ALGORITHM ***UniqueElements*** **(A[0 ... n − 1])**

//Determines whether all the elements in a given array are distinct.

//Input: An array $A[0..n-1]$

//Output: Returns "*true*" if all the elements in A are distinct.

// and "*false*" otherwise.

for $i \leftarrow$ **0 to** $n-$ **2 do**

  for $j \leftarrow i +$ **1 to** $n-$ **1 do**

    if $A[i] = A[j]$ **return false**

```python
def unique_element(arr):
    for i in range(0, len(arr)-2):
        for j in range(i+1, len(arr)):
            if arr[i] == arr[j]:
                return False

    return True
```

ALGORITHM ***Binary*** (n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

count ← 1

while $n > 1$ **do**

  count ← count + 1

  $n \leftarrow \lfloor n/2 \rfloor$

return count

```python
def binary(n):
    count = 1

    while n > 1:
        count += 1
        n /= 2

    return count
```

ALGORITHM $BinRec(n)$

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return 1**

else return $BinRec(\lfloor n/2 \rfloor) + 1$

```python
def binary_rec(n):
    if n == 1: return 1

    return binary_rec(n//2) + 1
```

```python
def merge(my_list, aux, lo, mid, hi):
       # copy items from my_list to aux list
        aux[lo:hi+1] = my_list[lo:hi+1]
        i = lo
        j = mid + 1

        # i ----> mid | j ----> hi

        for k in range(lo, hi+1):
            if i > mid: # i cross mid, copy from j half
                my_list[k] = aux[j]
                j += 1
            elif j > hi: # j cross hi, copy from i half
                my_list[k] = aux[i]
                i += 1
            elif aux[j] < aux[i]:
                my_list[k] = aux[j]
                j += 1
            else:
                my_list[k] = aux[i]
                i+= 1
```

```python
def merge_sort(my_list, aux, lo, hi):
    if hi <= lo: return

    mid = (hi + lo) // 2

    divide(my_list, aux, lo, mid)       # first half lo --> mid
    divide(my_list, aux, mid + 1, hi)   # second half mid+1 --> hi

    merge(my_list, aux, lo, mid, hi)    # sort lo --> hi
```

```python
def swap(numbers, i, j):
    temp = numbers[i]
    numbers[i] = numbers[j]
    numbers[j] = temp
```

ALGORITHM Quicksort $(A[l..r])$

//Sorts a subarray by quicksort.

//Input: Subarray of array $A[0..n-1]$, defined by its left and right

// indices l and r.

//Output: Subarray $A[l..r]$ sorted in nondecreasing order.

if $l < r$

    $s \leftarrow Partition(A[l..r])$ *//s is a split position.*

    Quicksort$(A[l..s-1])$

    Quicksort$(A[s+1..r])$

```python
def quick_sort(numbers, lo, hi):
    if lo < hi:
        # choose partition type
        s = partition(numbers, lo, hi)
        quick_sort(numbers, lo, s - 1)
        quick_sort(numbers, s + 1, hi)
        return numbers
```

ALGORITHM LomutoPartition $(A[l..r])$
//Partitions subarray by Lomuto's algorithm using first element as pivot.
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right indices l and r $(l \leq r)$.
//Output: Partition of $A[l..r]$ and the new position of the pivot.
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
    if $A[i] < p$
        $s \leftarrow s + 1$
        Swap $(A[s], A[i])$
Swap $(A[l], A[s])$
return s

```python
"""
Partitions subarray by Lomuto's algorithm using first element as pivot.
Output: Partition of numbers and the new position of the pivot.
"""
def lomuto_partition(numbers, lo, hi):
    pivot = numbers[lo]
    s = lo

    for i in range(lo+1, hi):
        # If current element is smaller than or equal to pivot
        if numbers[i] <= pivot:
            # increment index of smaller element
            s += 1
            swap(numbers, s, i)

    swap(numbers, lo, s)

    return s
```

ALGORITHM HoarePartition $(A[l..r])$

//Partitions subarray by Hoare's algorithm using first element as pivot.

//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right indices l and r $(l \le r)$.

//Output: Partition of $A[l..r]$, with the split position returned as this function's value.

