Insertion sort pseudocode

```
INSERTION-SORT(A)
for j ← 2 to n
    do key ← A[j]
       ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Implementation of merge sort

```java
public static void mergeSort(int[] a) {
   int[] tmpArray = new int[a.length];
   mergeSort(a, tmpArray, 0, a.length - 1);
}

private static void mergeSort(int[] a, int[] tmpArray, int left, int right) {
   if (left < right) {
         int center = (left + right) / 2;
         mergeSort(a, tmpArray, left, center);
         mergeSort(a, tmpArray, center + 1, right);
         merge(a, tmpArray, left, center + 1, right);
   }
}
```

Implementation of merge sort

```
private static void merge(int[] a, int[] tmpArray, int leftPos, int rightPos, int rightEnd){
   int leftEnd = rightPos – 1, tmpPos = leftPos;
   int numElements = rightEnd - leftPos + 1;

   while (leftPos <= leftEnd && rightPos <= rightEnd)
         if (a[leftPos] <= a[rightPos])
                  tmpArray[tmpPos++] = a[leftPos++];
         else
                  tmpArray[tmpPos++] = a[rightPos++];

   while (leftPos <= leftEnd)
         tmpArray[tmpPos++] = a[leftPos++];

   while (rightPos <= rightEnd)
         tmpArray[tmpPos++] = a[rightPos++];

   for (int i = 0; i < numElements; i++, rightEnd--)
         a[rightEnd] = tmpArray[rightEnd];
}
```

Master theorem: intuition

- Recurrence: $T(n) \leq a \cdot T(n/b) + O(n^d)$
- An algorithm that divides a problem of size n into a subproblems, each of size n / b

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

a: number of subproblems (branching factor)
b: factor by which input size shrinks (shrinking factor)
d: need to do $O(n^d)$ work to create subproblems + “merge” their solutions

Find max subarray crossing midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

QuickSort

```
private static int median3(int[] a, int left, int right) {
        // Ensure a[left] <= a[center] <= a[right]
        int center = (left + right) / 2;
        if (a[center] < a[left])
                swap (a, left, center);
        if (a[right] < a[left])
                swap (a, left, right);
        if (a[right] < a[center])
                swap (a, center, right);

        // Place pivot at position right - 1
        swap (a, center, right - 1);
        return a[right - 1];
}
```

QuickSort

```
/* Main quicksort routine */
private static void quicksort(int[ ] a, int left, int right) {
        if (left + CUTOFF <= right) {
                int pivot = median3(a, left, right);
                // Begin partitioning
                int i = left+1, j = right - 2;
                while (true) {
                        while (a[i] < pivot) {i++;}
                        while (a[j] > pivot) {j--;}
                        if (i >= j) break; // i meets j
                        swap (a, i, j);
                }
                swap (a, i, right - 1); // Restore pivot
                quicksort(a, left, i - 1); // Sort small elements
                quicksort(a, i + 1, right); // Sort large elements
        } else
                insertionSort(a, left, right);
}
```

SelectionSort(arr, n)

```
if n ≤ 1
  return arr
maxnum = arr[0]
maxIndex = 0
for i = 1 to n -1
    if maxnum < arr[i]
       maxnum = arr[i]
       maxIndex = i
arr[maxIndex] = arr[n-1]
arr[n-1] = maxnum
SelectionSort(arr, n-1)
```

ShellSort with {1,2,4,8,…,n/2}

```
public static void shellSort(int[ ] a) {
    int j;
    for (int gap = a.length/2; gap > 0; gap /=2)
        for (int i = gap; i < a.length; i++) {
            int tmp = a[i];
            for (j = i; j >= gap && tmp < a[j-gap]; j-= gap)
                a[j] = a[j-gap];
            a[j] = tmp;
        }
}
```

CountingSort

Algorithm CountingSort(S)
(values in S are between 0 and m-1)

```
for j ← 0 to m-1 do     // initialize m buckets
  b[j] ← 0
for i ← 0 to n-1 do     // place elements in their appropriate buckets
  b[S[i]] ← b[S[i]] + 1
i ← 0
for j ← 0 to m-1 do     // place elements in buckets
  for r ← 1 to b[j] do // back in S
      S[i] ← j
      i ← i + 1
```

BucketSort

BUCKET-SORT(A, n)

for i ← 1 **to** n
    do insert A[i] into list B[⌊nA[i]⌋]
for i ← 0 **to** n - 1
    do sort list B[i] with QuickSort
concatenate lists B[0], B[1], . . . , B[n -1]
together in order
return the concatenated lists

$\left.\begin{array}{l}\text{for } i \leftarrow 1 \text{ to } n \\ \quad \text{do insert } A[i] \text{ into list } B[\lfloor nA[i] \rfloor]\end{array}\right\}$ O(n)

$\left.\begin{array}{l}\text{for } i \leftarrow 0 \text{ to } n-1 \\ \quad \text{do sort list } B[i] \text{ with QuickSort}\end{array}\right\}$ Θ(n)

$\left.\begin{array}{l}\text{concatenate lists } B[0], B[1], \ldots, B[n-1] \text{ together in order} \\ \text{return the concatenated lists}\end{array}\right\}$ O(n)

Θ(n)

Codes (1/2)

```
// items to be sorted are in {0,…,10^d-1},
// i.e., the type of d-digit integers
void radixsort(int A[], int n, int d)
{
   int i;
   for (i=0; i<d; i++)
      bucketsort(A, n, i);
}

// To extract d-th digit of x
int digit(int x, int d)
{
   int i;
   for (i=0; i<d; i++)
      x /= 10; // integer division
   return x%10;
}
```

Codes (2/2)

```
void bucketsort(int A[], int n, int d)
// stable-sort according to d-th digit
{
   int i, j;
   Queue *C = new Queue[10];
   for (i=0; i<10; i++) C[i].makeEmpty();
   for (i=0; i<n; i++)
      C[digit(A[i],d)].EnQueue(A[i]);
   for (i=0, j=0; i<10; i++)
      while (!C[i].empty())
      { // copy values from queues to A[]
         C[i].DeQueue(A[j]);
         j++;
      }
}
```

10 classic sorting algorithms

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	√	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	×	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	√	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
HeapSort	×	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$
QuickSort	×	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(1)$
ShellSort	×	$O(n)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$
CountingSort	√	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$
BucketSort	√	$O(n)$	$O(n+k)$	$O(n^2)$	$O(k)$
RadixSort	√	$O(nk)$	$O(nk)$	$O(nk)$	$O(n)$