Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Remember: m = n(n-1)/2 in worst case

```
MAX-SUBARRAY-SUM(Array A)
    sum = sum’ = A[0]
    for i = 1 to A.length - 1
        sum’ = max(A[i], sum’ + A[i])
        sum = max(sum, sum’)
      return sum
```

```
LONGEST-COMMON-SUBSEQUENCE(String A, String B) // O(nm)
      return LCS(A.length-1, B.length-1)

LCS(Integer i, Integer j)
      if i = -1 OR j = -1 then return 0
      if mem[i][j] then return mem[i][j]
      if a[i] == b[j] return mem[i][j] = 1 + LCS(i-1, j-1)
      else return mem[i][j] = max(LCS(i, j-1), LCS(i-1, j))
```

```
LONGEST-COMMON-SUBSEQUENCE(String A, String B)
      N = A.length, M = B.length;
      Array dp[A.length][B.length]
      for i = 0 to N
            for j = 0 to M
                  if i = 0 OR j = 0 then dp[i][j] = 0;
                  else if A[i-1] = B[j-1] then dp[i][j] = 1 + dp[i-1][j-1]
                  else dp[i][j] = max(dp[i][j-1], dp[i-1][j])
      return dp[n][m]
```

```
// EZ, Check Slides.
   -  BST-INSERT
   -  BST-SEARCH
   -  BST-REMOVE
   -  BST-PREORDER
   -  BST-INORDER
   -  BST-POSTORDER
   -  BST-MAX
   -  BST-MIN
```

```
BST-SUCCESSOR(Node x)
      if x.right ≠ NIL
            return BST-MIN(x.right)
      y = x.p
      while y ≠ NIL AND x == y.right
            x = y
            y = y.p
      return y
```

```
BST-PREDECESSOR(Node x)
      if x.left ≠ NIL
            return BST-MAX(x.left)
      y = x.p
      while y ≠ NIL AND x == y.left
            x = y
            y = y.p
      return y;
```

```
BUBBLE-SORT(Array A) // Stable, In-Place
      for i = 1 to A.length-1
            for j = A.length downto i+1
                  if A[j] < A[j-1] then
                        exchange A[j] with A[j-1]
```

```
SELECTION-SORT(Array A) // Not Stable, In-Place
      for i = 1 to A.length-1
            m = i
            for j = i to n
                  if A[j] < A[m] then m = j
            exchange A[i] with A[m]
```

```
INSERTION-SORT(Array A) // Stable, In-Place
      for j = 2 to A.length
            k = A[j]
            i = j - 1
            while i > 0 and A[i] > key
                  A[i + 1] = A[i]
                  i = i - 1
            A[i+1] = k
```

```
MERGE-SORT(List A, Int l, Int r) // Not stable, Out-Of-Place
      if l = r then
            return A[l]
      m = floor((l+r)/2)
      return MERGE(MERGE-SORT(A, l, m), MERGE-SORT(A, m+1, r))

MERGE(List A, List B)
      List C = ∅
      While A ≠ ∅ AND B ≠ ∅
      if a.getFirst() < b.getFirst()
            C.Append(A.getFirst())
            A.removeFirst()
      else
            C.Append(B.getFirst())
            B.removeFirst()

      if a = ∅ then C.append(B)
      else if b = ∅ then C.append(A)
      return C
```

```
RADIX-SORT(Array A, Integer d)// Non-Comparison, Stable, In-place
      for i = 1 to d
            StableSortOnDigit(A, i)
```

```
BUCKET-SORT(Array A) // Out-of-place, stability depends
      n = A.length
      Array_of_Lists B[n] = ∅
      for i = 1 to n
            insert A[i] into list B[floor(n*A[i])]
      for i = 0 to n - 1
            INSERTION-SORT(B[i])
      concatenate all lists B[0]..B[n-1]
```

```
QUICK-SORT(Array A, Int l, Int r) // Not stable, can be in-place
      if l < r
            m = PARTITION(A, l, r)
            QUICK-SORT(A, l, m-1)
            QUICK-SORT(A, m+1, r)

PARTITION(Array A, Integer l, Integer r)
      x = A[r]
      i = p-1
      for j = p to r-1
      if A[j] <= x
            i = i+1
            exchange A[i] with A[j]
      exchange A[i+1] with A[r]
      return i+1
```

```
HEAP-SORT(Array A) // Not Stable, in-Place, A is 1-indexed
      // n is a pointer to the last element in heap
      int n = A.length - 1

      // build a max heap from A
      for i = n downto 0 do SWIM(i)

