Sorting

Saikrishna Arcot
(edits by M. Hudachek-Buswell)

March 10, 2017

Sorting

- Given a set of items, people can “easily” sort the items into ascending or descending order.

Sorting

- Given a set of items, people can “easily” sort the items into ascending or descending order.
- However, a computer cannot simply look at more than two items and easily sort them (whereas people can). Because of this, computers need to use an algorithm to sort items.

Sorting

- Given a set of items, people can "easily" sort the items into ascending or descending order.
- However, a computer cannot simply look at more than two items and easily sort them (whereas people can). Because of this, computers need to use an algorithm to sort items.
- There are multiple such algorithms for sorting items. Some are easier to implement, but take longer to run.

In-place Sorts

- An in-place sort is a sorting algorithm that doesn't copy over elements into another array/list. (Creating variables to store a fixed number of items is allowed.) In other words, regardless of the length of the array to be sorted, a fixed amount of (additional) space is used.

In-place Sorts

- An in-place sort is a sorting algorithm that doesn't copy over elements into another array/list. (Creating variables to store a fixed number of items is allowed.) In other words, regardless of the length of the array to be sorted, a fixed amount of (additional) space is used.
- An out-of-place sort is a sorting algorithm that does allocate a variable amount of additional space.

Stable Sorts

- A stable sort is a sort in which the order of duplicate items is preserved. For example, in the array, if there is a 4 near the starting of the array (let's call this 4a) and another 4 near the ending of the array (let's call this 4b), then after the array is sorted, 4a is guaranteed to be before 4b.

Stable Sorts

- A stable sort is a sort in which the order of duplicate items is preserved. For example, in the array, if there is a 4 near the starting of the array (let's call this 4a) and another 4 near the ending of the array (let's call this 4b), then after the array is sorted, 4a is guaranteed to be before 4b.
- An unstable sort is a sort in which the order of duplicate items may change.

Sorting Algorithms

- Six sorting algorithms will be covered:

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort
  - Insertion sort

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort
  - Insertion sort
  - Selection sort

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Merge sort

Sorting Algorithms

- Six sorting algorithms will be covered:
    - Bubble sort
    - Insertion sort
    - Selection sort
    - Merge sort
    - Quick sort

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Merge sort
  - Quick sort
  - Radix sort

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Merge sort
  - Quick sort
  - Radix sort
- All of the above algorithms except for the last one are known as **comparison sorts** because they directly compare two items; radix sort does not directly compare two items.

Sorting Algorithms

- Six sorting algorithms will be covered:
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Merge sort
  - Quick sort
  - Radix sort
- All of the above algorithms except for the last one are known as **comparison sorts** because they directly compare two items; radix sort does not directly compare two items.
- The first three sorting algorithms may also be referred to as $O(n^2)$ sorts because they run in $O(n^2)$ time for the average case.

Bubble Sort

- In bubble sort, compare the first two items. If they are in the wrong order, then swap them; if not, then keep them as-is. Then, move on to the next two items.

Bubble Sort

- In bubble sort, compare the first two items. If they are in the wrong order, then swap them; if not, then keep them as-is. Then, move on to the next two items.
- Repeat the above step until you get to the end of the array. Once you reach the end of the array, you have performed an iteration of bubble sort.

Bubble Sort

- In bubble sort, compare the first two items. If they are in the wrong order, then swap them; if not, then keep them as-is. Then, move on to the next two items.
- Repeat the above step until you get to the end of the array. Once you reach the end of the array, you have performed an iteration of bubble sort.
- At this point, the largest item in the array is in the last spot. Run the previous two steps on the array again, but don't include the last spot. Now, the two largest items are at the end of the array (in the correct spots). Repeat until the array is sorted.

Bubble Sort

- In bubble sort, compare the first two items. If they are in the wrong order, then swap them; if not, then keep them as-is. Then, move on to the next two items.
- Repeat the above step until you get to the end of the array. Once you reach the end of the array, you have performed an iteration of bubble sort.
- At this point, the largest item in the array is in the last spot. Run the previous two steps on the array again, but don't include the last spot. Now, the two largest items are at the end of the array (in the correct spots). Repeat until the array is sorted.
- If you do not make any swaps during an iteration, then this means that the array is sorted, and you can terminate early.

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69
6	31	72	38	43	86	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69
6	31	72	38	43	86	93	69
6	31	72	38	43	86	93	69

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69
6	31	72	38	43	86	93	69
6	31	72	38	43	86	93	69
6	31	72	38	43	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	38	72	43	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	38	72	43	86	69	93
6	31	38	43	72	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	38	72	43	86	69	93
6	31	38	43	72	86	69	93
6	31	38	43	72	86	69	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93

Current iteration

6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	72	38	43	86	69	93
6	31	38	72	43	86	69	93
6	31	38	43	72	86	69	93
6	31	38	43	72	86	69	93
6	31	38	43	72	69	86	93

Notice how the 86 and 93 weren't compared; this is because 93 is guaranteed to be the largest item in the array, and is guaranteed to be in the correct spot.

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Notice how the 69 and 86 weren't compared; this is because 86 and 93 are guaranteed to be the largest items in the array, and are guaranteed to be in the correct spots.

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Bubble Sort

Previous iterations

86	6	31	72	38	43	93	69
6	31	72	38	43	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Current iteration

6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93
6	31	38	43	72	69	86	93

Because no swaps were made in this iteration, the array must be sorted.

Bubble Sort

procedure BUBBLESORT(*array*)
    length $\leftarrow$ length of *array*
    $i \leftarrow 0$
    swapped $\leftarrow$ TRUE
    while $i < length - 1$ and *swappedis* TRUE **do**
        swapped $\leftarrow$ FALSE
        for $j \leftarrow 0, length - i - 1$ **do**
            if $array[j] > array[j+1]$ **then**
                swap $array[j]$ and $array[j+1]$
                swapped $\leftarrow$ TRUE
            end if
        end for
    end while
end procedure

Bubble Sort Performance

- In the best case, if the array is already sorted in the correct order, only one iteration of bubble sort will be done because no swaps will be made on that iteration. This means that $n - 1$ comparisons will be done and therefore the best case big-O of bubble sort is $O(n)$.

Bubble Sort Performance

- In the best case, if the array is already sorted in the correct order, only one iteration of bubble sort will be done because no swaps will be made on that iteration. This means that $n-1$ comparisons will be done and therefore the best case big-O of bubble sort is $O(n)$.
- In the worst case, if the array is sorted, but in the reverse order, then all $n-1$ iterations of bubble sort will need to be done because at least one swap will be made on each iteration. This means that $\frac{n(n-1)}{2}$ comparisons will be done and therefore the worst case big-O of bubble sort is $O(n^2)$.

Bubble Sort Performance

- In the best case, if the array is already sorted in the correct order, only one iteration of bubble sort will be done because no swaps will be made on that iteration. This means that $n-1$ comparisons will be done and therefore the best case big-O of bubble sort is $O(n)$.
- In the worst case, if the array is sorted, but in the reverse order, then all $n-1$ iterations of bubble sort will need to be done because at least one swap will be made on each iteration. This means that $\frac{n(n-1)}{2}$ comparisons will be done and therefore the worst case big-O of bubble sort is $O(n^2)$.
- In the average case, bubble sort runs in $O(n^2)$ time, because the actual number of comparisons that would be done is somewhere between $n-1$ and $\frac{n(n-1)}{2}$.

Bubble Sort Performance

- In the best case, if the array is already sorted in the correct order, only one iteration of bubble sort will be done because no swaps will be made on that iteration. This means that $n-1$ comparisons will be done and therefore the best case big-O of bubble sort is $O(n)$.
- In the worst case, if the array is sorted, but in the reverse order, then all $n-1$ iterations of bubble sort will need to be done because at least one swap will be made on each iteration. This means that $\frac{n(n-1)}{2}$ comparisons will be done and therefore the worst case big-O of bubble sort is $O(n^2)$.
- In the average case, bubble sort runs in $O(n^2)$ time, because the actual number of comparisons that would be done is somewhere between $n-1$ and $\frac{n(n-1)}{2}$.
- Bubble sort runs in-place and is a stable sort.

Insertion Sort

- In insertion sort, assume the first item is sorted. Then, take the second item, and “slide” it to the left so that it is correctly placed in the sorted portion of the array. The first two items are now considered sorted, and one iteration has been done.

