

Pseudocode for Sorting Algorithms

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

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

MERGE-SORT(A)

```
1  if  $A.length > 1$ 
2       $m = \lfloor A.length/2 \rfloor$ 
3       $L = A[0..m-1]$ 
4       $R = A[m..A.length]$ 
5      MERGE-SORT( $L$ ) // keep splitting subarrays, takes  $\Theta(\log_2 n)$  time
6      MERGE-SORT( $R$ )
7      let  $i, j, k = 0$  // initialize indices for  $A$  and subarrays
8      // Copy smallest value from  $L[i]$  or  $R[j]$  into  $A[k]$ 
9      while  $i < L.length$  and  $j < R.length$ 
10         if  $L[i] < R[j]$ 
11              $A[k] = L[i]$ 
12              $i = i + 1$ 
13         else
14              $A[k] = R[j]$ 
15              $j = j + 1$ 
16          $k = k + 1$ 
17     while  $i < L.length$ 
18          $A[k] = L[i]$  // copy remaining elements
19          $i = i + 1$ 
20          $k = k + 1$ 
21     while  $j < R.length$ 
22          $A[k] = R[j]$  // copy remaining elements
23          $j = j + 1$ 
24          $k = k + 1$ 
```

```

PARENT(i)
1  return  $\lfloor i/2 \rfloor$ 

LEFT(i)
1  return  $2i$ 

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

MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if  $l \leq A.heap-size$  and  $A[l] > A[i]$ 
4      largest = l // index of largest element is left child
5  else
6      largest = i // out of parent and left child, parent is larger
7  // compare largest out of parent and left with the right (if it's in the valid heap)
8  if  $r \leq A.heap-size$  and  $A[r] > A[largest]$ 
9      largest = r // right child has the largest value
10 if largest  $\neq i$ 
11     // largest node was a child, keep going down to ensure max-heap property is not violated
12     exchange  $A[i]$  with  $A[largest]$ 
13     MAX-HEAPIFY(A, largest)

MAX-HEAPIFY-ITERATIVE(A, i)
1  while TRUE
2      l = LEFT(i)
3      r = RIGHT(i)
4      if  $l \leq A.heap-size$  and  $A[l] > A[i]$ 
5          largest = l
6      else
7          largest = i
8      if  $r \leq A.heap-size$  and  $A[r] > A[i]$ 
9          largest = r
10     if largest = i
11         return
12     exchange  $A[i]$  with  $A[largest]$ 
13     i = largest

BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY(A, i)

```

```

HEAPSORT( $A$ )
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size$ 
5      MAX-HEAPIFY( $A, 1$ )

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

HEAP-EXTRACT-MAX( $A$ )
1  if  $A.heap-size < 1$ 
2      error "heap underflow"
3   $max = A[1]$ 
4  // Now adjust max-heap by propagating largest value to  $A[1]$ 
5   $A[1] = A[A.heap-size]$ 
6   $A.heap-size = A.heap-size - 1$ 
7  MAX-HEAPIFY( $A, 1$ )
8  return  $max$ 

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

MAX-HEAP-INSERT( $A, key$ )
1   $A.heap-size = A.heap-size + 1$ 
2   $A[A.heap-size] = -\infty$ 
3  HEAP-INCREASE-KEY( $A, A.heap-size, key$ )

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

```

MIN-HEAP-MINIMUM(A)

1 **return** $A[1]$

MIN-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] < A[i]$ 
4       $\text{smallest} = l$ 
5  else  $\text{smallest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{smallest}]$ 
7       $\text{smallest} = r$ 
8  if  $\text{smallest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{smallest}]$ 
10     MIN-HEAPIFY( $A, \text{smallest}$ )
```

MIN-HEAP-EXTRACT-MIN(A)

```
1  if  $A.\text{heap-size} < 1$ 
2      error "heap underflow"
3   $\text{min} = A[1]$ 
4   $A[1] = A[A.\text{heap-size}]$  // violate min-heap property to re-adjust heap
5   $A.\text{heap-size} = A.\text{heap-size} - 1$  // decrease heap size, eliminating  $\text{min}$  from valid heap
6  MIN-HEAPIFY( $A, 1$ )
7  return  $\text{min}$ 
```

MIN-HEAP-DECREASE-KEY(A, i, key)

```
1  if  $\text{key} > A[i]$ 
2      error "new key is larger than current key"
3   $A[i] = \text{key}$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] > A[i]$ 
5      exchange  $A[i]$  with  $A[\text{PARENT}(i)]$ 
6       $i = \text{PARENT}(i)$ 
```

MIN-HEAP-INSERT(A, key)

```
1   $A.\text{heap-size} = A.\text{heap-size} + 1$ 
2   $A[A.\text{heap-size}] = \infty$ 
3  MIN-HEAP-DECREASE-KEY( $A, A.\text{heap-size}, \text{key}$ )
```

PARTITION(A, p, r)

```
1   $x = A[r]$  // this partition procedure always picks the right-most element
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

QUICKSORT(A, p, r)

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$  // initialize
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ 
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ 
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

RADIX-SORT(A, d)

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

BUCKET-SORT(A)

```
1  let  $B[0..n - 1]$  be a new array
2   $n = A.length$ 
3  for  $i = 0$  to  $n - 1$ 
4      make  $B[i]$  an empty list
5  for  $i = 1$  to  $n$ 
6      insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$ 
7  for  $i = 0$  to  $n - 1$ 
8      sort list  $B[i]$  with insertion sort
9  concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order
```