Sorting

Consider sorting the values in an array A of size N. Most sorting algorithms involve what are called **comparison sorts**; i.e., they work by comparing values. Comparison sorts can never have a worst-case running time less than O(N log N). Simple comparison sorts are usually O(N^2); the more clever ones are O(N log N).

Three interesting issues to consider when thinking about different sorting algorithms are:

- Does an algorithm always take its worst-case time?
- What happens on an already-sorted array?
- How much space (other than the space for the array itself) is required?

We will discuss four comparison-sort algorithms:

1. selection sort
2. insertion sort
3. merge sort
4. quick sort

Selection sort and insertion sort have worst-case time O(N^2). Quick sort is also O(N^2) in the worst case, but its expected time is O(N log N). Merge sort is O(N log N) in the worst case.

Selection Sort

The idea behind selection sort is:

1. Find the smallest value in A; put it in A[0].
2. Find the second smallest value in A; put it in A[1].
3. etc.

The approach is as follows:

- Use an outer loop from 0 to N-1 (the loop index, k, tells which position in A to fill next).
- Each time around, use a nested loop (from k+1 to N-1) to find the smallest value (and its index) in the unsorted part of the array.
- Swap that value with A[k].

Note that after i iterations, A[0] through A[i-1] contain their final values (so after N iterations, A[0] through A[N-1] contain their final values and we're done!)

Here's the code for selection sort:

```
public static void selectionSort(Comparable[] A) {
  int j, k, minIndex;
  Comparable min;
  int N = A.length;

  for (k = 0; k < N; k++) {
     min = A[k];
     minIndex = k;
     for (j = k+1; j < N; j++) {
        if (A[j].compareTo(min) < 0) {
      min = A[j];
      minIndex = j;
  }
     }
     A[minIndex] = A[k];
     A[k] = min;
  }
}
```

What is the time complexity of selection sort? Note that the inner loop executes a different number of times each time around the outer loop, so we can't just multiply N * (time for inner loop). However, we can notice that:

- 1st iteration of outer loop: inner executes N - 1 times
- 2nd iteration of outer loop: inner executes N - 2 times
- ...
- Nth iteration of outer loop: inner executes 0 times

This is our old favorite sum:

$N-1 + N-2 + ... + 3 + 2 + 1 + 0$

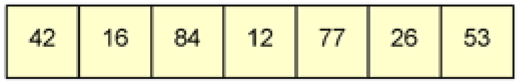
which we know is $O(N^2)$.

What if the array is already sorted when selection sort is called? It is still $O(N^2)$; the two loops still execute the same number of times, regardless of whether the array is sorted or not.

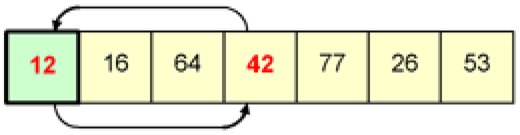
Selection sort code

- The algorithm works as follows:
- Find the minimum value in the list
- Swap it with the value in the first position.
- Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

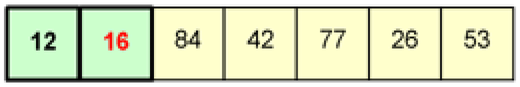
```
    void SelectionSort(int A[], int n)
     {
       int i, j, small, temp;
(1)    for (i = 0; i < n-1; i++) {
(2)      small = i;
(3)      for (j = i+1; j < n; j++)
(4)        if (A[j] < A[small])
(5)        small = j;
(6)    temp = A[small];
(7)    A[small] = A[i];
(8)    A[i] = temp;
         }
       }
```



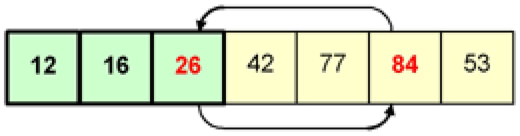
The array, before the selection sort operation begins.



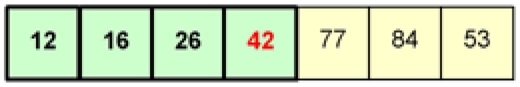
The smallest number (**12**) is swapped into the first element in the structure.