Insertion Sort

- In insertion sort, assume the first item is sorted. Then, take the second item, and "slide" it to the left so that it is correctly placed in the sorted portion of the array. The first two items are now considered sorted, and one iteration has been done.
- Repeat with the third item and so on until the entire array is sorted.

Insertion Sort

- In insertion sort, assume the first item is sorted. Then, take the second item, and "slide" it to the left so that it is correctly placed in the sorted portion of the array. The first two items are now considered sorted, and one iteration has been done.
- Repeat with the third item and so on until the entire array is sorted.
- Unlike bubble sort, a fixed number of iterations are done for insertion sort.

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	72	86	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69

Current iteration

6	31	72	86	38	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69

Current iteration

6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69

Current iteration

6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69
6	31	38	72	86	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69

Current iteration

6	31	72	86	38	43	93	69
6	31	72	38	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	43	86	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	43	86	93	69
6	31	38	43	72	86	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	43	86	93	69
6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69
6	31	38	43	72	86	69	93

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69
6	31	38	43	72	86	69	93
6	31	38	43	72	69	86	93

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69
6	31	38	43	72	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	69	72	86	93

Insertion Sort

Previous iterations

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	72	86	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	72	86	93	69
6	31	38	43	72	86	93	69

Current iteration

6	31	38	43	72	86	93	69
6	31	38	43	72	86	69	93
6	31	38	43	72	69	86	93
6	31	38	43	69	72	86	93
6	31	38	43	69	72	86	93

Insertion Sort

procedure $\textsc{InsertionSort}(array)$
    $length \leftarrow$ length of $array$
    for $i \leftarrow 1, length - 1$ **do**
        $j \leftarrow i$
        while $j > 0$ and $array[j-1] > array[j]$ **do**
            swap $array[j-1]$ and $array[j]$
            $j \leftarrow j - 1$
        end while
    end for
end procedure

Insertion Sort Performance

- In the best case, if the array is already sorted in the correct order, then only one comparison will be done for each iteration of insertion sort (because the item will be already in the right slot). This means that $n - 1$ comparisons will be done and therefore the best case big-O of insertion sort is $O(n)$.

Insertion Sort Performance

- In the best case, if the array is already sorted in the correct order, then only one comparison will be done for each iteration of insertion sort (because the item will be already in the right slot). This means that $n - 1$ comparisons will be done and therefore the best case big-O of insertion sort is $O(n)$.
- In the worst case, if the array is sorted, but in the reverse order, then each iteration of insertion sort will do the maximum number of comparisons possible (because each item has to slide all the way to the left). Specifically, this means that $\frac{n(n-1)}{2}$ comparisons will be done and therefore the worst case big-O of insertion sort is $O(n^2)$.

Insertion Sort Performance

- In the best case, if the array is already sorted in the correct order, then only one comparison will be done for each iteration of insertion sort (because the item will be already in the right slot). This means that $n - 1$ comparisons will be done and therefore the best case big-O of insertion sort is $O(n)$.
- In the worst case, if the array is sorted, but in the reverse order, then each iteration of insertion sort will do the maximum number of comparisons possible (because each item has to slide all the way to the left). Specifically, this means that $\frac{n(n-1)}{2}$ comparisons will be done and therefore the worst case big-O of insertion sort is $O(n^2)$.
- In the average case, insertion sort runs in $O(n^2)$ time.

Insertion Sort Performance

- In the best case, if the array is already sorted in the correct order, then only one comparison will be done for each iteration of insertion sort (because the item will be already in the right slot). This means that $n-1$ comparisons will be done and therefore the best case big-O of insertion sort is $O(n)$.
- In the worst case, if the array is sorted, but in the reverse order, then each iteration of insertion sort will do the maximum number of comparisons possible (because each item has to slide all the way to the left). Specifically, this means that $\frac{n(n-1)}{2}$ comparisons will be done and therefore the worst case big-O of insertion sort is $O(n^2)$.
- In the average case, insertion sort runs in $O(n^2)$ time.
- Insertion sort runs in-place and is a stable sort.

Selection Sort

- In selection sort, search the entire array for the smallest item (start by assuming the first item you see *is* the smallest item). Swap that item with the first item.

Selection Sort

- In selection sort, search the entire array for the smallest item (start by assuming the first item you see *is* the smallest item). Swap that item with the first item.
- Then, search the entire array (excluding the first item) for the next smallest item. Swap that item with the second item.

Selection Sort

- In selection sort, search the entire array for the smallest item (start by assuming the first item you see *is* the smallest item). Swap that item with the first item.
- Then, search the entire array (excluding the first item) for the next smallest item. Swap that item with the second item.
- Repeat until the entire array is sorted.

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69

Current iteration

86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69

Current iteration

6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69

Current iteration

6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69

Current iteration

6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Current iteration

6	31	38	43	86	72	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Current iteration

6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Current iteration

6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Current iteration

6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Current iteration

6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69

Current iteration

6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	86	93

Selection Sort

Previous iterations

86	6	31	72	38	43	93	69
6	86	31	72	38	43	93	69
6	31	86	72	38	43	93	69
6	31	38	72	86	43	93	69
6	31	38	43	86	72	93	69
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86

Current iteration

6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	93	86
6	31	38	43	69	72	86	93
6	31	38	43	69	72	86	93

Selection Sort

procedure $\textsc{SelectionSort}(array)$
    $length \leftarrow$ length of $array$
    for $i \leftarrow 0, length$ **do**
        $minIndex \leftarrow i$
        for $j \leftarrow i, length - 1$ **do**
            if $array[j] < array[minIndex]$ **then**
                $minIndex \leftarrow j$
            end if
        end for
        swap $array[minIndex]$ and $array[i]$
    end for
end procedure

Selection Sort Performance

- Selection sort does the same number of comparisons in all cases ($\frac{n(n-1)}{2}$), since there is no early termination of any kind. The big-O of selection sort is $O(n^2)$.

Selection Sort Performance

- Selection sort does the same number of comparisons in all cases ($\frac{n(n-1)}{2}$), since there is no early termination of any kind. The big-O of selection sort is $O(n^2)$.
- Selection sort runs in-place, but is **not** a stable sort.

Merge and Quick Sorts

- Unlike the previous sorts that have been covered, merge and quick sorts are recursive sorts rather than iterative sorts.

Merge and Quick Sorts

- Unlike the previous sorts that have been covered, merge and quick sorts are recursive sorts rather than iterative sorts.
- To be more specific, these two sorts are also known as divide-and-conquer sorts, which means that the array is divided into two (or more) parts, and those parts are then sorted. If needed, at the end, the sorted parts are merged together.

Merge and Quick Sorts

- Unlike the previous sorts that have been covered, merge and quick sorts are recursive sorts rather than iterative sorts.
- To be more specific, these two sorts are also known as divide-and-conquer sorts, which means that the array is divided into two (or more) parts, and those parts are then sorted. If needed, at the end, the sorted parts are merged together.
- One major advantage of this is that portions of the sorting algorithm can be done in parallel (i.e. in multiple threads).

Merge and Quick Sorts

- Unlike the previous sorts that have been covered, merge and quick sorts are recursive sorts rather than iterative sorts.
- To be more specific, these two sorts are also known as divide-and-conquer sorts, which means that the array is divided into two (or more) parts, and those parts are then sorted. If needed, at the end, the sorted parts are merged together.
- One major advantage of this is that portions of the sorting algorithm can be done in parallel (i.e. in multiple threads).
- In both cases, remember that an array of length 1 is always sorted.

Merge Sort

- In merge sort, divide the array into 2 equal parts; one part contains the elements from the left half of the array while the other part contains the elements from the right half of the array. (If there is an odd number of elements, the middle element will go to either the first part or the second part.)

Merge Sort

- In merge sort, divide the array into 2 equal parts; one part contains the elements from the left half of the array while the other part contains the elements from the right half of the array. (If there is an odd number of elements, the middle element will go to either the first part or the second part.)
- Then, perform merge sort on each half of the array. After this is done, each part should be sorted.

Merge Sort

- Finally, merge the two parts together. To do this, have a marker on the first item in each part. Take the smaller of the two items and add that into the larger (merged) array, and move that marker forward. Repeat until all of the items have been added into the merged array.

