Algorithm (Short Notes)

Selection Sort:

```
Procedure Selectsort(A[1..n])
begin
  for i=1 to n-1 do
    Min := i
    for j= i+1 to n do
        if A[j] < A[Min] then Min := j
    endfor
    swap (A[i], A[min])
endfor
end</pre>
```

Merge Sort:

```
\frac{\text{MERGE}(A[1..n], m):}{i \leftarrow 1; \ j \leftarrow m+1}
\text{for } k \leftarrow 1 \text{ to } n
\text{if } j > n
B[k] \leftarrow A[i]; \ i \leftarrow i+1
\text{else if } i > m
B[k] \leftarrow A[j]; \ j \leftarrow j+1
\text{else if } A[i] < A[j]
B[k] \leftarrow A[i]; \ i \leftarrow i+1
\text{else}
B[k] \leftarrow A[j]; \ j \leftarrow j+1
\text{for } k \leftarrow 1 \text{ to } n
A[k] \leftarrow B[k]
```

```
\frac{\text{MergeSort}(A[1..n]):}{\text{if } n > 1}
m \leftarrow \lfloor n/2 \rfloor
\text{MergeSort}(A[1..m])
\text{MergeSort}(A[m+1..n])
\text{Merge}(A[1..n], m)
```

Quick Sort:

```
\frac{\text{PARTITION}(A[1..n], p):}{\text{swap } A[p] \longleftrightarrow A[n]}
i \leftarrow 0
j \leftarrow n
\text{while } (i < j)
\text{repeat } i \leftarrow i + 1 \text{ until } (i \ge j \text{ or } A[i] \ge A[n])
\text{repeat } j \leftarrow j - 1 \text{ until } (i \ge j \text{ or } A[j] \le A[n])
\text{if } (i < j)
\text{swap } A[i] \longleftrightarrow A[j]
\text{swap } A[i] \longleftrightarrow A[n]
\text{return } i
```

```
QUICKSORT(A[1..n]):

if (n > 1)

Choose a pivot element A[p]

r \leftarrow \text{Partition}(A, p)

QUICKSORT(A[1..r-1])

QUICKSORT(A[r+1..n])
```

gradeup

Insertion Sort:

```
Insertion-Sort (A, n) \triangleright A[1 ... n]

for j \leftarrow 2 to n

do key \leftarrow A[j]

i \leftarrow j - 1

while i > 0 and A[i] > key

do A[i+1] \leftarrow A[i]

i \leftarrow i - 1

A[i+1] = key
```

Bubble Sort:

Heap Sort:

Heapify (A, i)

- 1. l ← left [i]
- 2. $r \leftarrow \text{right } [i]$
- 3. if $l \le \text{heap-size } [A] \text{ and } A[l] > A[i]$
- 4. then largest $\leftarrow l$
- 5. else largest $\leftarrow i$
- 6. if $r \le \text{heap-size } [A] \text{ and } A[i] > A[\text{largest}]$
- 7. then largest $\leftarrow r$
- 8. if largest ≠ i
- 9. then exchange $A[i] \leftrightarrow A[largest]$
- 10. Heapify (A, largest)

Heapsort(A)

- 1. BUILD_HEAP (A)
- 2. for $i \leftarrow \text{length (A) down to 2 do}$
 - 1. exchange $A[1] \leftrightarrow A[i]$
 - 2. heap-size $[A] \leftarrow$ heap-size [A] 1
 - 3. Heapify (A, 1)

Binary tree traversal Algorithms:

```
PreOrder(\nu):
                              InOrder(v):
                                                             PostOrder(\nu):
  if v = Null
                                if v = Null
                                                               if v = Null
      return
                                     return
                                                                   return
  else
                                else
                                                               else
      print label(v)
                                       INORDER(left(v))
                                                                  PostOrder(left(v))
      Preorder(left(v))
                                     print label(v)
                                                                  Post Order(right(v))
      PreOrder(right(v))
                                       InOrder(right(v))
                                                                   print label(v)
```

