The Sound of Sorting Algorithm Cheat Sheet

```
Function selectionSort(A : Array of Element; n : ℕ)
    for i := 1 to n do
        min := i
        for j := i + 1 to n do                      // find smallest element
            if A[j] < A[min] then
                min := j
        endfor
        swap(A[i], A[min])                          // swap element to the beginning
        invariant A[1] ≤ ··· ≤ A[i]
    endfor

Function insertionSort(A : Array of Element; n : ℕ)
    for i := 2 to n do                              // {A[1]} is sorted
        j := i
        while (j > 0) & (A[j − 1] > A[j])           // find right position j
            swap(A[j − 1], A[j])                    // move larger elements to the back
            j := j − 1
        endwhile
        invariant A[1] ≤ ··· ≤ A[i]
    endfor

Function mergeSort(A : Array of Element; lo, hi : ℕ)
    if hi − lo ≤ 1 then return                      // base case
    mid := (lo + hi)/2                              // middle element
    mergeSort(lo, mid), mergeSort(mid, hi)          // sort halves
    B := allocate (Array of Element size hi − lo)
    i := lo,   j := mid,   k := 1                   // running indexes
    while (i < mid) & (j < hi)
        if A[i] < A[j]   B[k++] := A[i++]           // merge!
        else             B[k++] := A[j++]
    endwhile
    while i < mid   do B[k++] := A[i++]             // copy remainder
    while j < hi    do B[k++] := A[j++]
    A[lo, ..., hi − 1] := B[1, ..., (hi − lo)]      // copy back
    dispose (B)

Procedure bubbleSort(A : Array [1 ... n] of Element)
    for i := 1 to n do
        for j := 1 to n − i do
            if A[j] > A[j + 1] then                 // If right is smaller,
                swap(A[j], A[j + 1])                // move to the right

Procedure heapSort(A : Array [1 ... n] of Element)
    buildHeap(A)                                    // construct a max-heap in the array
    while n > 1
        swap(A[1], A[n])                            // take maximum
        n := n − 1                                  // shrink heap area in array
        siftDown(A, 1)                              // correctly order A[1] in heap

Procedure buildHeap(A : Array [1 ... n] of Element)
    for i := ⌊n/2⌋ downto 1 do
        siftDown(i)                                 // reestablish max-heap invariants

Procedure siftDown(A : Array [1 ... n] of Element; i : ℕ)
    if 2i > n then return
    k := 2i                                         // select right or left child
    if (2i + 1 ≤ n) & (A[2i] ≤ A[2i + 1]) then      // find smaller child
        k := k + 1
    if A[i] < A[k] then                             // if child is larger than A[i],
        swap(A[i], A[k])                            // switch with child
        siftDown(A, k)                              // and move element further down

Procedure cocktailShakerSort(A : Array [1 ... n] of Element)
    lo := 1, hi := n, mov := lo
    while lo < hi do
        for i := hi downto lo + 1 do
            if A[i − 1] > A[i] then                 // move smallest element in
                swap(A[i − 1], A[i]), mov := i      // A[hi..lo] to A[lo]
        endfor
        lo := mov
        for i := lo to hi − 1 do
            if A[i] > A[i + 1] then                 // move largest element in
                swap(A[i], A[i + 1]), mov := i      // A[lo..hi] to A[hi]
        endfor
        hi := mov

Procedure gnomeSort(A : Array [1 ... n] of Element)
    i := 2
    while i ≤ n do
        if A[i] ≥ A[i − 1] then                     // move to right while
            i++                                     // elements grow larger
        else
            swap(A[i], A[i − 1])                    // swap backwards while
            if i > 2 then i−−                       // element grow smaller
    endwhile
```


http://panthema.net/2013/sound-of-sorting

```
Procedure quickSort(A : Array of Element; ℓ, r : ℕ)
    if ℓ ≥ r then return
    q := pickPivotPos(A, ℓ, r)
    m := partition(A, ℓ, r, q)
    quickSort(A, ℓ, m − 1),   quickSort(A, m + 1, r)

Function partition(A : Array of Element; ℓ, r : ℕ, q : ℕ)