Merge Sort

- Finally, merge the two parts together. To do this, have a marker on the first item in each part. Take the smaller of the two items and add that into the larger (merged) array, and move that marker forward. Repeat until all of the items have been added into the merged array.
- Note that while merging the two parts together, if all of the items in one of the parts have been added into the larger array, you can directly copy over the remaining items from the other part into the larger array.

Merge Sort

86	6	31	72	38	43	93

Merge Sort

86	6	31	72	38	43	93

86	6	31

72	38	43	93

Merge Sort

86	6	31	72	38	43	93

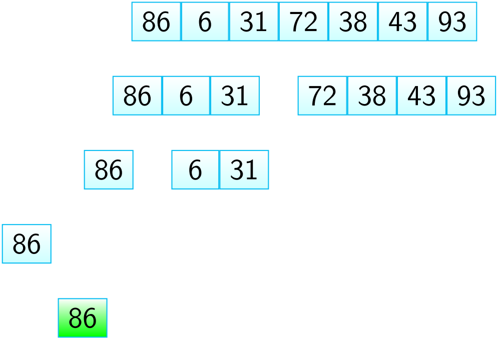
86	6	31

72	38	43	93

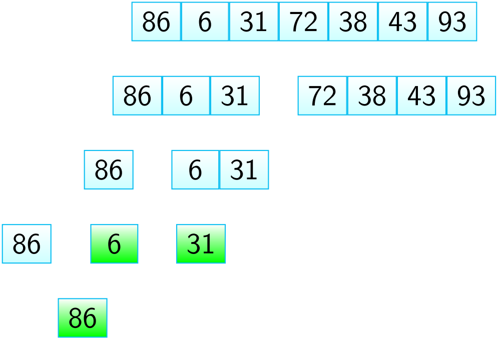
86

6	31

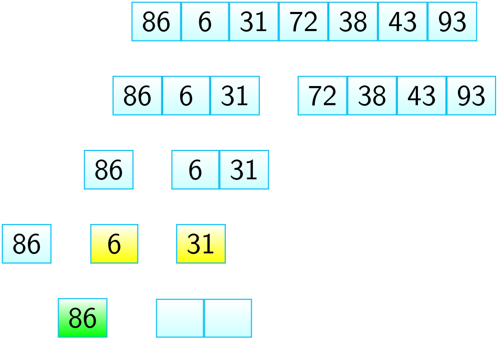
Merge Sort



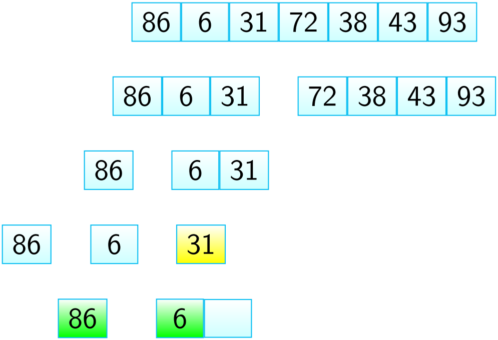
Merge Sort



Merge Sort



Merge Sort



Merge Sort

86 6 31 72 38 43 93

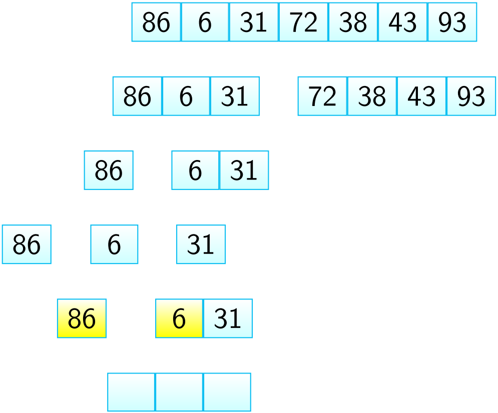
86 6 31 72 38 43 93

86 6 31

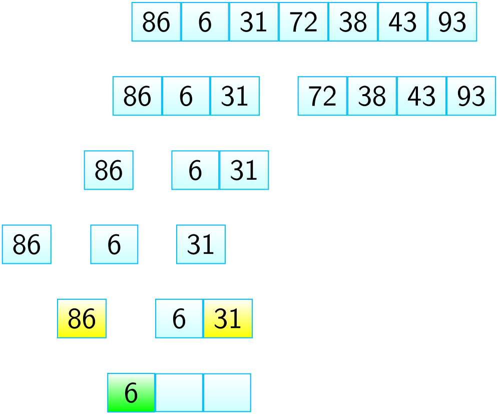
86 6 31

86 6 31

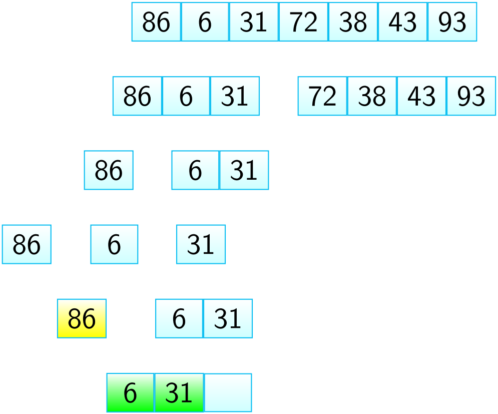
Merge Sort



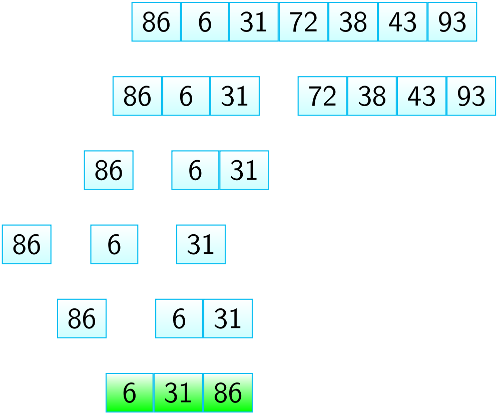
Merge Sort



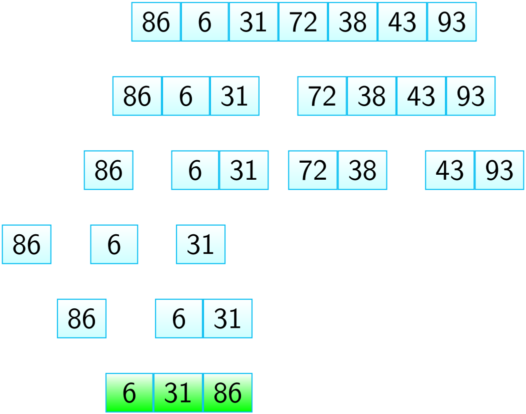
Merge Sort



Merge Sort



Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
86
6 31
6 31 86

Merge Sort

86 6 31 72 38 43 93

86 6 31 | 72 38 43 93

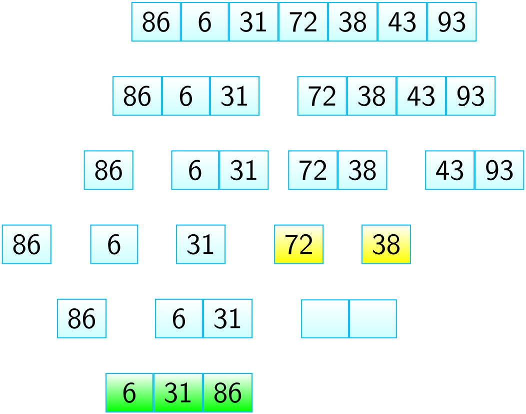
86 | 6 31 | 72 38 | 43 93

86 | 6 | 31 | 72 | 38

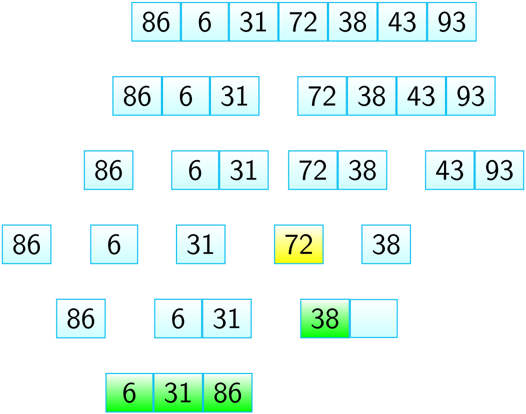
86 | 6 31

6 31 86

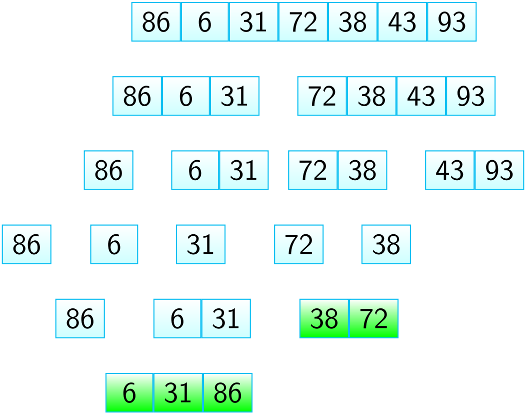
Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
86
6 31
6 31 86