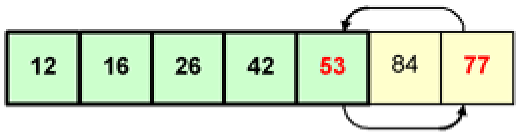
In the data that remains, **16** is the smallest; and it does not need to be moved.



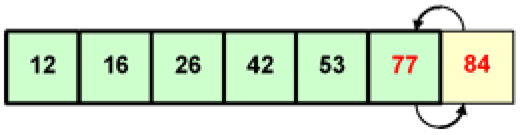
26 is the next smallest number, and it is swapped into the third position.



42 is the next smallest number; it is already in the correct position.



53 is the smallest number in the data that remains; and it is swapped to the appropriate position.



Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*

Insertion Sort

The idea behind insertion sort is:

1. Put the first 2 items in correct relative order.
2. Insert the 3rd item in the correct place relative to the first 2.
3. Insert the 4th item in the correct place relative to the first 3.
4. etc.

As for selection sort, a nested loop is used; however, a different invariant holds: after the ith time around the outer loop, the items in A[0] through A[i-1] are in order relative to each other (but are not necessarily in their final places). Also, note that in order to insert an item into its place in the (relatively) sorted part of the array, it is necessary to move some values to the right to make room.

Here's the code:

```
public static void insertionSort(Comparable[] A) {
  int k, j;
  Comparable tmp;
  int N = A.length;

  for (k = 1; k < N, k++) {
     tmp = A[k];
     j = k - 1;
     while ((j > = 0) && (A[j].compareTo(tmp) > 0)) {
         A[j+1] = A[j]; // move one value over one place to the right
         j--;
     }
     A[j + 1] = tmp; // insert kth value in correct place relative to
previous
                     // values
  }
}
```

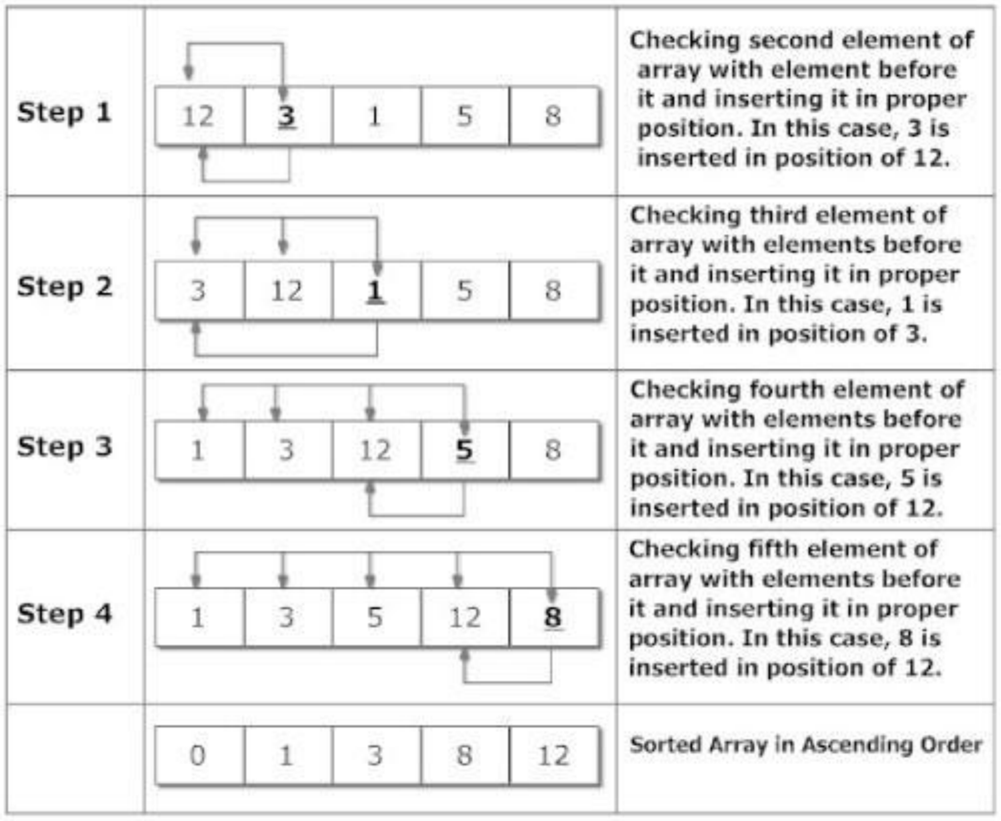
What is the time complexity of insertion sort? Again, the inner loop can execute a different number of times for every iteration of the outer loop. In the **worst** case:

