RICERCA BINARIA

```java
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                   // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                  // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid - 1);   // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);  // recur right of the middle
  }
}
```

INVERTIRE ARRAY

```java
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                                 // if at least two elements in subarray
    int temp = data[low];                           // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high - 1);          // recur on the rest
  }
}
```

```
Algorithm IterativeReverseArray(A, i, j ):
    while i < j do
        Swap A[i ] and A[ j ]
        i = i + 1
        j = j - 1
    return
```

FIBONACCI

```java
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
  if (n <= 1)
    return n;
  else
    return fibonacciBad(n-2) + fibonacciBad(n-1);
}
```

```java
/** Restituisce un arrai contenente la coppia F(n) e F(n-1). */
public static long[] fibonacciGood(int n) {
  if (n <= 1) {
    long[] answer = {n, 0};
    return answer;
  } else {
    long[] temp = fibonacciGood(n - 1);             // restituisce {F(n-1), F(n-2)}
    long[] answer = {temp[0] + temp[1], temp[0]};   // vogliamo {F(n), F(n-1)}
    return answer;
  }
}
```

CODA DI PRIORITÀ con LISTE NON ORDINATE

```java
/** An implementation of a priority queue with an unsorted list. */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public UnsortedPriorityQueue( ) { super( ); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Returns the Position of an entry having minimal key. */
  private Position<Entry<K,V>> findMin( ) {   // only called when nonempty
    Position<Entry<K,V>> small = list.first( );
    for (Position<Entry<K,V>> walk : list.positions( ))
      if (compare(walk.getElement( ), small.getElement( )) < 0)
        small = walk;          // found an even smaller key
    return small;
  }
  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);     // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    list.addLast(newest);
    return newest;
  }

  /** Returns (but does not remove) an entry with minimal key. */
  public Entry<K,V> min( ) {
    if (list.isEmpty( )) return null;
    return findMin( ).getElement( );
  }

  /** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin( ) {
    if (list.isEmpty( )) return null;
    return list.remove(findMin( ));
  }

  /** Returns the number of items in the priority queue. */
  public int size( ) { return list.size( ); }
}
```

CODA DI PRIORITÀ con LISTE ORDINATE

```java
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public SortedPriorityQueue( ) { super( ); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public SortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);     // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    Position<Entry<K,V>> walk = list.last( );
    // walk backward, looking for smaller key
    while (walk != null && compare(newest, walk.getElement( )) < 0)
      walk = list.before(walk);
    if (walk == null)
      list.addFirst(newest);                    // new key is smallest
    else
      list.addAfter(walk, newest);              // newest goes after walk
    return newest;
  }
```

```
26    /** Returns (but does not remove) an entry with minimal key. */
27    public Entry<K,V> min( ) {
28      if (list.isEmpty( )) return null;
29      return list.first( ).getElement( );
30    }
31
32    /** Removes and returns an entry with minimal key. */
33    public Entry<K,V> removeMin( ) {
34      if (list.isEmpty( )) return null;
35      return list.remove(list.first( ));
36    }
37
38    /** Returns the number of items in the priority queue. */
39    public int size( ) { return list.size( ); }
```