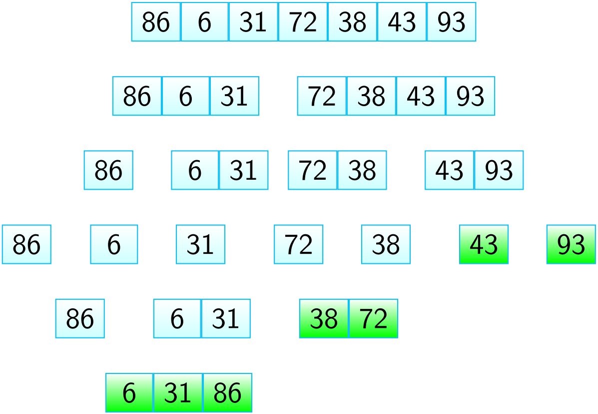
Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
86
6 31
38
6 31 86

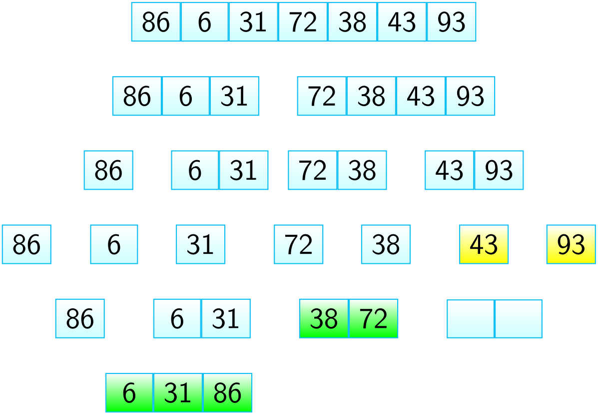
Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
86
6 31
38 72
6 31 86

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
6 31 86

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
6 31 86

Merge Sort

86 6 31 72 38 43 93

86 6 31   72 38 43 93

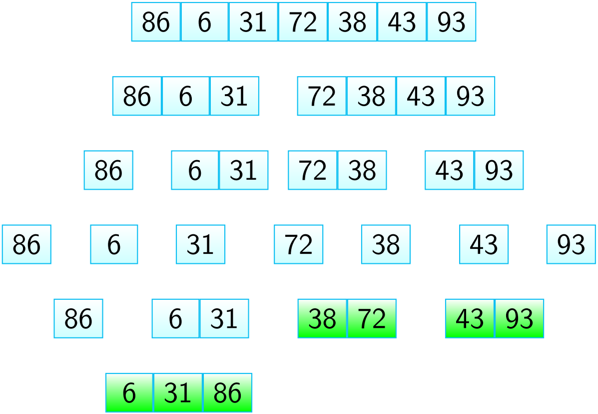
86   6 31   72 38   43 93

86   6   31   72   38   43   93

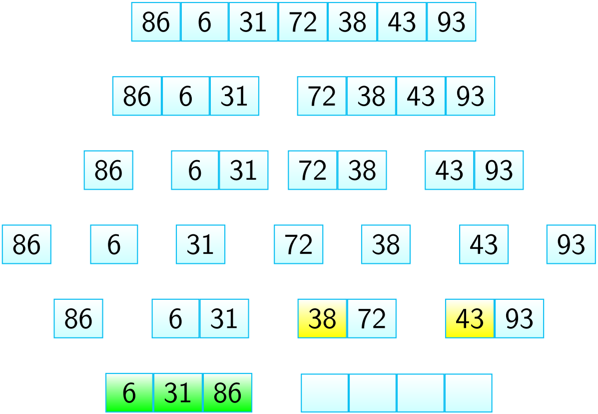
86   6 31   38 72   43

6 31 86

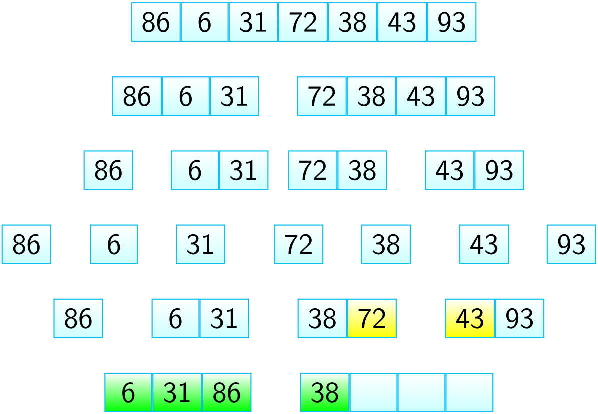
Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86

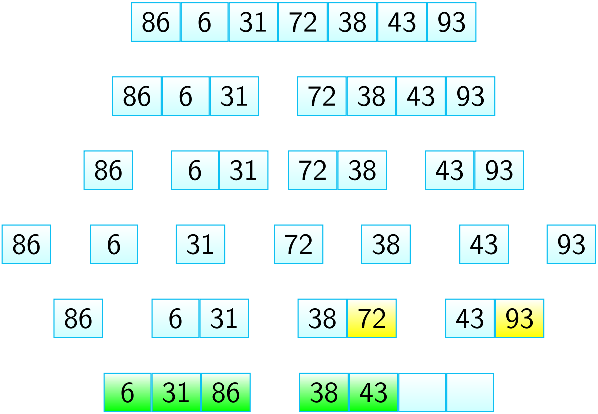
Merge Sort


86 6 31 72 38 43 93
86 6 31 72 38 43 93
86 6 31 72 38 43 93
86 6 31 72 38 43 93
86 6 31 38 72 43 93
6 31 86