- 1st iteration of outer loop: inner executes 1 time
- 2nd iteration of outer loop: inner executes 2 times
- 3rd iteration of outer loop: inner executes 3 times
- ...
- N-1st iteration of outer loop: inner executes N-1 times

So we get:

$1 + 2 + ... + N-1$

which is still O(N^2).



Step 1	12, **3**, 1, 5, 8	Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2	3, 12, **1**, 5, 8	Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3	1, 3, 12, **5**, 8	Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4	1, 3, 5, 12, **8**	Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
	0, 1, 3, 8, 12	Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Merge Sort

As mentioned above, merge sort takes time O(N log N), which is quite a bit better than the two O(N^2) sorts described above (for example, when N=1,000,000, N^2=1,000,000,000,000, and N $\log_2$N = 20,000,000; i.e., N^2 is 50,000 times larger than N log N!).

The key insight behind merge sort is that it is possible to **merge** two sorted arrays, each containing N/2 items to form one sorted array containing N items in time O(N). To do this merge, you just step through the two arrays, always choosing the smaller of the two values to put into the final array (and only advancing in the array from which you took the smaller value).

Now the question is, how do we get the two sorted arrays of size N/2? The answer is to use recursion; to sort an array of length N:

1. Divide the array into two halves.
2. Recursively, sort the left half.
3. Recursively, sort the right half.
4. Merge the two sorted halves.

The base case for the recursion is when the array to be sorted is of length 1 -- then it is already sorted, so there is nothing to do. Note that the merge step (step 4) needs to use an auxiliary array (to avoid overwriting its values). The sorted values are then copied back from the auxiliary array to the original array.

An outline of the code for merge sort is given below. It uses an auxiliary method with extra parameters that tell what part of array A each recursive call is responsible for sorting.

```
public static void mergeSort(Comparable[] A) {
  mergeAux(A, 0, A.length - 1); // call the aux. function to do all the
work
}

private static void mergeAux(Comparable[] A, int low, int high)
{
  // base case
     if (low == high) return;

  // recursive case
  // Step 1: Find the middle of the array (conceptually, divide it in
half)
     int mid = (low + high) / 2;
  // Steps 2 and 3: Sort the 2 halves of A
     mergeAux(A, low, mid);
     mergeAux(A, mid+1, high);

  // Step 4: Merge sorted halves into an auxiliary array
     Comparable[] tmp = new Comparable[high-low+1];
     int left = low;    // index into left half
     int right = mid+1; // index into right half
     int pos = 0;       // index into tmp

     while ((left <= mid) && (right <= high)) {
  // choose the smaller of the two values "pointed to" by left, right
  // copy that value into tmp[pos]
  // increment either left or right as appropriate
  // increment pos
  if (A[left].compareTo(A[right] < 0) {
      tmp[pos] = A[left];
      left++;
  }
  else {
```

```
            tmp[pos] = A[right];
            right++;
        }
        pos++;
     }
     // here when one of the two sorted halves has "run out" of values, but
     // there are still some in the other half; copy all the remaining values
     // to tmp

           }
           // here when one of the two sorted halves has "run out" of values,
      but
           // there are still some in the other half; copy all the remaining
      values
           // to tmp
           // Note: only 1 of the next 2 loops will actually execute
       while (left <= mid)
            {
               A[pos] = A[left];
               left++;
               pos++;
            }
       while (right <= high) {
           A[pos] = A[right];
           right++;
           pos++;
       }
```

Algorithms like merge sort -- that work by dividing the problem in two, solving the smaller versions, and then combining the solutions -- are called **divide and conquer** algorithms.