HEAP IMPLEMENTAZIONE CON ARRAY

```
1   /** An implementation of a priority queue using an array-based heap. */
2   public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3     /** primary collection of priority queue entries */
4     protected ArrayList<Entry<K,V>> heap = new ArrayList<>( );
5     /** Creates an empty priority queue based on the natural ordering of its keys. */
6     public HeapPriorityQueue( ) { super( ); }
7     /** Creates an empty priority queue using the given comparator to order keys. */
8     public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
9     // protected utilities
10    protected int parent(int j) { return (j−1) / 2; }          // truncating division
11    protected int left(int j) { return 2*j + 1; }
12    protected int right(int j) { return 2*j + 2; }
13    protected boolean hasLeft(int j) { return left(j) < heap.size( ); }
14    protected boolean hasRight(int j) { return right(j) < heap.size( ); }
15    /** Exchanges the entries at indices i and j of the array list. */
16    protected void swap(int i, int j) {
17      Entry<K,V> temp = heap.get(i);
18      heap.set(i, heap.get(j));
19      heap.set(j, temp);
20    }
21    /** Moves the entry at index j higher, if necessary, to restore the heap property. */
22    protected void upheap(int j) {
23      while (j > 0) {                 // continue until reaching root (or break statement)
24        int p = parent(j);
25        if (compare(heap.get(j), heap.get(p)) >= 0) break;    // heap property verified
26        swap(j, p);
27        j = p;                                    // continue from the parent's location
28      }
29    }
30    /** Moves the entry at index j lower, if necessary, to restore the heap property. */
31    protected void downheap(int j) {
32      while (hasLeft(j)) {                      // continue to bottom (or break statement)
33        int leftIndex = left(j);
34        int smallChildIndex = leftIndex;              // although right may be smaller
35        if (hasRight(j)) {
36          int rightIndex = right(j);
37          if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
38            smallChildIndex = rightIndex;             // right child is smaller
39        }
40        if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
41          break;                                      // heap property has been restored
42        swap(j, smallChildIndex);
43        j = smallChildIndex;                          // continue at position of the child
44      }
45    }
46
47    // public methods
48    /** Returns the number of items in the priority queue. */
49    public int size( ) { return heap.size( ); }
50    /** Returns (but does not remove) an entry with minimal key (if any). */
51    public Entry<K,V> min( ) {
52      if (heap.isEmpty( )) return null;
53      return heap.get(0);
54    }
```

```
55    /** Inserts a key-value pair and returns the entry created. */
56    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
57      checkKey(key);       // auxiliary key-checking method (could throw exception)
58      Entry<K,V> newest = new PQEntry<>(key, value);
59      heap.add(newest);                          // add to the end of the list
60      upheap(heap.size( ) − 1);                  // upheap newly added entry
61      return newest;
62    }
63    /** Removes and returns an entry with minimal key (if any). */
64    public Entry<K,V> removeMin( ) {
65      if (heap.isEmpty( )) return null;
66      Entry<K,V> answer = heap.get(0);
67      swap(0, heap.size( ) − 1);                 // put minimum item at the end
68      heap.remove(heap.size( ) − 1);             // and remove it from the list;
69      downheap(0);                               // then fix new root
70      return answer;
71    }
72  }
```

HEAPSORT

```
HeapSort(S) //S: lista/array da ordinare
// Usiamo un Heap che chiameremo heap
Output: <array a ordinato rispetto alle chiavi degli elementi in S>
while (!S.isEmpty):
    heap.insert(S.remove())
    // Assumiamo che gli elementi di S siano coppie (k, v)
    // S.remove() rimuove un elemento da S secondo un criterio qualsiasi
while (!heap.isEmpty):
    a.add(heap.removeMin()) //add aggiunge alla fine dell'array
return a
```

HEAPIFY

```
protected void heapify() {
  int startIndex = parent(size()-1);    // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--)   // loop until processing the root
    downheap(j);
}
```

REPLACE COPPIA

```
replace(e, k) {
      i = e.position;
      e.key = k;
      if (heap.get(i).key < heap.parent(i).key)
            upheap(i);
      else
            downheap(i); // Potrebbe non essere
                         // necessario
}
```

Sorting di un Insieme usando una Coda di Priorità

```
// Usiamo una coda di priorità coda
Output: <array a ordinato rispetto alle chiavi degli elementi in S>
while (!S.isEmpty):
    coda.insert(S.remove())
    // Assumiamo che gli elementi di S siano coppie (k, v)
    // S.remove() rimuove un elemento da S secondo un criterio qualsiasi
while (!coda.isEmpty):
    a.add(coda.removeMin()) //add aggiunge alla fine dell'array
return a
```

MERGESORT ITERATIVO

```
// Credits: Prof. Shun Yan Cheung – Emory College
   public static void sort(double[] a)
   {
      int width;
      for ( width = 1; width < a.length; width = 2*width )
      {
         // Combine pairs of array a of width "width"
         int i;
         for ( i = 0; i < a.length; i = i + 2*width )
         {
            int left, middle, right;
            left = i;
            middle = i + width;
            right  = i + 2*width;
            merge( a, left, middle, right );
             // Merge è quello solito
         }
      }
   }
```

BUCKET SORT

```
Algorithm bucketSort(S):
    Input: Sequence S of entries with integer keys in the range [0, N - 1]
    Output: Sequence S sorted in nondecreasing order of the keys
    B = <array di N sequenze> // Inizialmente vuoto
    // Fase 1
    while (!S.isEmpty()) {
        e = S.removeFirst()
        k = e.key()
        B[k].addLast(e)
    }
    // Fase 2
    for (k = 0; k < N; k++)
        while (!B[k].isEmpty())
            S.addLast(B[k].removeFirst())
```