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38 43

Merge Sort

86 6 31 72 38 43 93

86 6 31   72 38 43 93

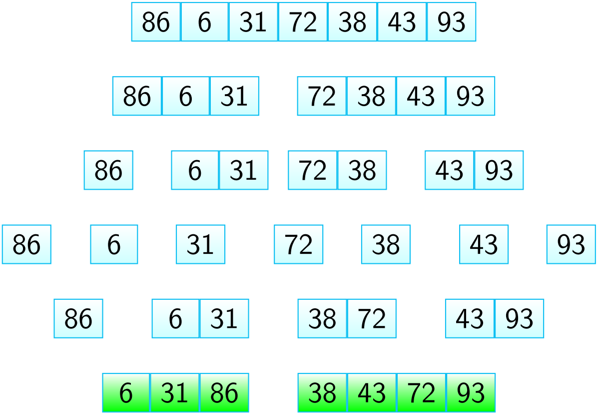
86   6 31   72 38   43 93

86   6   31   72   38   43   93

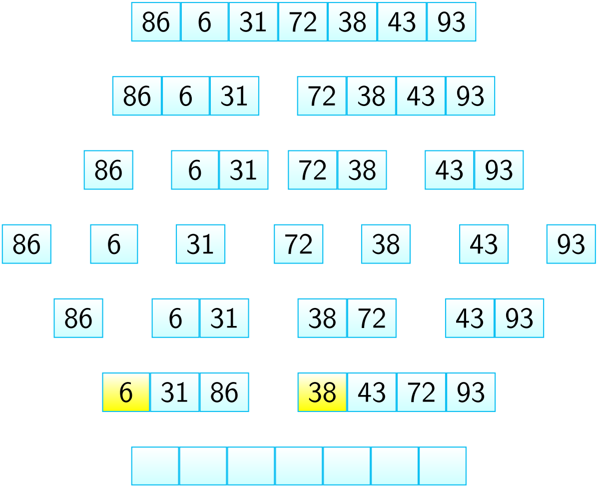
86   6 31   38 72   43 93

6 31 86   38 43 72

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38 43 72 93

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38 43 72 93

Merge Sort

86 6 31 72 38 43 93

86 6 31 | 72 38 43 93

86 | 6 31 | 72 38 | 43 93

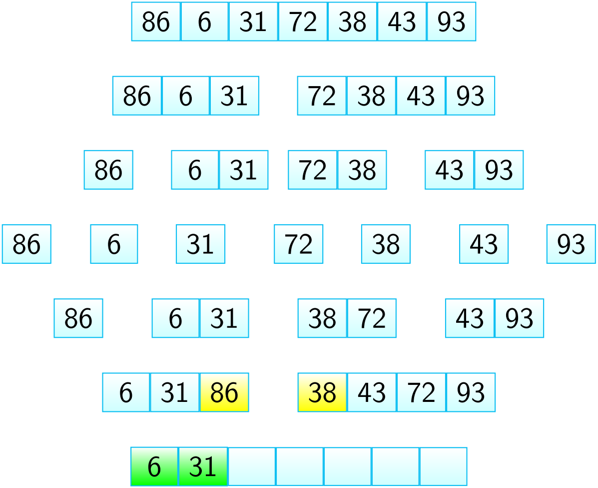
86 | 6 | 31 | 72 | 38 | 43 | 93

86 | 6 31 | 38 72 | 43 93

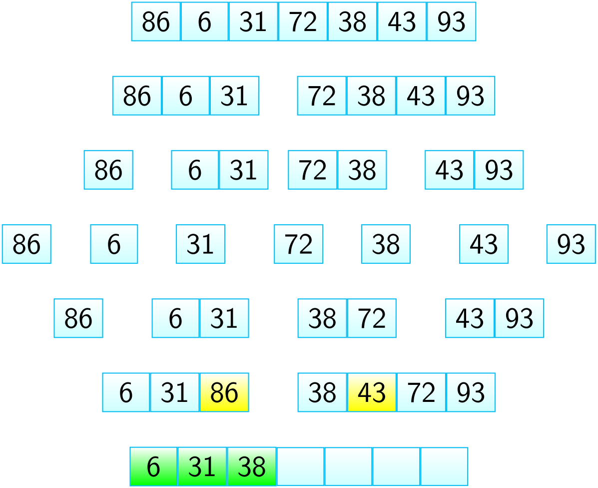
6 31 86 | 38 43 72 93

6

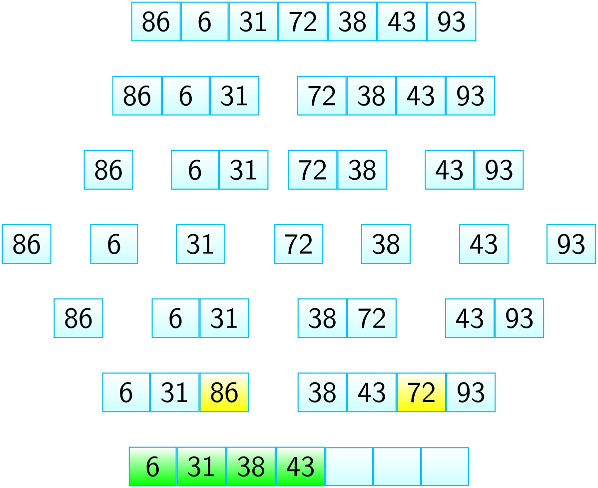
Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38 43 72 93
6 31

Merge Sort


86 6 31 72 38 43 93
86 6 31 72 38 43 93
86 6 31 72 38 43 93
86 6 31 72 38 43 93
86 6 31 38 72 43 93
6 31 86 38 43 72 93
6 31 38

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38 43 72 93
6 31 38 43

Merge Sort

86 6 31 72 38 43 93

86 6 31   72 38 43 93

86   6 31   72 38   43 93

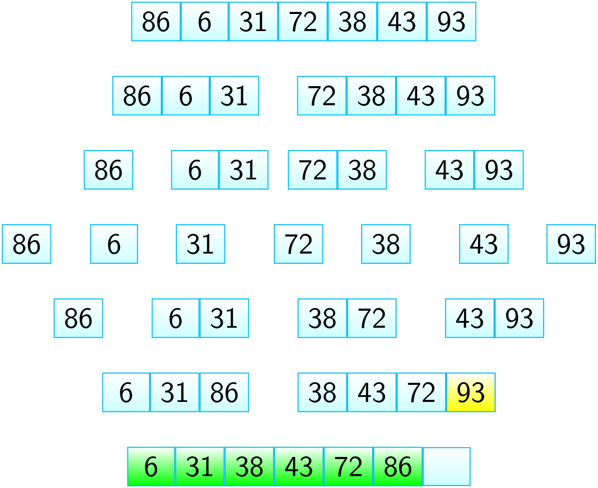
86   6   31   72   38   43   93

86   6 31   38 72   43 93

6 31 86   38 43 72 93

6 31 38 43 72

Merge Sort


86 6 31 72 38 43 93
86 6 31
72 38 43 93
86
6 31
72 38
43 93
86
6
31
72
38
43
93
86
6 31
38 72
43 93
6 31 86
38 43 72 93
6 31 38 43 72 86

Merge Sort

86 6 31 72 38 43 93

86 6 31 | 72 38 43 93

86 | 6 31 | 72 38 | 43 93

86 | 6 | 31 | 72 | 38 | 43 | 93

86 | 6 31 | 38 72 | 43 93

6 31 86 | 38 43 72 93

6 31 38 43 72 86 93

Merge Sort

procedure $\textsc{MergeSort}(\mathit{array})$
  $\mathit{length} \leftarrow$ length of array
  $\mathit{midIndex} \leftarrow \mathit{length}/2$
  $\mathit{leftArray} \leftarrow \mathit{array}[0..\mathit{midIndex} - 1]$
  $\mathit{rightArray} \leftarrow \mathit{array}[\mathit{midIndex}..\mathit{length} - 1]$
  $\textsc{MergeSort}(\mathit{leftArray})$
  $\textsc{MergeSort}(\mathit{rightArray})$
  $\mathit{leftIndex} \leftarrow 0$
  $\mathit{rightIndex} \leftarrow 0$
  $\mathit{currentIndex} \leftarrow 0$

Merge Sort

while *leftIndex* $<$ *midIndex* and *rightIndex* $<$ *length* $-$ *midIndex* **do**
  if $leftArray[leftIndex] < rightArray[rightIndex]$ **then**
    $array[currentIndex] \leftarrow leftArray[leftIndex]$
    $leftIndex \leftarrow leftIndex + 1$
  else
    $array[currentIndex] \leftarrow rightArray[rightIndex]$
    $rightIndex \leftarrow rightIndex + 1$
  end if
  $currentIndex \leftarrow currentIndex + 1$
end while

Merge Sort

while $leftIndex < midIndex$ **do**
  $array[currentIndex] \leftarrow leftArray[leftIndex]$
  $leftIndex \leftarrow leftIndex + 1$
  $currentIndex \leftarrow currentIndex + 1$
end while
while $rightIndex < length - midIndex$ **do**
  $array[currentIndex] \leftarrow rightArray[rightIndex]$
  $rightIndex \leftarrow rightIndex + 1$
  $currentIndex \leftarrow currentIndex + 1$
end while
end procedure

Merge Sort Performance

- In the worst case (when the array is sorted in reverse order, for example), merge sort will run in $O(n \log n)$ time.

Merge Sort Performance

- In the worst case (when the array is sorted in reverse order, for example), merge sort will run in $O(n \log n)$ time.
- The best case for merge sort is when, for example, you are merging two arrays, and all of the items in one array are smaller than all of the items in the other array. In this case, you will make m comparisons, where m is the length of the smaller array. However, even in this case, the big-O of merge sort is $O(n \log n)$.

Merge Sort Performance

- In the worst case (when the array is sorted in reverse order, for example), merge sort will run in $O(n \log n)$ time.
- The best case for merge sort is when, for example, you are merging two arrays, and all of the items in one array are smaller than all of the items in the other array. In this case, you will make m comparisons, where m is the length of the smaller array. However, even in this case, the big-O of merge sort is $O(n \log n)$.
- Merge sort is out-of-place, but it is stable.

Quick Sort

- In quick sort, choose an item at random to be the pivot.

Quick Sort

- In quick sort, choose an item at random to be the pivot.
- Swap the pivot with the first item.