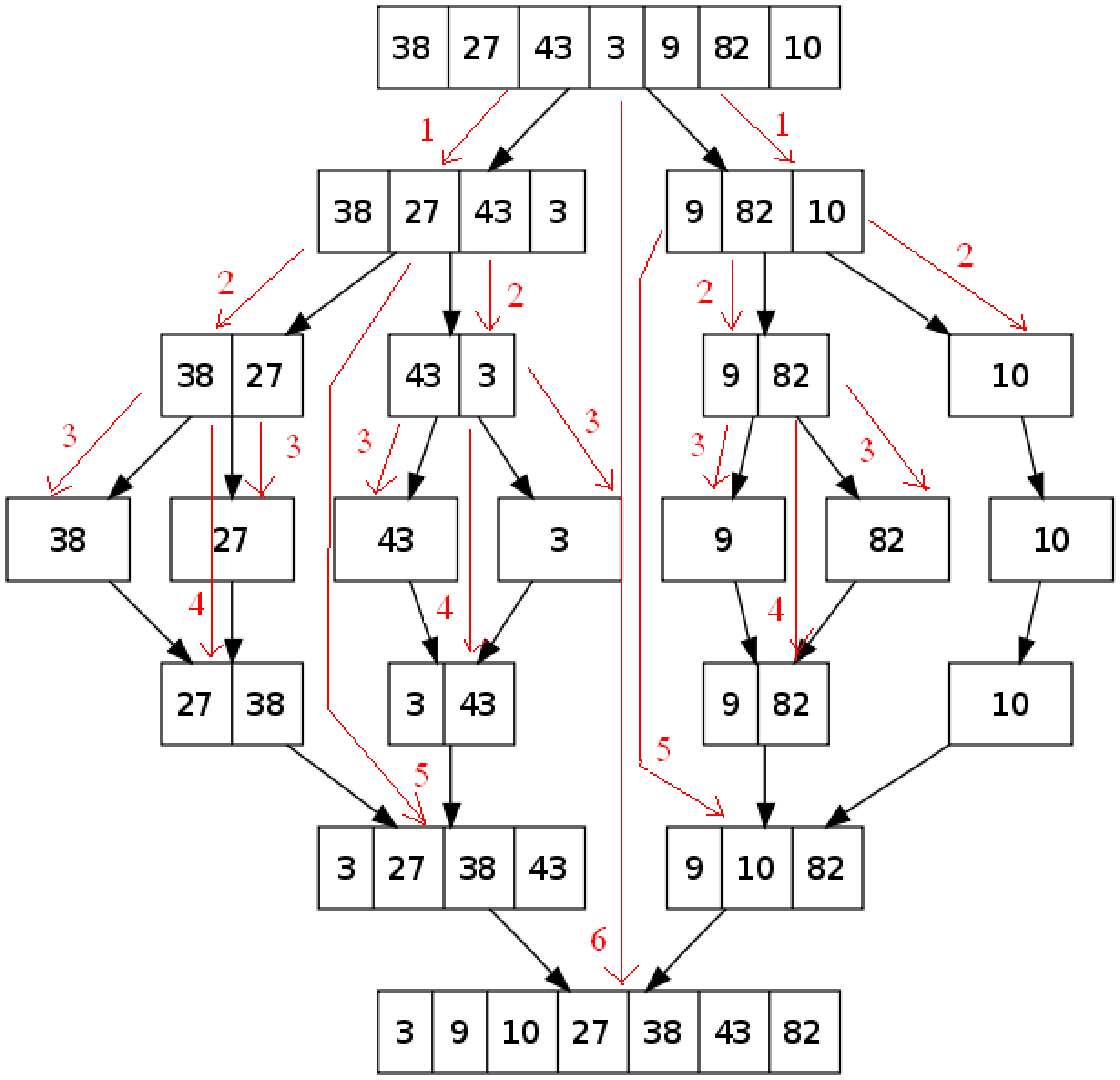
To determine the time for merge sort, it is helpful to visualize the calls made to mergeAux as shown below (each node represents one call, and is labeled with the size of the array to be sorted by that call):

