort the graph topologically.
* Transpose the graph.
* Apply DFS on the graph on the basis of the above generated topological sequence.
* When in one depth no visited vertices can be found, one strongly connected component is found.
* Time complexity: O(V+E)
* Space complexity: O(V)

**KRUSKAL’S ALGORITHM**

* Constructs MST.
* **STEPS:**
  + Sort edges by their weights.
  + Traverse in ascending order, if two edges do not belong to the same set, bring them in to the same set, using dS for disjoint sets.
  + If they belong to the same set, do not include into the same set.
  + Time complexity: O(ElogE)

**PRIM’S ALGORITHM**

* Constructs MST w.r.t. some source vertex.
* **STEPS:**
  + Maintain three arrays key, mst and parent.
  + Mark all keys with infinity, mst with false, and parent with -1.
  + Mark key for source as 0.
  + Pick the minimum element from the key array (source in case of start).
  + Mark as true in the mst.
  + Check for neighbour nodes of the key element.
  + Check their weights and update their respective key values if less than current value.
  + Update their parent values.
  + Again, pick least key element until all mst values become true.
  + Time complexity: O(ElogE)

**DIJKSTRA’S ALGORITHM**

* Does not work for negative edges.
* **STEPS:**
  + Maintain three arrays value, processed and parent.
  + Marks all values with infinity, processed with false and parent with -1.
  + Mark value for source at 0.
  + Pick the minimum array from the value array (source is case of start).
  + Mark it true in the processed array.
  + Check for neighbour nodes of the value picked element.
  + Find new distance from the source node to the neighbour node.
  + If the new distance is less than the current node, update the value array.
  + With the value array update the parent node.
  + Repeat the above steps (V-1) times.
  + Time complexity: O(ElogV)

**BELLMAN FORD ALGORITHM**

* Relax the edges (v-1) times and finally one more time to check if a negative edge cycle exists or not in any order of edges.
* If in some step during relaxation none of the edges is identified with a new shortest path, it becomes clear that the graph doesn’t have a negative edge cycle.
* Maintain two arrays parent and value, where parent values are initialized with -1 and value array is initialized with infinity for all but 0 for source.
* Time complexity: O(VE)

**FLOYD WARSHALL**

* All pair shortest path.
* Can detect negative edge cycle if shortest distance from a vertex to itself is negative.
* **STEPS:**
  + Create an adjacency matrix and initialize using directed weights.
  + For V steps, at every step compare the distance through one newly included vertex.
  + If the new comparison gives a lower distance, update the matrix.
* Time complexity: O(n^3)
* Space complexity: O(n^2)

**ACTIVITY SELECTION PROBLEM**

* Time complexity: O(nlogn)

**JOB SEQUENCING**

* Time complexity: O(n^2)

**HUFFMAN ENCODING**

* Time complexity: O(nlogn)

**FRACTIONAL KNAPSACK PROBLEM**

* Time complexity: O(nlogn)

**DP**

* DP efficiently solves problems that display overlapping subproblem and optimal substructure properties.
* Overlapping subproblem property can be visualised as, when a problem is constructed of subproblems each of which arise multiple times.
* Optimal substructure property can be visualised as, when an optimal solution to a problem can be constructed through optimal solutions of its subproblems.
* Memoization is to solving a problem by solving its subproblems and saving the results, so that any time they may come again, they don’t have to be solved again, rather just looked up. It solves and saves on demand from top to bottom.
* Tabulation is to solving a problem by solving all its subproblems iteratively from the base case scenario and saving the results to look up later. It solves all the subproblems from bottom to top.

**LONGEST COMMON SUBSEQUENCE**

* Time complexity: O(n^2)
* Space complexity: O(n^2)

**01 Knapsack**

* Time complexity: O(N\*W)
* Space complexity: O(N\*W)

**MCM**

* Time complexity: O(n^3)
* Space complexity: O(n^2)

**TRAVELLING SALESMAN**

* NP Hard
* Time complexity: O(n^n) with brute-force,

O(n^2 \* 2^n) with dp.