Quick Sort

- In quick sort, choose an item at random to be the pivot.
- Swap the pivot with the first item.
- Have a left “marker” that starts with the second item (the item after the pivot), and a right “marker” that starts at the last item.

Quick Sort

- In quick sort, choose an item at random to be the pivot.
- Swap the pivot with the first item.
- Have a left "marker" that starts with the second item (the item after the pivot), and a right "marker" that starts at the last item.
- If the item pointed to by the left marker is smaller than the pivot, move the marker one item to the right. Repeat this step until the marker points to an item that is larger than the pivot or goes *beyond* the right marker (they cross over). (If the item and the pivot are equal, then either can be done.)

Quick Sort

- If the item pointed to by the right marker is larger than the pivot, move the marker one item to the left. Repeat this step until the marker points to an item that is smaller than the pivot or goes *beyond* the left marker. (If the item and the pivot are equal, then either can be done.)

Quick Sort

- If the item pointed to by the right marker is larger than the pivot, move the marker one item to the left. Repeat this step until the marker points to an item that is smaller than the pivot or goes *beyond* the left marker. (If the item and the pivot are equal, then either can be done.)
- After the markers cross over, swap the pivot with the right marker (note that the right marker is now **to the left of the left marker**).

Quick Sort

- If the item pointed to by the right marker is larger than the pivot, move the marker one item to the left. Repeat this step until the marker points to an item that is smaller than the pivot or goes *beyond* the left marker. (If the item and the pivot are equal, then either can be done.)
- After the markers cross over, swap the pivot with the right marker (note that the right marker is now **to the left of the left marker**).
- The pivot is now in the right place within the final sorted array. All items to the left of the pivot (if there are any) are smaller than the pivot, and all items to the right of the pivot (if there are any) are larger than the pivot. Perform quicksort on the smaller items and on the larger items.

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

86	6	31	72	38	43	93	69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

38	6	31	72	86	43	93	69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

	L						
38	6	31	72	86	43	93	69
							R

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

		L					
38	6	31	72	86	43	93	69
							R

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

			L				
38	6	31	72	86	43	93	69
							R

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

L

38	6	31	72	86	43	93	69

R

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

			L				
38	6	31	72	86	43	93	69
					R		

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

			L				
38	6	31	72	86	43	93	69
				R			

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

			L				
38	6	31	72	86	43	93	69
			R				

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

			L				
38	6	31	72	86	43	93	69
		R					

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6	38	72	86	43	93	69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6

38

72	86	43	93	69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6

38

72	86	43	93	69

31	6

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6

38

72	86	43	93	69

31	6

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6

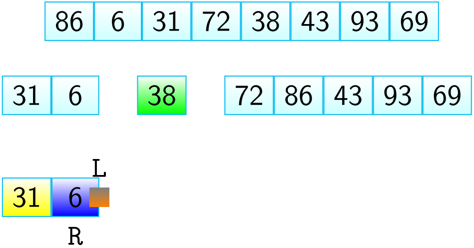
38

72	86	43	93	69

	L
31	6
	R

Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6

38

72	86	43	93	69

6	31

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6

38

72	86	43	93	69

6

31

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6	38	72	86	43	93	69	

6	31	38

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31 6   38   72 86 43 93 69

6   31   38   72 86 43 93 69

Quick Sort

(Randomly-selected pivots are in yellow.)

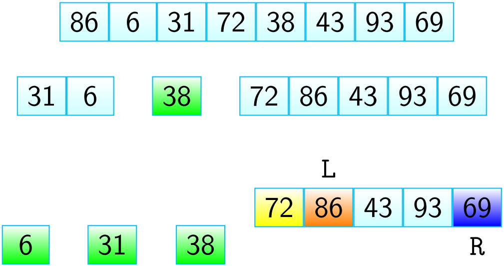
86	6	31	72	38	43	93	69

31 6   38   72 86 43 93 69

6   31   38   72 86 43 93 69

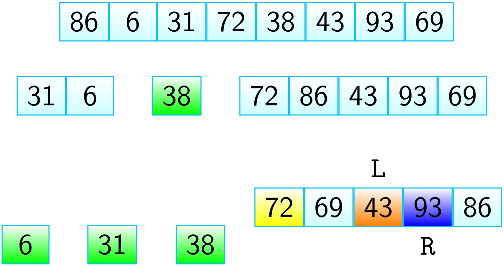
Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6	38	72	86	43	93	69

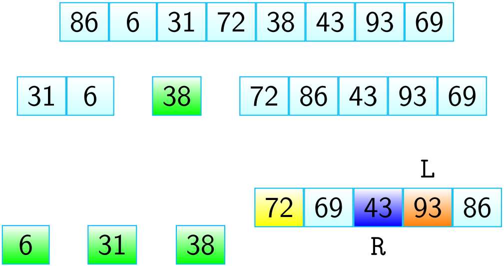
L

6	31	38	72	69	43	93	86

R

Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31 6   38   72 86 43 93 69

6   31   38   43 69 72 93 86

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6		38		72	86	43	93	69

6		31		38		43	69		72		93	86

Quick Sort

(Randomly-selected pivots are in yellow.)

86 6 31 72 38 43 93 69

31 6 | 38 | 72 86 43 93 69

6 | 31 | 38 | 43 69 | 72 | 93 86

6 | 31 | 38

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31 6   38   72 86 43 93 69

6   31   38   43 69   72   93 86

6   31   38   43 69

Quick Sort

(Randomly-selected pivots are in yellow.)

86 6 31 72 38 43 93 69

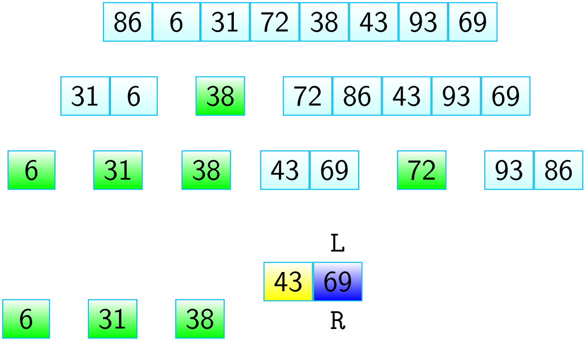
31 6   38   72 86 43 93 69

6   31   38   43 69   72   93 86

6   31   38   43 69

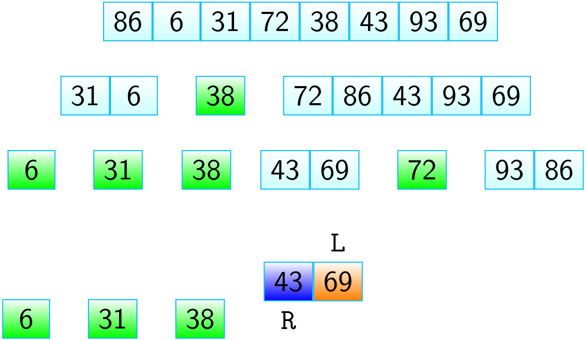
Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6		38		72	86	43	93	69

6		31		38		43	69		72		93	86

6		31		38		43	69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31 6   38   72 86 43 93 69

6   31   38   43 69   72   93 86

6   31   38   43   69

Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6		38		72	86	43	93	69

6		31		38		43	69		72		93	86

6		31		38		43		69		72

Quick Sort

(Randomly-selected pivots are in yellow.)

86 | 6 | 31 | 72 | 38 | 43 | 93 | 69

31 | 6   **38**   72 | 86 | 43 | 93 | 69

6   **31**   **38**   43 | 69   **72**   93 | 86

6   **31**   **38**   **43**   **69**   **72**   93 | 86

Quick Sort

(Randomly-selected pivots are in yellow.)

86 6 31 72 38 43 93 69

31 6 38 72 86 43 93 69

6 31 38 43 69 72 93 86

6 31 38 43 69 72 93 86

Quick Sort

(Randomly-selected pivots are in yellow.)