    p := A[q]                                          // pivot element
    swap(A[q], A[r])                                   // swap to the end
    i := ℓ
                  ℓ       i       j       r
    invariant   | ≤ p   | > p   |  ?    | p |
    for j := ℓ to r − 1 do
        if A[j] ≤ p then
            swap(A[i], A[j]),  i++                     // move smaller to the front
                  ℓ           i           r
    assert      |   ≤ p     |   > p     | p |
    swap(A[i], A[r])                                   // move pivot into the middle
                  ℓ           i           r
    assert      |   ≤ p     | p |   > p     |
    return i

Procedure quickSortTernary(A : Array of Element; ℓ, r : ℕ)
    if ℓ ≥ r then return
    q := pickPivotPos(A, ℓ, r)
    (m, m′) := partitionTernary(A, ℓ, r, q)
    quickSortTernary(A, ℓ, m − 1),   quickSortTernary(A, m′ + 1, r)

Function partitionTernary(A : Array of Element; ℓ, r : ℕ; q : ℕ)
    p := A[q]                                          // pivot element
    i := ℓ,   j := ℓ,   k := r
                  ℓ       i       j   k   r
    invariant   | < p   | > p   | ?   | = p |
    while (j ≤ k)                                      // three-way comparison
        if A[j] = p       then swap(A[j], A[k]),   k−− ;
        else if A[j] < p then swap(A[j], A[i]),   i++ , j++ ;
        else   j++ ;
                  ℓ           i           k       r
    assert      |   < p     |   > p     | = p |
    i′ := i + r − k + 1
    swap(A[i . . . i′], A[k + 1 . . . r])              // move = p area to the middle
                  ℓ           i  i′               r
    assert      |   < p     | = p |   > p     |
    return (i, i′)

Procedure LSDRadixSort(A : Array [1 . . . n] of Element)
K := 4                                                 // number of buckets per round
D := ⌈log_K(max{A[i] + 1 | i = 1, . . . , n})⌉         // calculate number of rounds
B := allocate (Array of Element size n)                // temporary array B
for d := 0 to D − 1 do                                 // sort by the d-th K-digit.
    redefine key(x) := (x div K^d) mod K
    KSortCopy(A, B, n),  swap(A, B)                    // sort from A to B, and swap back
    invariant  A ist nach den K-Ziffern d..0 sortiert.
dispose (B)

Procedure KSortCopy(A, B : Array [1 . . . n] of Element; K : ℕ)
    c = ⟨0, . . . , 0⟩  : Array [0 . . . K − 1] of ℕ
    for i := 1 to n do  c[key(A[i])]++                 // count occurrences
    sum := 1
    for k := 0 to K − 1 do                             // exclusive prefix sum
        next := sum + c[k],   c[k] := sum,   sum := next
    for i := 1 to n do
        B[ c[key(A[i])]++ ] := A[i]                    // move element A[i] into bucket of B

Procedure MSDRadixSort(A : Array [1 . . . n] of Element)
K := 4                                                 // number of buckets per round
D := ⌈log_K(max{A[i] + 1 | i = 1, . . . , n})⌉         // count number of round
MSDRadixSortRec(A, D − 1, K)

Procedure MSDRadixSortRec(A : Array [1 . . . n] of Element; d, K : ℕ)
c = ⟨0, . . . , 0⟩  : Array [0 . . . K − 1] of ℕ        // KSort with in-place permuting
redefine key(x) := (x div K^d) mod K
for i := 1 to n do  c[key(A[i])]++                     // count occurrences
b = ⟨0, . . . , 0⟩  : Array [0 . . . K] of ℕ
sum := 1
for k := 0 to K do                                     // inclusive prefix sum into b
    sum := sum + c[k],   b[k] := sum
assert b[K] = n
for i := 1 to n do
    while ( j := −− b[key(A[i])] ) > i                 // walk on cycles until i
        swap(A[i], A[j])                               // move A[i] into right bucket
    i := i + c[key(A[i])]                              // bucket of A[i] is finished
invariant  A ist nach den K-Ziffern d..(D − 1) sortiert
if d = 0 return                                        // done?
for k := 0 to K − 1 do                                 // recursion into each of the K buckets if
    if c[k] > 1                                        // it contains two or more elements
        MSDRadixSortRec(A[b[k] . . . b[k + 1] − 1], d − 1, K)
dispose (b), dispose (c)
```


http://panthema.net/2013/sound-of-sorting