RADIX SORT

Algorithm *radixSort(S, N)*
Input sequenza S di d-tuple tali che $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ e $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$ per ogni tupla $(x_1, \ldots, x_d)$ in S
Output sequenza S in ordine lessicografico
for $i \leftarrow d$ **downto** 1
  bucketSort(S, N, i) // i-esima cifra

Algorithm *binaryRadixSort(S)*
Input sequenza S di interi a b-bit
Output sequence S sorted
for $i \leftarrow 0$ **to** $b - 1$
  bucketSort(S, 2, i)

MAPPE

```
Algorithm get(k):
   B = S.positions() {B è iteratore sulle posizioni di
   S}
   while B.hasNext() do
      p = B.next()     {posizione successiva in  B }
      if p.element().getKey() = k     then
         return p.element().getValue()
   return null {non esiste un elemento con chiave
    k}
```

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().getKey() = k  then
      t = p.element().getValue()
      S.set(p,(k,v))
      return t {ritorna il valore precedente}
S.addLast((k,v))
n = n + 1   {incrementa la variabile che
   memorizza il numero di elementi}
return null    {non ci sono elementi con chiave
   uguale a  k }
```

INSIEMI

```
Algoritmo genericMerge(A, B)
    S ← sequenza vuota
while ¬A.isEmpty() ∧ ¬B.isEmpty()
    a ← A.first().element(); b ←
        B.first().element()
        if a < b
    aIsLess(a, S); A.remove(A.first())
        else if b < a
    bIsLess(b, S); B.remove(B.first())
        else { b = a }
        bothAreEqual(a, b, S)
A.remove(A.first()); B.remove(B.first())
 while ¬A.isEmpty()
 aIsLess(a, S); A.remove(A.first())
    while ¬B.isEmpty()
 bIsLess(b, S); B.remove(B.first())
    return S
```

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().getKey() = k  then
      t = p.element().getValue()
      S.remove(p)
      n = n – 1    {decrementa il numero di
   elementi}
      return t {restituisce il valore rimosso}
return null   {non ci sono elementi con chiave
   uguale a k }
```

QUICKSORT

```
Algorithm partition(S, p)
    Input sequenza S, posizione p del pivot
    Output sottosequenze L, E, G di
    elementi di S minori uguali o maggiori
    del pivot
    L, E, G ← sequenze vuote
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

```
Algorithm inPlaceQuickSort(S, l, r)
    Input sequenza S, ranghi l and r
    Output sequenza S con gli
        elementi di rango  fra l e r
        riordinati in ordine crescente
    if l ≥ r
        return
    i ← un numero casuale fra l e r
    x ← S.elemAtRank(i)
    (h, k) ← inPlacePartition(x)
    inPlaceQuickSort(S, l, h − 1)
    inPlaceQuickSort(S, k + 1, r)
```

```
1   /** Quick-sort contents of a queue. */
2   public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
3     int n = S.size( );
4     if (n < 2) return;                          // queue is trivially sorted
5     // divide
6     K pivot = S.first( );                       // using first as arbitrary pivot
7     Queue<K> L = new LinkedQueue<>( );
8     Queue<K> E = new LinkedQueue<>( );
9     Queue<K> G = new LinkedQueue<>( );
10    while (!S.isEmpty( )) {                     // divide original into L, E, and G
11      K element = S.dequeue( );
12      int c = comp.compare(element, pivot);
13      if (c < 0)                                // element is less than pivot
14        L.enqueue(element);
15      else if (c == 0)                          // element is equal to pivot
16        E.enqueue(element);
17      else                                      // element is greater than pivot
18        G.enqueue(element);
19    }
20    // conquer
21    quickSort(L, comp);                         // sort elements less than pivot
22    quickSort(G, comp);                         // sort elements greater than pivot
23    // concatenate results
24    while (!L.isEmpty( ))
25      S.enqueue(L.dequeue( ));
26    while (!E.isEmpty( ))
27      S.enqueue(E.dequeue( ));
28    while (!G.isEmpty( ))
29      S.enqueue(G.dequeue( ));
30  }
```

