

# Sorting an Array

When the elements of an array are in random order we may want to rearrange the array into ascending or descending order. One reason to do this is to be able to use a binary search on the array. There are various methods devised to do the sorting. Some sorts are the selection sort, bubble sort, insertion sort, quicksort, merge sort, and shell sort. Some sorts are better suited for certain types of data and the amount of data in the array. All sorts are not equal. Some are fast, some slow, some efficient, some less efficient.

## Selection Sort

The basic theory behind this sort is to search an array `a` for the smallest (or largest) value and place it in cell position 0, then search the remaining cells for the next smallest (or largest) value and place it in cell position 1, etc. After `a.length-1` passes the array is in sorted order. The selection sort is an incremental sort that does not allow an effective and automatic loop exit if the array becomes ordered during an early pass.

```
void selectionSort(int[] a){
    for (int i = 0; i < a.length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[minIndex]){minIndex = j;}
        }
        if (minIndex != i){
            int temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
        }
    }
}
```

Note: Our textbook breaks the selectionSort into three smaller methods: selectionSort, findMinimum, and swap. You can see these methods on page 417 of your textbook. Our textbook also breaks the bubble sort (shown below) into two smaller methods: bubbleSort and swap. You can see these methods on pages 417-419.

## Bubble Sort

The basic theory behind this sort is to search adjacent pairs of items in the array. Whenever two items are out of order with respect to each other, they are swapped. After one pass through the array, the item that comes last in order will be in the final array position. That is, the last item will “sink” to the bottom of the array and preceding items will gradually “percolate” to the top. The bubble sort is an incremental sort that does allow for an early exit if the variable `exchangeMade` is false.

```
void bubbleSort(int[] a){
    int k = 0;
    boolean exchangeMade = true;
    while ((k < a.length - 1) && exchangeMade){
        exchangeMade = false;
        k++;
        for (int j = 0; j < a.length - k; j++){
            if (a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                exchangeMade = true;
            }
        }
    }
}
```

## Insertion Sort

The insertion sort works similar to the way you might organize a hand of cards. The unsorted cards face down on the table and are picked up one by one. As each new unsorted card is picked up, it is inserted into the correct place in your organized hand of cards. The insertion sort works by splitting the list into two sublists. The first sublist, which is always fully sorted, gets larger as the sort progresses. It can be thought of as the hand that holds the organized cards. The second sublist is unsorted and contains all the elements not yet inserted into the first sublist. The second sublist gets smaller as the first sublist gets larger. The second sublist is like the table from where cards are picked up. The insertion sort is an incremental sort that does not allow an effective and automatic loop exit if the array becomes ordered during an early pass.

```
void insertionSort(int[] a){
    int itemToInsert, j;
    boolean stillLooking;
    for (int k = 1; k < a.length; k++){
        itemToInsert = a[k];
        j = k - 1;
        stillLooking = true;

        while ((j >= 0) && stillLooking) {
            if (itemToInsert < a[j]) {
                a[j+1] = a[j];
                j--;
            } else {
                stillLooking = false;
            }
        }
        a[j+1] = itemToInsert;
    }
}
```

## QuickSort

The basic theory behind this sort is: Break an array into two parts and then move elements around so that all the larger values are in one end and all the smaller values are in the other. Each of the two parts is then subdivide in the same manner, and so on, until the subparts contain only a single value, at which point the array is sorted. The quicksort is a divide and conquer method. It is recognized as being one of the most efficient sorting methods. Although with some data it is not always the most efficient, the average quicksort requires very few steps. The quicksort method can be coded using either an iterative or a recursive approach. Show below is the recursive approach.

```
void quickSort(int[] a, int left, int right){

    if (left >= right) return;

    int i = left;
    int j = right;
    int pivotValue = a[(left + right)/2];
    while (i < j){
        while (a[i] < pivotValue) i++;
        while (pivotValue < a[j]) j--;
        if (i <= j) {
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
        }
    }
    quickSort(a, left, j);
    quickSort(a, i, right);
}
```