86 6 31 72 38 43 93 69

31 6   38   72 86 43 93 69

6   31   38   43 69   72   93 86

L

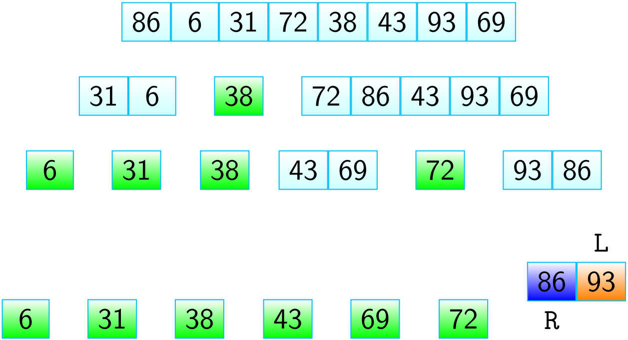
86 93

R

6   31   38   43   69   72

Quick Sort

(Randomly-selected pivots are in yellow.)



Quick Sort

(Randomly-selected pivots are in yellow.)

86	6	31	72	38	43	93	69

31	6	38	72	86	43	93	69

6	31	38	43	69	72	93	86

6	31	38	43	69	72	86	93

Quick Sort

(Randomly-selected pivots are in yellow.)

86 6 31 72 38 43 93 69

31 6 38 72 86 43 93 69

6 31 38 43 69 72 93 86

6 31 38 43 69 72 86 93

Quick Sort

(Randomly-selected pivots are in yellow.)

86 | 6 | 31 | 72 | 38 | 43 | 93 | 69

31 | 6   38   72 | 86 | 43 | 93 | 69

6   31   38   43 | 69   72   93 | 86

6   31   38   43   69   72   86   93

6 | 31 | 38 | 43 | 72 | 86 | 93

Quick Sort

procedure $\textsc{QuickSort}(array)$

  $\textsc{QuickSort}(array, 0, \text{length of } array)$

end procedure

procedure $\textsc{QuickSort}(array, left, right)$

  $pivotIndex \leftarrow$ randomly-selected index within bounds of region being sorted

  $pivot \leftarrow array[pivotIndex]$

  Swap $array[left]$ and $array[pivotIndex]$

  $leftIndex \leftarrow left + 1$

  $rightIndex \leftarrow right - 1$

Quick Sort

while $leftIndex < rightIndex$ **do**
while $leftIndex < rightIndex$ and $array[leftIndex] <= pivot$ **do**
$leftIndex \leftarrow leftIndex + 1$
end while
while $leftIndex < rightIndex$ and $array[rightIndex] >= pivot$ **do**
$rightIndex \leftarrow rightIndex - 1$
end while

Quick Sort

if *leftIndex* $<$ *rightIndex* **then**
  Swap *array*[*leftIndex*] and *array*[*rightIndex*]
  $leftIndex \leftarrow leftIndex + 1$
  $rightIndex \leftarrow rightIndex - 1$
end if
end while
Swap *pivot* and *array*[*rightIndex*]
$\textsc{QuickSort}(array, left, rightIndex - 1)$
$\textsc{QuickSort}(array, rightIndex + 1, right)$
end procedure

Quick Sort Performance

- In the best case, the pivot that is chosen each time will divide the array in half. This results in $O(n \log n)$ performance.

Quick Sort Performance

- In the best case, the pivot that is chosen each time will divide the array in half. This results in $O(n \log n)$ performance.
- In the worst case, the pivot that is chosen each time will be either the largest or smallest value. This means that all of the other values will be smaller or larger than the pivot. This effectively turns quicksort into something like selection sort, and results in $O(n^2)$ performance.

Quick Sort Performance

- In the best case, the pivot that is chosen each time will divide the array in half. This results in $O(n \log n)$ performance.
- In the worst case, the pivot that is chosen each time will be either the largest or smallest value. This means that all of the other values will be smaller or larger than the pivot. This effectively turns quicksort into something like selection sort, and results in $O(n^2)$ performance.
- This quick sort is in-place, but it is **not** stable. Quick sort can also be done such that it is stable, but it must be out-of-place.

Radix Sort

- Radix sort is different than all of the previous sorting algorithms in that no comparisons are actually done.

Radix Sort

- Radix sort is different than all of the previous sorting algorithms in that no comparisons are actually done.
- However, radix sort can only be done on numbers of some base (base 10, base 8, base 16, base 256 (this lets you perform radix sort on ASCII letters), etc.)

Radix Sort

- Radix sort is different than all of the previous sorting algorithms in that no comparisons are actually done.
- However, radix sort can only be done on numbers of some base (base 10, base 8, base 16, base 256 (this lets you perform radix sort on ASCII letters), etc.)
- There are two variants of radix sort:

Radix Sort

- Radix sort is different than all of the previous sorting algorithms in that no comparisons are actually done.
- However, radix sort can only be done on numbers of some base (base 10, base 8, base 16, base 256 (this lets you perform radix sort on ASCII letters), etc.)
- There are two variants of radix sort:
  - One variant starts by looking at the least significant digit and works upwards. This is called Least Significant Digit (LSD) Radix Sort.

Radix Sort

- Radix sort is different than all of the previous sorting algorithms in that no comparisons are actually done.
- However, radix sort can only be done on numbers of some base (base 10, base 8, base 16, base 256 (this lets you perform radix sort on ASCII letters), etc.)
- There are two variants of radix sort:
  - One variant starts by looking at the least significant digit and works upwards. This is called Least Significant Digit (LSD) Radix Sort.
  - The other variant starts by looking at the most significant digit and works downwards. This is called Most Significant Digit (MSD) Radix Sort.

LSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 “buckets”, and label them from 0 to 9. Treat each bucket as a queue.

LSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 "buckets", and label them from 0 to 9. Treat each bucket as a queue.
- For each number, take the first digit (least significant digit), and add the number into the appropriate bucket. (For example, if the number is 27, then the first digit is 7, and add the number into bucket 7.)

LSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 "buckets", and label them from 0 to 9. Treat each bucket as a queue.
- For each number, take the first digit (least significant digit), and add the number into the appropriate bucket. (For example, if the number is 27, then the first digit is 7, and add the number into bucket 7.)
- After all of the numbers have been added, remove all of the numbers one at a time, starting from bucket 0.

LSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 "buckets", and label them from 0 to 9. Treat each bucket as a queue.
- For each number, take the first digit (least significant digit), and add the number into the appropriate bucket. (For example, if the number is 27, then the first digit is 7, and add the number into bucket 7.)
- After all of the numbers have been added, remove all of the numbers one at a time, starting from bucket 0.
- Repeat this process for each digit in the *longest* (not necessarily the *largest*) number. (In other words, if the longest number has 4 digits, then you would repeat this process 3 more times.)

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
						86			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
						86			
						6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31					86			
						6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72				86			
						6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72				86		38	
						6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	
						6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31							

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31							

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72						

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72						

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43					

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93				

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93				

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93				

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93				

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86			

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6		

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6		

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6		

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
	31	72	43			86		38	69
			93			6			

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
			31						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
			31				72		

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
			31	43			72		

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
			31	43			72		93

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
			31	43			72	86	93

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
6			31	43			72	86	93

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
6			31	43			72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6							

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6							

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6							

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6							

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31						

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38					

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38					

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43				

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43				

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43				

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69			

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69			

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69	72		

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69	72		

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69	72	86	

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69	72	86	

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69	72	86	93

0	1	2	3	4	5	6	7	8	9

LSD Radix Sort

86	6	31	72	38	43	93	69

31	72	43	93	86	6	38	69

6	31	38	43	69	72	86	93

0	1	2	3	4	5	6	7	8	9

LSD Radix Sort

```
procedure LSDRadixSort(array)
    buckets ← list of 10 lists
    iterations ← length of longest number
    length ← length of array
    for i ← 1, iterations do
        for j ← 0, length − 1 do
            bucket ← i^th digit of array[j]
            add array[j] to the end of buckets[bucket]
        end for
        index ← 0
```

LSD Radix Sort

for $bucket \leftarrow 0, 10$ **do**
  while $buckets[bucket]$ isn't empty **do**
    $array[index] \leftarrow$ remove first item from $buckets[bucket]$
    $index \leftarrow index + 1$
  end while
end for
end for
end procedure

LSD Radix Sort Performance

- Unlike the previous sorting algorithms, the efficiency of radix sort depends on the number of items in the array (n) *and* on the length of the longest number (k).

LSD Radix Sort Performance

- Unlike the previous sorting algorithms, the efficiency of radix sort depends on the number of items in the array (n) *and* on the length of the longest number (k).
- In the best, worst, and average case, LSD radix sort runs in $O(kn)$ time.

