Pseudocode for Sorting Algorithms

This is a collection of all the pseudocode for the sorting algorithms discussed.

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

MERGE-SORT(A)

```
if A.length > 1
    m = ⌊A.length/2⌋
    L = A[0..m − 1]
    R = A[m..A.length]
    MERGE-SORT(L) // keep splitting subarrays, takes Θ(log_2 n) time
    MERGE-SORT(R)
    let i, j, k = 0 // initialize indices for A and subarrays
    // Copy smallest value from L[i] or R[j] into A[k]
    while i < L.length and j < R.length
        if L[i] < R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        k = k + 1
    while i < L.length
        A[k] = L[i] // copy remaining elements
        i = i + 1
        k = k + 1
    while j < R.length
        A[k] = R[j] // copy remaining elements
        j = j + 1
        k = k + 1
```

```
PARENT(i)
    return ⌊i/2⌋
```

```
LEFT(i)
    return 2i
```

```
RIGHT(i)
    return 2i + 1
```

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l // index of largest element is left child
    else
        largest = i // out of parent and left child, parent is larger
    // compare largest out of parent and left with the right (if it's in the valid heap)
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r // right child has the largest value
    if largest ≠ i
        // largest node was a child, keep going down to ensure max-heap property is not violated
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

```
MAX-HEAPIFY-ITERATIVE(A, i)
    while TRUE
        l = LEFT(i)
        r = RIGHT(i)
        if l ≤ A.heap-size and A[l] > A[i]
            largest = l
        else
            largest = i
        if r ≤ A.heap-size and A[r] > A[i]
            largest = r
        if largest = i
            return
        exchange A[i] with A[largest]
        i = largest
```

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MEAX-HEAPIFY(A, 1)
```

```
HEAP-MAXIMUM(A)
return A[1]
```

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
// Now adjust max-heap by propagating largest value to A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
// Compare parent and current values, go up to satisfy max-heap property
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

```
HEAP-DELETE(A, i)
swap A[i] and A[A.heap-size]
A.heap-size = A.heap-size − 1
if A[i] > A[PARENT(i)]
    HEAP-INCREASE-KEY(A, i, A[i])
else
    MAX-HEAPIFY(A, i) // check if we need to stop or keep going down
```

```
MIN-HEAP-MINIMUM(A)
return A[1]
```

```
MIN-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] < A[i]
    smallest = l
else smallest = i
if r ≤ A.heap-size and A[r] < A[smallest]
    smallest = r
if smallest ≠ i
    exchange A[i] with A[smallest]
    MIN-HEAPIFY(A, smallest)
```

```
MIN-HEAP-EXTRACT-MIN(A)
if A.heap-size < 1
    error "heap underflow"
min = A[1]
A[1] = A[A.heap-size] // violate min-heap property to re-adjust heap
A.heap-size = A.heap-size − 1 // decrease heap size, eliminating min from valid heap
MIN-HEAPIFY(A, 1)
return min
```

```
MIN-HEAP-DECREASE-KEY(A, i, key)
if key > A[i]
    error "new key is larger than current key"
A[i] = key
while i > 1 and A[PARENT(i)] > A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
MIN-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = ∞
MIN-HEAP-DECREASE-KEY(A, A.heap-size, key)
```

PARTITION(A, p, r)

```
x = A[r] // this partition procedure always picks the right-most element
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
```

QUICKSORT(A, p, r)

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

COUNTING-SORT(A, B, k)

```
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0 // initialize
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

RADIX-SORT(A, d)

```
for i = 1 to d
    use a stable sort to sort array A on digit i // like counting sort
```

BUCKET-SORT(A)

```
n = A.length
let B[0..n - 1] be a new array
for i = 0 to n - 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n - 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```