## Merge Sort

The merge sort begins by placing each element into its own individual list, thus dividing the original list into equal sized parts. The merge sort then combines every two small lists into one list. The list is merged in a way so that the newly created lists are all sorted. Every two new lists are then merged into a new sorted list. This continues until all smaller lists have been merged into a single list, which is completely sorted. The merge sort is a divide and conquer method that uses a recursive approach. Our textbook takes the merge sort and breaks it down into three methods show below.

```
void mergeSort(int[] a){

    int[] copyBuffer = new int[a.length];
    mergeSortHelper(a, copyBuffer, 0, a.length - 1);
}

void mergeSortHelper(int[] a, int[] copyBuffer, int low, int high){
    if (low < high){
        int middle = (low + high) / 2;
        mergeSortHelper(a, copyBuffer, low, middle);
        mergeSortHelper(a, copyBuffer, middle + 1, high);
        merge(a, copyBuffer, low, middle, high);
    }
}

void merge(int[] a, int[] copyBuffer, int low, int middle, int high){

    int i1 = low, i2 = middle + 1;

    for (int i = low; i <= high; i++){
        if (i1 > middle)
            copyBuffer[i] = a[i2++];
        else if (i2 > high)
            copyBuffer[i] = a[i1++];
        else if (a[i1] < a[i2])
            copyBuffer[i] = a[i1++];
        else
            copyBuffer[i] = a[i2++];
    }
    for (int i = low; i <= high; i++)
        a[i] = copyBuffer[i];
}
```

Examples:

- 1) **Selection Sort** - Sort the following array into ascending order. Show the list of numbers for each step. Be sure to show braces around each list of numbers, a vertical line to indicate the two different sublists of sorted and unsorted numbers, a circle around number being sorted, and an arrow to indicate which numbers got swapped.

Initial Unsorted List: {12, 34, 25, 10, 89, 52, 48}

Step 1: \_\_\_\_\_

Step 2: \_\_\_\_\_

Step 3: \_\_\_\_\_

Step 4: \_\_\_\_\_

Step 5: \_\_\_\_\_

Step 6: \_\_\_\_\_

Step 7: \_\_\_\_\_

- 2) **Insertion Sort** - Sort the following array into ascending order. Show the list of numbers for each step. Be sure to show braces around each list of numbers, a vertical line to indicate the two different sublists of sorted and unsorted numbers, a circle around number being sorted, and an arrow to indicate which numbers got swapped.

Initial Unsorted List: {60, 12, 34, 25, 101, 89, 52}

Step 1: \_\_\_\_\_

Step 2: \_\_\_\_\_

Step 3: \_\_\_\_\_

Step 4: \_\_\_\_\_

Step 5: \_\_\_\_\_

Step 6: \_\_\_\_\_

Step 7: \_\_\_\_\_

- 3) **Bubble Sort** - Sort the following array into ascending order. Show the list of numbers for each step. Be sure to show braces around each list of numbers, a vertical line to indicate the two different sublists of sorted and unsorted numbers, a circle around number being sorted, and an arrow to indicate which numbers got swapped. Also mark out the numbers when a swap is made and indicate the new numbers above the old numbers.

Initial Unsorted List: {78, 54, 39, 90, 18, 27, 63}

Step 1: \_\_\_\_\_

Step 2: \_\_\_\_\_

Step 3: \_\_\_\_\_

Step 4: \_\_\_\_\_

Step 5: \_\_\_\_\_

Step 6: \_\_\_\_\_

Step 7: \_\_\_\_\_

- 4) **Quick Sort** - Sort the following array into ascending order. Show the list of numbers for each step. Be sure to show braces around each list of numbers, a vertical line to indicate the two different sublists of sorted and unsorted numbers, a circle around number being sorted, and an arrow to indicate which numbers got swapped. Also mark out the numbers when a swap is made and indicate the new numbers above the old numbers.

Initial Unsorted List: {5, 12, 3, 11, 2, 7, 20, 10, 8, 4, 9}

Step 1: \_\_\_\_\_

Step 2: \_\_\_\_\_

Step 3: \_\_\_\_\_

Step 4: \_\_\_\_\_

Step 5: \_\_\_\_\_

Step 6: \_\_\_\_\_

Step 7: \_\_\_\_\_

Step 8: \_\_\_\_\_

Step 9: \_\_\_\_\_

Step 10: \_\_\_\_\_

- 5) **Merge Sort** - Sort the following array into ascending order. Show the list of numbers for each step. Be sure to show braces around each list of numbers, a vertical line to indicate the two different sublists of sorted and unsorted numbers, a circle around number being sorted, and an arrow to indicate which numbers got swapped. Also mark out the numbers when a swap is made and indicate the new numbers above the old numbers.

Initial Unsorted List: {73, 82, 45, 87, 54, 29, 21, 79}

Step 1: \_\_\_\_\_

Step 2: \_\_\_\_\_

Step 3: \_\_\_\_\_

Step 4: \_\_\_\_\_

Step 5: \_\_\_\_\_

Step 6: \_\_\_\_\_

Step 7: \_\_\_\_\_

Step 8: \_\_\_\_\_