```
Algorithm PQ-Sort(S, C)
Input sequence S, comparator C
for the elements of S
Output sequence S sorted in
P ← priority queue with
comparator C
while ¬S.isEmpty ()
     e ← S.removeFirst ()
     P.insert (e, 0)
while ¬P.isEmpty()
     e ← P.removeMin().key()
     S.insertLast(e)
```

```
Algorithm SelectionSort(A);
Input sequence A
Output sorted sequence A
  for i = 1 to n - 1
    posmin ← i
    for j = (i + 1) to n
      if a[j] < a[posmin]
        posmin ← j
    if posmin != i
        tmp ← a[i]
        a[i] ← a[posmin]
        a[posmin] ← tmp
```

```
Algorithm InsertionSort(A)
Input sequence A
Output sorted sequence A
     for i ← 1 to length[A]
       do value ← A[i]
            j ← i-1
        while j >= 0 and A[j] > value
          do A[j + 1] ← A[j]
             j ← j-1
        A[j+1] ← value
```

```
Algorithm mergeSort(S, C)
Input sequence S with n
elements, comparator C
Output sequence S sorted
according to C
if S.size() > 1
     (S1, S2) ← partition(S, n/2)
     mergeSort(S1, C)
     mergeSort(S2, C)
     S ← merge(S1, S2)
```

```
Algorithm merge(A, B)
Input sequences A and B with
n/2 elements each
Output sorted sequence of A ∪ B
S ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
     if A.first().element() < B.first().element()
          S.insertLast(A.remove(A.first()))
     else
          S.insertLast(B.remove(B.first()))
while ¬A.isEmpty()
     S.insertLast(A.remove(A.first()))
while ¬B.isEmpty()
     S.insertLast(B.remove(B.first()))
return S
```

```
Algorithm inPlaceQuickSort(S, l, r)
Input sequence S, ranks l and r
Output sequence S with the elements of rank
between l and r rearranged in increasing order.
if l ≥ r
     return
i ← a random integer between l and r
x ← S.elemAtRank(i)
(h, k) ← inPlacePartition(x)
inPlaceQuickSort(S, l, h - 1)
inPlaceQuickSort(S, k + 1, r)
```

```
Algorithm quickSelect(A, l, r, k)
Input sequence A, left index l, right index r,
Output
if l = r
     return A[l]
Select pivotIndex between l and r
pivotIndex ← partition(A, l, r, pivotIndex)
if(k = pivotIndex)
     return A[k]
else if(k < pivotIndex)
     quickSelect(A, l, pivotIndex-1, k)
else
     quickSelect(A, pivotIndex +1, r, k)
```

```
Algorithm heapsort(A, n)
Input sequence A and size n
Output sorted A
for i ← n/2 -1 to 0
     heapify(A,n,i)
for i ← n -1 to 0
     swap(A, i, 0)
     heapify(A, i, 0)
```

```
Algorithm DFS(G, v)
Input grafo G vertice iniziale v di G (con grafo
inizialmente settato ad UNEXPLORED)
Output etichettatura degli spigoli di G
nella componente connessa di v
come tree-edge e back-edge
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
     if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
               setLabel(e, DISCOVERY)
               DFS(G, w)
     else
          setLabel(e, BACK)
```

```
Algorithm pathDFS(G, v, z)
Input grafo G vertice iniziale v di G e vertice di
arrivo z (con grafo inizialmente settato ad
UNEXPLORED)
Output percorso tra v e z
setLabel(v, VISITED)
S.push(v)
if(v = z)
     return S.elements()
for all e ∈ G.incidentEdges(v)
     if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
               setLabel(e, DISCOVERY)
               S.push(e)
               pathDFS(G, w, z)
               S.pop(e)
     else
          setLabel(e, BACK)
S.pop(e)
```

```
Algorithm cycleDFS(G, v)
setLabel(v, VISITED)
S.push(v)
for all e ∈ G.incidentEdges(v)
     if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          S.push(e)
          if getLabel(w) = UNEXPLORED
               setLabel(e, DISCOVERY)
               cycleDFS(G, w)
               S.pop(e)
          else
               T ← new empty stack
               repeat
                    o ← S.pop()
                    T.push(o)
               until o = w
               return T.elements()
S.pop(v)
```

```
D-DFS(Digraph G, Vertex v)
Input graph G and source vertex V
Output labeled graph (of v)
(N.B. getDiscoveryLabel = getDiscLabel)
setDiscoveryLabel(v, getNextDLabel());
for all e in outgoingEdges(v)
    if(getLabel(e) == UNEXPLORED)
    w = opposite(e, v)
        if(getDiscoveryLabel(w)==UNEXPLORED)
            setLabel(e, DISCOVERY)
            D-DFS(G, w)
        else if(getLeavingLabel(w) == 0)
            setLabel(e, BACK)
        else if (getDiscLabel(v)<getDiscLabel(w))
            setLabel(e, FORWARD)
        else
            setLabel(e, CROSS)
    setLeavingLabel(v, getNextLLabel())
```

```
Algorithm FloydWarshall(G)
Input digrafo G
Output la chiusura transitiva G* di G
i ← 1
for all v ∈ G.vertices()
    denota v come v i
    i ← i + 1
G_0 ← G
for k ← 1 to n do
    G_k← G_k-1
    for i ← 1 to n (i ≠ k) do
        for j ← 1 to n (j ≠ i, k) do
            if G_k-1 .areAdjacent(v_i, v_k) ∧
              G_k-1.areAdjacent(v_k, v_k)
              if ¬ G_k.areAdjacent(v_i, v_j)
                 G_k.insertDirectedEdge(v_i,v_j,k)
    return G_n
```

```
Algorithm topologicalDFS(G, v)
Input grafo G e vertice iniziale v di G
(supponendo archi e nodi inizializzati a
UNEXPLORED)
Output etichettatura dei vertici di G
nella componente connessa di v
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            topologicalDFS(G, w)
        else
            { e è uno spigolo forward o cross }
    etichetta v con il numero n
    n ← n – 1 // side effect!
```

```
Algorithm BFS(G, s)
Input graph G, vortex s
Output grafo delle etichettature partendo dal
vertice s
L_0 ← new empty sequence
L_0.insertLast(s)
setLabel(s, VISITED )
i ← 0
while ¬L_I.isEmpty()
    L_I+1 ← new empty sequence
    for all v ∈ L_I.elements()
        for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
                w ← opposite(v,e)
                if getLabel(w) = UNEXPLORED
                    setLabel(e, DISCOVERY )
                    setLabel(w, VISITED )
                    L_I+1.insertLast(w)
                else
                    setLabel(e, CROSS )
    i ← i +1
```

```
Method TopologicalSort(G)
H ← G
// copia temporanea di G
n ← G.numVertices()
while H è non vuoto do
    trova un pozzo v // esiste sempre?
    etichetta v ← n
    n ← n - 1
    rimuovi v da H // anche gli spigoli incidenti
```

```
Algorithm DijkstraDistances(G, s)
Q ← new heap-based priority queue
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
        { relax edge e }
        z ← G.opposite(u,e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z,r)
            Q.replaceKey(getLocator(z),r)
```

```
Algorithm DijkstraShortestPathsTree(G, s)
Q ← new heap-based priority queue
for all v ∈ G.vertices()
    setParent(v, 0)
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
        { relax edge e }
        z ← G.opposite(u,e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z,r)
            setParent(z,e)
            Q.replaceKey(getLocator(z),r)
```

```
Algorithm PrimJarnikMST(G)
Q ← new heap-based priority queue
s ← a vertex of G
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
        z ← G.opposite(u,e)
        r ← weight(e)
        if r < getDistance(z)
            setDistance(z,r)
            setParent(z,e)
            Q.replaceKey(getLocator(z),r)
```

```
Algorithm Kruskal(G)
A = 0
foreach v ∈ G.vertices
    make-set(v)
foreach (u, v) ordered by weight of (u,v)
    if FIND-SET(u) ≠ FIND-SET(v):
        A = A U{(u,v)}
        UNION(u,v)
return A
```