```
1   /** Sort the subarray S[a..b] inclusive. */
2   private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
3                                                    int a, int b) {
4     if (a >= b) return;        // subarray is trivially sorted
5     int left = a;
6     int right = b-1;
7     K pivot = S[b];
8     K temp;                    // temp object used for swapping
9     while (left <= right) {
10      // scan until reaching value equal or larger than pivot (or right marker)
11      while (left <= right && comp.compare(S[left], pivot) < 0) left++;
12      // scan until reaching value equal or smaller than pivot (or left marker)
13      while (left <= right && comp.compare(S[right], pivot) > 0) right--;
14      if (left <= right) {       // indices did not strictly cross
15        // so swap values and shrink range
16        temp = S[left]; S[left] = S[right]; S[right] = temp;
17        left++; right--;
18      }
19    }
20    // put pivot into its final place (currently marked by left index)
21    temp = S[left]; S[left] = S[b]; S[b] = temp;
22    // make recursive calls
23    quickSortInPlace(S, comp, a, left - 1);
24    quickSortInPlace(S, comp, left + 1, b);
25  }
```

HASHING

Liste di Trabocco e Scansione Lineare

```
Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = null then          {k è una nuova chiave}
      n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ null then          {k trovata}
      n = n - 1
```

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.getKey () = k
            return c.getValue()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
        return null
```

MAPPE ORDINATE

```
Algorithm find(k, low, high) {
  if (high < low) return high + 1;        // no entry qualifies
  mid = (low + high) / 2;
  comp = compare(key, table.get(mid));
  if (comp == 0)
    return mid;                           // found exact match
  else if (comp < 0)
    return findIndex(key, low, mid - 1);  // answer is left of mid (or possibly mid)
  else
    return findIndex(key, mid + 1, high); // answer is right of mid
}

/** Version of findIndex that searches the entire table */
Algorithm find(k) { return find(key, 0, table.size() - 1); }
```

```
Algorithm higherEntry(k) {
  j = find(k); // Restituisce posizione ultimo elemento == k o primo > k
  while (j < size() - 1 && k == table[j].getKey())
    j++;
  return j;
}
```

```
algorithm merge(A, B) is
    inputs A, B : list
    returns list

    C := new empty list
    while A is not empty and B is not empty do
        if head(A) ≤ head(B) then
            append head(A) to C
            drop the head of A
        else
            append head(B) to C
            drop the head of B

    // By now, either A or B is empty. It remains to empty the other input list.
    while A is not empty do
        append head(A) to C
        drop the head of A
    while B is not empty do
        append head(B) to C
        drop the head of B

    return C
```

ALBERI BINARI DI RICERCA

```
Algorithm TreeSearch(k, v)
    if T.isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, right(v))
```

```
Algorithm subMap(v, k1, k2, buffer) {
    If (v == null)
        return;
    If (k1 > v.key)
        subMap(v.rightChild(), k1, k2, buffer);
    else {
        subMap(v.leftChild(), k1, k2, buffer)
        If (k2 >= v.key) {
            buffer.add(v.pair);
            subMap(v.rightChild(), k1, k2, buffer)
        }
    }
}
```

```
Algorithm inorder(v) {
    if (v == null)
        return;
    inorder(v.leftChild());
    visit(v);
    inorder(v.rightChild());
}
```

```
Algorithm postOrder(v) {
    if (v == null)
        return;
    postOrder(v.leftChild());
    postOrder(v.rightChild());
    visit(v);
}
```

```
Algorithm eulerTour(v) {
    If (v is a leaf) // O foglia o 2 figli
        print v.val;
        return
    print("(");
    eulerTour(v.leftChild());
    print(v.val); // Numero o operatore aritmetico
    eulerTour(v.rightChild());
    print(")");
}
```

algoritmo `search`$(chiave\ k) \rightarrow elem$

1. $v \leftarrow$ radice di T
2. **while** ($v \neq$ `null`) **do**
3. **if** ($k = chiave(v)$) **then return** $elem(v)$
4. **else if** ($k < chiave(v)$) **then** $v \leftarrow$ figlio sinistro di v
5. **else** $v \leftarrow$ figlio destro di v
6. **return** `null`

algoritmo `max`$(nodo\ u) \rightarrow nodo$

1. $v \leftarrow u$
2. **while** (figlio destro di $v \neq$ `null`) **do**
3. $v \leftarrow$ figlio destro di v
4. **return** v

algoritmo `pred`$(nodo\ u) \rightarrow nodo$

1. **if** (u ha figlio sinistro $sin(u)$) **then**
2. **return** `max`$(sin(u))$
3. **while** ($parent(u) \neq null$ e u è figlio sinistro di suo padre) **do**
4. $u \leftarrow parent(u)$
5. **return** $parent(u)$