      // Repeatedly extract maximum from heap
      while n > 0
            exchange A[0] with A[n]
            n = n - 1
            SINK(0)

SWIM(Integer k) // Swims A[k] up to its correct position
      if k = 0 then exit
      if x[k] > x[k/2] then exchange x[k] with x[k/2]
      SWIM(k/2)

SINK(Integer k) // Sinks A[k] down to its correct position
      if 2k = n AND A[k] < A[2k] then
            exchange A[k] with A[2k]
      else if 2k < n then
            g = Index of max(A[2k], A[2k+1])
            exchange A[k] with A[g]
            SINK(g)
```

```
COUNTING-SORT(Array A) // Non-Comparison, Stable, Out-of-Place
      Array B[A.length] = Ø
      Array C[A.max_element()+1]
      for i = 0 to A.length
            C[A[i]] = C[A[i]] + 1
      for i = 1 to A.max_element()
            C[i] = C[i] + C[i-1]
      for i = 0 to n-1
            B[C[A[i]]-1] = A[i]
            C[A[i]] = C[A[i]] - 1
      return B
```

```
BFS(Graph G, vertex s) // O(|V| + |E|)
      foreach vertex u ∈ G.V
            u.distance = ∞
            u.color = WHITE

      s.color = GRAY
      d.distance = 0
      Queue q = Ø
      q.enqueue(s)
      while q ≠ Ø
            u = q.dequeue()
            foreach vertex v ∈ G.Adj[u]
                  if v.color = WHITE
                        v.color = GRAY
                        v.distance = u.distance + 1
                        v.parent = s
                        q.enqueue(v)
            u.color = BLACK
```

```
DFS(Graph G) // O(|V| + |E|)
      foreach vertex u ∈ G.V
            u.color = WHITE
      foreach vertex u ∈ G.V
            if u.color == WHITE
                  DFS-VISIT(G, u)

DFS-VISIT(vertex s)
      s.color = GRAY
      foreach vertex u ∈ G.Adj[s]
            if u.color = WHITE
                  u.parent = s
                  DFS-VISIT(u)
      s.color = BLACK
```

```
TOPOLOGICAL-SORT(Graph G) // O(|V| + |E|)
      Stack S = Ø
      List order = Ø
      Foreach vertex v ∈ G.V
            if visited[v] = false
                  DFS-VISIT(v)
      While S ≠ Ø
            order.add(S.pop())

      return order

DFS-VISIT(vertex s, Stack S)
      visited[s] = true
      foreach vertex u ∈ G.Adj[s]
            if visited[u] = false
                  DFS-VISIT(u)
      S.push(s)
```

```
PRIM-MST(Graph G, vertex s) // O( (E.log(V) )
      List<Edge> MST = Ø
      MIN-PQ<Edge> Q = Ø
      foreach vertex v ∈ G.Adj[s]
            Q.push(Edge(s, v))

      while Q ≠ Ø
            Edge e = Q.EXTRACT-MIN()
            if e not in mst
                  MST.add(e)
            vertex u = e.to()
            foreach vertex v ∈ G.Adj[u]
                  if v ∉ MST
                        Q.push(Edge(u, v))
      return MST
```

```
KRUSKAL-MST(Graph G) // O(E.log(E)) or O(E.log(V))
      List<Edge> MST = Ø
      List E = SORT(G.E) // Sort all edges by weight in ascending order
      foreach edge in E
            if e doesn’t create a cycle
                  MST.add(e)
      return MST
```

```
SINGLE-SOURCE-DIJKSTRA(Graph G, vertex s) // O(|E| log(V))
      foreach vertex u ∈ G.V
            u.distance = ∞
            u.marked = false
      s.distance = 0
      s.marked = true

      MIN-PQ<Edge> Q = Ø
      foreach vertex v ∈ G.Adj[s]
            v.distnace = weight(s, v)
            Q.push(Edge(s, v))

      while Q ≠ Ø
            vertex u = Q.EXTRACT-MIN().to
            u.marked = true
            foreach vertex v ∈ G.Adj[u]
                  v.distance = min(v.distance, u.distance + weight(u, v))
                  if v.marked = false
                        Q.push(Edge(u, v))
```

```
FORD-FULKERSON(Network N) // O(E.F*), Check MaxFlow.docx

      Begin with flow = 0.

      While there is an augmenting path in the residual network

            Augment the flow through that path.
```