**Algorithm BoruvkaMST(G)**

```
T ← V {subgraph containing just the vertices of G,
no edges}
while T has fewer than n-1 edges do
    for each connected component C in T do
        Let edge e be the smallest-weight edge
        from C to another component in T.
        if e is not already in T then
            Add edge e to T
return T
```

**Algorithm countingSort(A,n)**

```
Input sequenza A con size n
Output sequenza A ordinata senza confronto
max ← 0
min ← 0
for i ← 1 to n
    if(a[i] > a[max])
        max ← i
    else if(a[i] < min) min ← i
B ← empty sequence
for i ← 1 to n
    B[A[i] – min]++
h ← 0
for i ← 0 to max -min
    while(B[i]--)
        A[h++] ← i + min
```

**BUCKET-SORT(array A, int n)**

```
for(i = 1; i <= n; i++)
    insert A[i] nella lista B[floor(n*A[i])]
for(i = 0; i <= n-1; i++)
    sort B[i]
concatenate B[0] ... B[n-1]
```

**RADIX-SORT(array A, int d)**

```
for i←1 to d
   do use a stable sort to sort array A on digit i
// es. Java semplificato
static void rs(int[] a, int start, int endp1, int
mask) {
    if((endp1 - start <= 1) || (mask == 0))
        return;
    int j = start, k = endp1;
    while(j < k) {
        while((j < k) && ((a[j] & mask) == 0x0))
            j++;
        while((j < k) && ((a[k-1] & mask)!=0x0))
            k--;
        if(j < k) swap(a, j, k-1);
    }
    rs(a, start, j, mask >>> 1);
    rs(a, j, endp1, mask >>> 1);
}
```