$p \leftarrow A[l]$

$i \leftarrow l;\ j \leftarrow r + 1$

repeat

  repeat $i \leftarrow i + 1$ **until** $A[i] \ge p$

  repeat $j \leftarrow j - 1$ **until** $A[j] \le p$

  swap$(A[i], A[j])$

until $i \ge j$

swap $(A[i], A[j])$ //*undo last swap when* $i \ge j$

swap $(A[l], A[j])$

return j

```
"""
Partitions subarray by Hoare's algorithm using first element as pivot.
Output: Partition of numbers with the split position returned as this
function's value
"""
def hoare_partition(numbers, lo, hi):
    p = numbers[lo]
    i = lo+1
    j = hi

    while i < j:

        while numbers[i] < p:
            i += 1
        while numbers[j] > p:
            j -= 1

        swap(numbers, i, j)

    swap(numbers, i, j)
    swap(numbers, lo, j)

    return j
```

ALGORITHM Prim(G)

// Prim's algorithm for constructing a minimum spanning tree.

// Input: A weighted connected graph $G = (V, E)$.

// Output: E_T , the set of edges composing a minimum spanning tree of G.

$V_T \leftarrow \{v_0\}$ *//the set of tree vertices can be initialized with any vertex.*

$E_T \leftarrow \emptyset$

for $i \leftarrow$ **1 to** $|V| - 1$ **do**

find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$.

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

```
from collections import defaultdict
import heapq

def prims(graph, s):
    mst = defaultdict(set)
    visited = set([s])
    edges = [
        (cost, s, w)
        for w, cost in graph[s]
    ]
    heapq.heapify(edges)

    while edges:
        cost, v, w = heapq.heappop(edges)
        if w not in visited and w in graph:
            visited.add(w)
            mst[v].add(w)
            for adj, cost in graph[w]:
                if adj not in visited:
                    heapq.heappush(edges, (cost, w, adj))

    return mst

graph = {
 'A': {('C', 31)},
 'B': {('C', 15)},
 'C': {('G', 77), ('H', 40)},
 'E': {('C', 17), ('I', 3)},
 'G': {('B', 22), ('E', 23)},
 'H': {('G', 66)},
 'I': {('J', 70), ('K', 31)},
}
print(prims(graph, 'A'))
```

ALGORITHM Dijkstra (G, s)

//Dijkstra's algorithm for single-source shortest paths.

//Input: A weighted connected graph $G = (V, E)$ with nonnegative weights and its vertex s.

//Output: The length d_v of a shortest path from s to v and its penultimate vertex p_v for every vertex v in V.

Initialize (Q) //initialize priority queue to empty.

for every vertex v **in** V **do**

  $d_v \leftarrow \infty; p_v \leftarrow null$

  Insert (Q, v, d_v) //initialize vertex priority in the priority queue.

$d_s \leftarrow 0$; *Decrease* (Q, s, d_s) //*update priority of* s *with* d_s.

$V_T \leftarrow \emptyset$

for $i \leftarrow 0$ to $|V| - 1$ **do**

  $u^* \leftarrow$ DeleteMin (Q) //delete the minimum priority element.

  $V_T \leftarrow V_T \cup \{u^*\}$

  for every vertex u in $V - V_T$ that is adjacent to u* **do**

    if $d_{u^*} + w(u^*, u) < d_u$ **do**

      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$

      Decrease (Q, u, d_u)

```python
# Time-Complexity O(E log V)
from queue import PriorityQueue

def dijkstra(graph, s):
    seen = set()
    cost = {s: 0}
    parent_path = {s: None}
    pq = PriorityQueue()

    pq.put((0, s))
    while not pq.empty():
        _, v = pq.get()
        seen.add(v)

        for w, distance in graph[v]:
            if w in seen: continue

            old_cost = cost.get(w, float('inf'))
            new_cost = cost[v] + distance

            if new_cost < old_cost:
                pq.put((new_cost, w))
                cost[w] = new_cost
                parent_path[w] = v

    return parent_path
```

```python
G = {
 'A': {('C', 31)},
 'B': {('C', 15), ('J', 58)},
 'C': {('C', 60), ('G', 77), ('H', 40)},
 'E': {('C', 17), ('E', 55), ('I',  3)},
 'G': {('B', 22), ('E', 23), ('G', 31)},
 'H': {('G', 66)},
 'I': {('J', 70), ('K', 31)},
 'J': {('H',  8), ('K', 28)},
 'K': {('I', 13)}
}

parent_path = dijkstra(G, 'A')
```