UNION-FIND

Implementazione con Sequenze

```
1  /** A Union-Find structure for maintaining disjoint sets. */
2  public class Partition<E> {
3    //--------------- nested Locator class ------------
4    private class Locator<E> implements Position<E> {
5      public E element;
6      public int size;
7      public Locator<E> parent;
8      public Locator(E elem) {
9        element = elem;
10       size = 1;
11       parent = this;              // convention for a cluster leader
12     }
13     public E getElement( ) { return element; }
14   } //--------- end of nested Locator class ---------
15   /** Makes a new cluster containing element e and returns its position. */
16   public Position<E> makeCluster(E e) {
17     return new Locator<E>(e);
18   }
19   /**
20    * Finds the cluster containing the element identified by Position p
21    * and returns the Position of the cluster's leader.
22    */
23   public Position<E> find(Position<E> p) {
24     Locator<E> loc = validate(p);
25     if (loc.parent != loc)
26       loc.parent = (Locator<E>) find(loc.parent); // overwrite parent after recursion
27     return loc.parent;
28   }
29   /** Merges the clusters containing elements with positions p and q (if distinct). */
30   public void union(Position<E> p, Position<E> q) {
31     Locator<E> a = (Locator<E>) find(p);
32     Locator<E> b = (Locator<E>) find(q);
33     if (a != b)
34       if (a.size > b.size) {
35         b.parent = a;
36         a.size += b.size;
37       } else {
38         a.parent = b;
39         b.size += a.size;
40       }
41   }
42 }
```

```
Algorithm union(p, q) {
    int pid = id[p];
    for (int i = 0; i < id.length; i++)
        if (id[i] == pid) id[i] = id[q];
}
```

GRAFI

Algorithm DFS(G, u):

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

Mark vertex u as visited.
for each of u's outgoing edges, $e = (u,v)$ **do**
  if vertex v has not been visited **then**
    Record edge e as the discovery edge for vertex v.
    Recursively call DFS(G, v).

```
1  /** Performs depth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
3                               Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4    known.add(u);                              // u has been discovered
5    for (Edge<E> e : g.outgoingEdges(u)) {     // for every outgoing edge from u
6      Vertex<V> v = g.opposite(u, e);
7      if (!known.contains(v)) {
8        forest.put(v, e);                      // e is the tree edge that discovered v
9        DFS(g, v, known, forest);              // recursively explore from v
10     }
11   }
12 }
```

```
Algorithm pathDFS(G, v, 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(v)
```

```
Algorithm cycleDFS(G, v, z)
  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)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

```
1  /** Returns an ordered list of edges comprising the directed path from u to v. */
2  public static <V,E> PositionalList<Edge<E>>
3  constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
4                Map<Vertex<V>,Edge<E>> forest) {
5    PositionalList<Edge<E>> path = new LinkedPositionalList<>();
6    if (forest.get(v) != null) {          // v was discovered during the search
7      Vertex<V> walk = v;                 // we construct the path from back to front
8      while (walk != u) {
9        Edge<E> edge = forest.get(walk);
10       path.addFirst(edge);              // add edge to *front* of path
11       walk = g.opposite(walk, edge);    // repeat with opposite endpoint
12     }
13   }
14   return path;
15 }
```

```
Algorithm DFS(G)
  Input Grafo G
  Output etichetta gli archi di G come
  discovery e back
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input grafo G, un vertice iniziale v of G
  Output etichetta gli archi di G come
  discovery e back nella componente
  connessa del grafo a cui appartiene 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)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

```
1  /** Performs DFS for the entire graph and returns the DFS forest as a map. */
2  public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
3    Set<Vertex<V>> known = new HashSet<>();
4    Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
5    for (Vertex<V> u : g.vertices())
6      if (!known.contains(u))
7        DFS(g, u, known, forest);          // (re)start the DFS process at u
8    return forest;
9  }
```

```
Algorithm BFS(G)
  Input Grafo G
  Output etichetta gli archi e
  partiziona I vertici di G
 for all u ∈ G.vertices()
  setLabel(u, UNEXPLORED)
 for all e ∈ G.edges()
  setLabel(e, UNEXPLORED)
 for all v ∈ G.vertices()
  if getLabel(v) = UNEXPLORED
     BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← nuova sequenza vuota
  L0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← nuova sequenza vuota
    for all v ∈ Li.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)
            Li+1.addLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