LSD Radix Sort Performance

- Unlike the previous sorting algorithms, the efficiency of radix sort depends on the number of items in the array (n) *and* on the length of the longest number (k).
- In the best, worst, and average case, LSD radix sort runs in $O(kn)$ time.
- LSD radix sort is stable, but not in-place.

MSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 “buckets”, and label them from 0 to 9. (Note that the buckets here aren’t exactly queues.)

MSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 “buckets”, and label them from 0 to 9. (Note that the buckets here aren’t exactly queues.)
- Find the length of the longest number. This will be the digit you start with.

MSD Radix Sort

- (Assuming radix sort is done in base 10) Create 10 “buckets”, and label them from 0 to 9. (Note that the buckets here aren’t exactly queues.)
- Find the length of the longest number. This will be the digit you start with.
- For each number, take the digit in the position you found in the previous step, and add the number into the appropriate bucket. (For example, if the longest number has 4 digits, then you would get the 4th digit (most significant digit) of each number.)

MSD Radix Sort

- After all of the numbers have been added into the buckets, for each bucket, if there are two or more numbers in the bucket, run MSD radix sort again, but use the next smaller/lower digit. After this is done, the numbers in each bucket should be sorted in ascending order.

MSD Radix Sort

- After all of the numbers have been added into the buckets, for each bucket, if there are two or more numbers in the bucket, run MSD radix sort again, but use the next smaller/lower digit. After this is done, the numbers in each bucket should be sorted in ascending order.
- Starting with the first bucket, remove the first number of each bucket until the bucket is empty. The numbers should now be sorted.

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
								86	

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6								86	

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31					86	

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31				72	86	

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31				72	86	
			38						

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43			72	86	
			38						

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43			72	86	93
			38						

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	38

0	1	2	3	4	5	6	7	8	9

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	38

0	1	2	3	4	5	6	7	8	9

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	38

0	1	2	3	4	5	6	7	8	9
	31								

6							

MSD Radix Sort

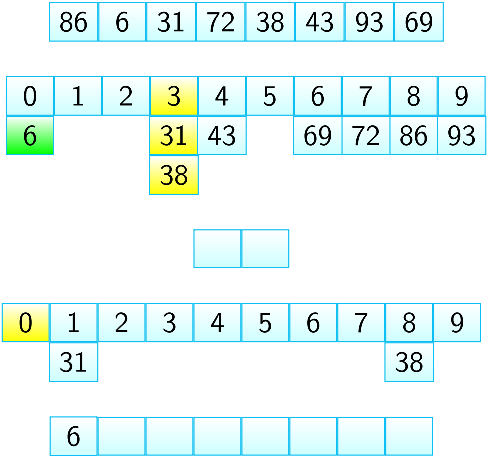
86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort



MSD Radix Sort

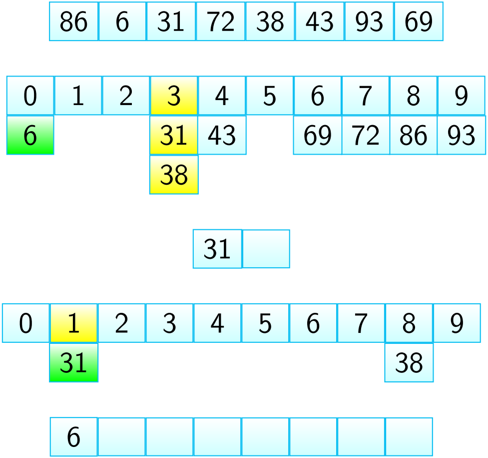
86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort



MSD Radix Sort

86	6	31	72	38	43	93	69

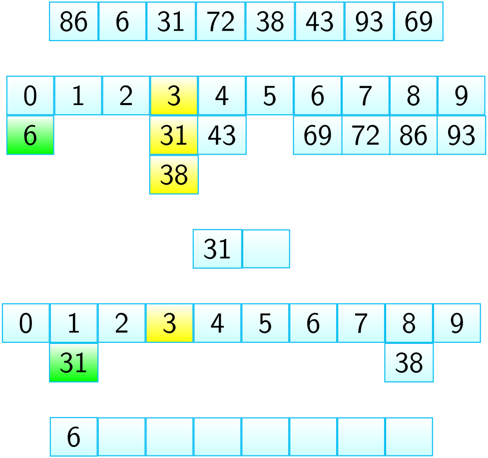
0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort



MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

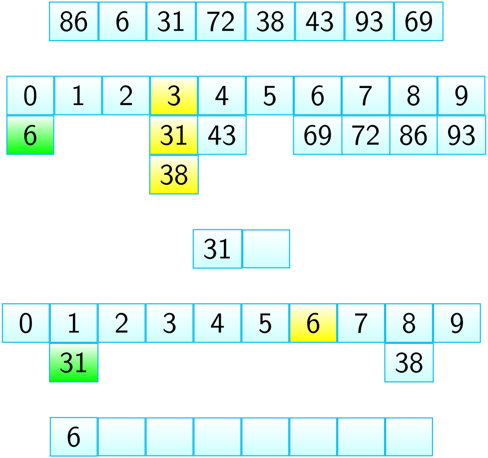
0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort



MSD Radix Sort

86	6	31	72	38	43	93	69

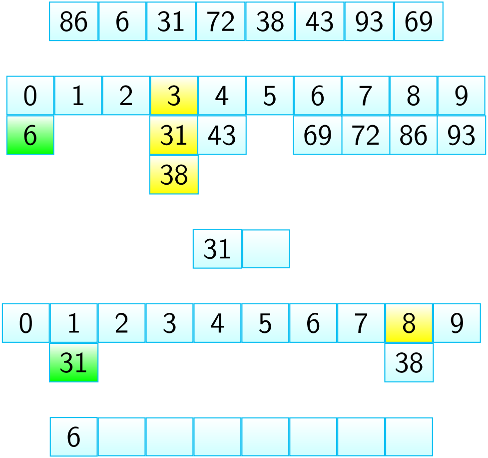
0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort



MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	38

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	38

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

31	38

0	1	2	3	4	5	6	7	8	9
	31							38	

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6							

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31						

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38					

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38					

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43				

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43				

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43				

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69			

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69			

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69	72		

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69	72		

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69	72	86	

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69	72	86	

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69	72	86	93

MSD Radix Sort

86	6	31	72	38	43	93	69

0	1	2	3	4	5	6	7	8	9
6			31	43		69	72	86	93
			38						

6	31	38	43	69	72	86	93

MSD Radix Sort

procedure $\textsc{MSDRadixSort}(array)$
    $maxLength \leftarrow$ length of longest number
    $\textsc{MSDRadixSort}(array, maxLength)$
end procedure
procedure $\textsc{MSDRadixSort}(array, i)$
    $buckets \leftarrow$ list of 10 lists
    $length \leftarrow$ length of array
    for $j \leftarrow 0, length - 1$ **do**
        $bucket \leftarrow i^{th}$ digit of $array[j]$
        add $array[j]$ to $buckets[bucket]$
    end for

MSD Radix Sort

$index \leftarrow 0$
for $bucket \leftarrow 0, 9$ **do**
    if number of items in $buckets[bucket] > 1$ and $i > 1$ **then**
        $\textsc{MSDRadixSort}(buckets[bucket], i - 1)$
    end if
    while $buckets[bucket]$ isn't empty **do**
        $array[index] \leftarrow$ remove first item from $buckets[bucket]$
        $index \leftarrow index + 1$
    end while
end for
end procedure

MSD Radix Sort Performance

- In the best, worst, and average case, MSD radix sort runs in $O(kn)$ time, where k is the length of the longest number.

MSD Radix Sort Performance

- In the best, worst, and average case, MSD radix sort runs in $O(kn)$ time, where k is the length of the longest number.
- However, note that, when “long” numbers are used, and the most significant digits are different, then MSD radix sort will likely take fewer iterations to finish than LSD radix sort, possibly just one (if there are few numbers).

MSD Radix Sort Performance

- In the best, worst, and average case, MSD radix sort runs in $O(kn)$ time, where k is the length of the longest number.
- However, note that, when "long" numbers are used, and the most significant digits are different, then MSD radix sort will likely take fewer iterations to finish than LSD radix sort, possibly just one (if there are few numbers).
- MSD radix sort is neither stable nor in-place.