Linear Search

Binary Search

```
int binarysearch (int a[], int n, int key)
           int first = 0, last = n - 1, middle;
                   while (first \leq = last)
                   middle = (first + last)/2; /*
                   calculate middle* /
                   if (a [middle] == value) /*
                   if value is found at mid */
                                  return middle;
                                  else if (a [middle] > value) /*
                                  if value is at left half */
                                  last = middle - 1;
                   }
                           first = middle + 1; /*
                           if value is in right half */
                           Return -1;
    }
```

Recursiv binarh search Algorithm:

```
int binarySearch(int a[], int start, int end, int key)
{
   if(start <= end)
   {
     int mid = (start + end)/2;
     if(a[mid] == key) return mid;
     if(a[mid] < key)
        return binarySearch(a, mid+1, end, key);
     else
        return binarySearch(a, start, mid-1, key);
   }
   return -1;
}</pre>
```

Asymptotic Notations:

Big Oh (O): If we write f(n) = O(g(n)), then there exists a function f(n) such that $\forall n \ge n0$, $f(n) \le cg(n)$ with any constant c and a positive integer n0.

Big Omega (Ω): If we write $f(n) = \Omega(g(n))$, then there exists a function f(n) such that $\forall n \ge n0$, $f(n) \ge cg(n)$ with any constant c and a positive integer no.

Big Theta (θ): If we write $f(n) = \theta(g(n))$, then there exists a function f(n) such that $\forall n \ge n0$, $c1g(n) \le f(n) \le c2g(n)$ with a positive integer n0, any positive constants c1 and c2.

Small Oh (o): If we write f(n) = o(g(n)), then there exists a function such that f(n) < c g(n) with any positive constant c and a positive integer n0.

Small Omega (\omega): If we write $f(n) = \omega(g(n))$, then these exists a function such that f(n) > cg(n) with any positive constant c and a positive integer n0.

Important Asymptotic relations:

```
f(n) \in O(g(n)) \text{ if and only if } g(n) \in \Omega(f(n))
f(n) \in o(g(n)) \text{ if and only if } g(n) \in \omega(f(n))
f(n) \in \Theta(g(n)) \text{ if and only if } f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))
f(n) \in o(g(n)) \text{ implies } f(n) \in O(g(n))
f(n) \in \omega(g(n)) \text{ implies } f(n) \in \Omega(g(n))
f(n) \in O(g(n)) \text{ implies } f(n) \notin \omega(g(n))
f(n) \in O(g(n)) \text{ implies } f(n) \notin o(g(n))
f(n) \sim g(n) \text{ implies } f(n) \notin o(g(n))
f(n) \sim g(n) \text{ implies } f(n) \in \Theta(g(n))
f(n) \sim g(n) \text{ is equivalent to } f(n) \in g(n) + o(g(n))
If f(n) \in \Theta(g(n)) and g(n) \in O(h(n)), then f(n) \in O(h(n))
If f(n) \in \Theta(g(n)) and g(n) \in O(h(n)), then f(n) \in O(h(n))
If f(n) \in O(g(n)) and g(n) \in O(h(n)), then f(n) \in O(h(n))
If f(n) \in O(g(n)) and g(n) \in O(h(n)), then f(n) \in O(h(n))
If f(n) \in O(g(n)) and g(n) \in O(h(n)), then f(n) \in O(h(n))
```

Worst case Time Complexities for popular data structures:

Data Structure	Worst Case Time Complexity			
	Access	Search	Insertions	Delete
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Singly Linked List	O(n)	O(n)	Begin: O(1),	Begin: O(1),
			End: O(n)	End: O(n)
Doubly Linked List	O(n)	O(n)	Begin: O(1),	Begin: O(1),
			End: O(n)	End: O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Time Complexities for popular sorting algorithms:

Sorting Algorithms	Best Case	Average Case	Worst Case
Quick Sort	$O(n \log(n))$	O(n log(n))	O(n2)
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n2)	O(n2)	O(n2)
Selection Sort	O(n2)	O(n2)	O(n2)
Insertion Sort	O(n)	O(n2)	O(n2)
Heap Sort	O(n log(n))	$O(n \log(n))$	O(n log(n))

Kruskal's algorithm:

- Make-Set(v) puts v in a set by itself
- \bullet FIND-Set(v) returns the name of v's set
- ullet Union(u,v) combines the sets that u and v are in

$\mathbf{MST\text{-}Kruskal}(G,w)$

```
\begin{array}{lll} 1 & A \leftarrow \emptyset \\ \mathbf{2} & \text{for each vertex } v \in V[G] \\ \mathbf{3} & \text{do Make-Set}(v) \\ \mathbf{4} & \text{sort the edges of } E \text{ into nondecreasing order by weight } w \\ \mathbf{5} & \text{for each edge } (u,v) \in E, \text{ taken in nondecreasing order by weight} \\ \mathbf{6} & \text{do if } \operatorname{Find-Set}(u) \neq \operatorname{Find-Set}(v) \\ \mathbf{7} & \text{then } A \leftarrow A \cup \{(u,v)\} \\ \mathbf{8} & \text{Union}(u,v) \\ \mathbf{9} & \text{return } A \end{array}
```

Prim's algorithm:

- INSERT(v) puts v in the structure
- Extract-Min() finds and returns the node with minimum key value
- \bullet Decrease-Key(v, w) updates (decreases) the key of v

```
MST-Prim(G, w, r)
 1
       for each u \in V[G]
 \mathbf{2}
                do key[u] \leftarrow \infty
                     \pi[u] \leftarrow \text{NIL}
 3
 4
      key[r] \leftarrow 0
       Q \leftarrow V[G]
 \mathbf{5}
 6
       while Q \neq \emptyset
 7
                \mathbf{do}\ u \leftarrow \text{Extract-Min}(Q)
 8
                     for each v \in Adj[u]
 \mathbf{9}
                             do if v \in Q and w(u,v) < key[v]
                                      then \pi[v] \leftarrow u
10
                                               key[v] \leftarrow w(u,v)
11
```

Dijkstra's Algorithm:

```
let T be a single vertex s;
while (T has fewer than n vertices)
{
    find edge (x,y)
        with x in T and y not in T
        minimizing d(s,x)+length(x,y)
    add (x,y) to T;
    d(s,y)=d(s,x)+length(x,y);
}
```

Dijkstra's Algorithm Implementation using Heap:

```
Make a heap of values (vertex,edge,distance);
Initially (v,-,infinity) for each vertex;
Let tree T be empty;
while (T has fewer than n vertices)
{
  let (v,e,d(v)) have the smallest weight in the heap;
  remove (v,e,d(v)) from the heap;
  add v and e to T;
  set distance(s,v) to d(v);
  for each edge f=(v,u)
      if u is not already in T
            find value (u,g,d(u)) in heap
      if d(v)+length(f) < d(g)
            replace (u,g,d(g)) with (u,f,d(v)+length(f))
}
```

Bellman Ford Algorithm:

```
\begin{aligned} &d(v[1]) \leftarrow 0 \\ &\text{For } (j=2 \text{ to } n) \\ &d(v[j]) \leftarrow \infty \end{aligned} \begin{aligned} &\text{For } (i=1 \text{ to } (|V|\text{-}1)) \\ &\text{For each } (u,v) \text{ in } E \\ &d(v) \leftarrow \text{Min } \{d(v), d(u) + l(u,v)\} \end{aligned} \begin{aligned} &\text{For each } (u,v) \text{ in } E \\ &\text{if } d(v) > d(u) + l(u,v) \\ &\text{Print}(\text{"Negative Cycle"}) \end{aligned}
```

Floyd-warshall algorithm:

}

```
Floyd-Warshall (w, n)
{
       for (i=1 \text{ to } n) {
              for (j=1 \text{ to } n) {
                     d[i, j] = w[i,j];
                     Pred[i, j]=NIL;
              }
       }
       for (k=1 \text{ to } n) {
              for (i=1 \text{ to } n) {
                     for (j=1 \text{ to } n) {
                             if((d[i, k] + d[k, j]) < d[i, j]) 
                                    d[i, j] = d[i, k] + d[k, j];
                                    Pred[i, j] = k;
                             }
                      }
       } //d is updated with all pairs shortest path
```