```java
/** Performs breadth-first search of Graph g starting at Vertex u. */
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
                    Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
  known.add(s);
  level.addLast(s);                       // first level includes only s
  while (!level.isEmpty()) {
    PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
    for (Vertex<V> u : level)
      for (Edge<E> e : g.outgoingEdges(u)) {
        Vertex<V> v = g.opposite(u, e);
        if (!known.contains(v)) {
          known.add(v);
          forest.put(v, e);               // e is the tree edge that discovered v
          nextLevel.addLast(v);           // v will be further considered in next pass
        }
      }
    level = nextLevel;                    // relabel 'next' level to become the current
  }
}
```

ALGORITMO DI KOSARAJU

```
DDFS(v, Stack s) {
    if (v.state != UNEXPLORED)
        return;
    v.state = EXPLORING;
    for(cur in v.outEdges) {
        DDFS(cur, s);
    }
    v.state = EXPLORED;
    s.push(v);
}
```

```
SCC(G) {
    forest = new List() // Lista componenti connesse
    // Prima DFS
    stack = new Stack();
    for(v in G) {
        if(v.state == UNEXPLORED)
            DDFS(v, stack);
    }
    // Reset dello stato dei nodi
    for(v in G)
        v.state = UNEXPLORED;
    // Seconda DFS sul grafo trasposto
    for(v in stack) { // Nota: vertici estratti dalla pila
        if(v.state == UNEXPLORED) {
            ret = new List();
            transposedDFS(G, v, ret);
            forest.add(ret);
        }
    }
}
```

```
transposeDFS(G, n, lista) {
        // Effettua DFS sul grafo trasposto di G a partire da n
        // Inserisce i nodi trovati in lista
        // Nota: non è necessario trasporre effettivamente il grafo
}
```

Algoritmo di Floyd-Warshall

```
Algorithm FloydWarshall(G)
G0 = G
for k = 1 to n {
    Gk = Gk-1
    for (i, j con i ≠ k e j ≠ k )
        if ((i, j) non appartiene a Gk-1)
            if ((i, k) ∈ Gk-1 and (k, j) ∈ Gk-1)
                <Aggiungi (i, k) a Gk>
```

Ordinamento Topologico con DFS

```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
      setLabel(u, UNEXPLORED)
  for all v ∈ G.vertices()
      if getLabel(v) = UNEXPLORED
          topologicalDFS(G, v)
```

```
Algorithm topologicalDFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the vertices of G
        in the connected component of v
  setLabel(v, VISITED)
for all e ∈ G.outEdges(v)
    { outgoing edges }
    w ← opposite(v,e)
    if getLabel(w) = UNEXPLORED
        { e is a discovery edge }
        topologicalDFS(G, w)
    else
        { e is a forward or cross edge }
  Label v with topological number n
  n ← n - 1
```

```
Algorithm TopologicalSort(G)
  H ← G   // Copia temporanea di G
  n ← G.numVertices()
  while H is not empty do
   Let v be a vertex with no outgoing edges
   Label v ← n
   n ← n − 1
   Remove v from H
```

Algoritmo di Dijkstra

```
Algorithm ShortestPath(G,s):
  Input: A weighted graph G with nonnegative edge weights, and a distinguished
    vertex s of G.
  Output: The length of a shortest path from s to v for each vertex v of G.
  Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s.
  Let a priority queue Q contain all the vertices of G using the D labels as keys.
  while Q is not empty do
    {pull a new vertex u into the cloud}
    u = value returned by Q.remove_min()
    for each vertex v adjacent to u such that v is in Q do
      {perform the relaxation procedure on edge (u,v)}
      if D[u] + w(u,v) < D[v] then
        D[v] = D[u] + w(u,v)
        Change to D[v] the key of vertex v in Q.
  return the label D[v] of each vertex v
```

Heap flessibile (v. pros

Algoritmo di BELLMAN-FORD

```
Algorithm BellmanFord(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    for i ← 1 to n − 1 do
        for each e ∈ G.edges()
            { relax edge e }
            u ← G.origin(e)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
```

Mantenimento Albero dei Cammini Minimi

```
relax(u) {
    for every v adjacent to u
        if d[v] > d[u] + w(u, v) {
            d[v] = d[u] + w(u, v)
            pred[v] = u
        }
```

Algoritmo di PRIM-JARNIK

Algorithm PrimJarnik(G):
Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  (u, e) = value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

Algoritmo di KRUSKAL

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G

for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n - 1$ edges **do**
  (u, v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u, v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster
return tree T

Union find