The height of this tree is O(log N). The total work done at each "level" of the tree (i.e., the work done by mergeAux excluding the recursive calls) is O(N):

- Step 1 (finding the middle index) is O(1), and this step is performed once in each call; i.e., a total of once at the top level, twice at the second level, etc, down to a total of N/2 times at the second-to-last level (it is not performed at all at the very last level, because there the base case applies, and mergeAux just returns). So for any one level, the total amount of work for Step 1 is at most O(N).
- For each individual call, Step 4 (merging the sorted half-graphs) takes time proportional to the size of the part of the array to be sorted by that call. So for a

whole level, the time is proportional to the sum of the sizes at that level. This sum is always N.

Therefore, the time for merge sort involves O(N) work done at each "level" of the tree that represents the recursive calls. Since there are O(log N) levels, the total worst-case time is O(N log N).



Quick Sort

Quick sort (like merge sort) is a divide and conquer algorithm: it works by creating two problems of half size, solving them recursively, then combining the solutions to the small problems to get a solution to the original problem. However, quick sort does more work than merge sort in the "divide" part, and is thus able to avoid doing any work at all in the "combine" part!

The idea is to start by **partitioning** the array: putting all small values in the left half and putting all large values in the right half. Then the two halves are (recursively) sorted. Once that's done, there's no need for a "combine" step: the whole array will be sorted!

The key question is how to do the partitioning? Ideally, we'd like to put exactly half of the values in the left part of the array, and the other half in the right part; i.e., we'd like to put all values less than the **median** value in the left and all values greater than the median value in the right. However, that requires first computing the median value (which is too expensive). Instead, we pick one value to be the **pivot**, and we put all values less than the pivot to its left, and all values greater than the pivot to its right (the pivot itself is then in its final place).

Here's the algorithm outline:

1. Choose a pivot value.
2. Partition the array (put all value less than the pivot in the left part of the array, then the pivot itself, then all values greater than the pivot).
3. Recursively, sort the values less than the pivot.
4. Recursively, sort the values greater than the pivot.

Note that, as for merge sort, we need an auxiliary method with two extra parameters -- low and high indexes to indicate which part of the array to sort. Also, although we could "recurse" all the way down to a single item, in practice, it is better to switch to a sort like insertion sort when the number of items to be sorted is small (e.g., 20).

Now let's consider how to choose the pivot item. (Our goal is to choose it so that the "left part" and "right part" of the array have about the same number of items -- otherwise we'll get a bad runtime).

An easy thing to do is to use the first value -- A[low] -- as the pivot. However, if A is already sorted this will lead to the worst possible runtime.

In this case, after partitioning, the left part of the array is empty, and the right part contains all values except the pivot. This will cause O(N) recursive calls to be made (to sort from 0 to N-1, then from 1 to N-1, then from 2 to N-1, etc). Therefore, the total time will be O(N^2).

Another option is to use a random-number generator to choose a random item as the pivot. This is OK if you have a good, fast random-number generator.

A simple and effective technique is the "median-of-three": choose the median of the values in A[low], A[high], and A[(low+high)/2]. Note that this requires that there be at least 3 items in the array, which is consistent with the note above about using insertion sort when the piece of the array to be sorted gets small.

Once we've chosen the pivot, we need to do the partitioning. (The following assumes that the size of the piece of the array to be sorted is at least 3.) The basic idea is to use two "pointers" (indexes) left and right. They start at opposite ends of the array and move toward each other until left "points" to an item that is greater than the pivot (so it doesn't belong in the left part of the array) and right "points" to an item that is smaller than the pivot. Those two "out-of-place" items are swapped, and we repeat this process until left and right cross:

1. Choose the pivot (using the "median-of-three" technique); also, put the smallest of the 3 values in A[low], put the largest of the 3 values in A[high], and put the pivot in A[high-1]. (Putting the smallest value in A[low] prevents "right" from falling off the end of the array in the following steps.)
2. Initialize: left = low+1; right = high-2
3. Use a loop with the condition:

   ```
   while (left <= right)
   ```

   The loop invariant is:

   ```
   all items in A[low] to A[left-1] are <= the pivot
   all items in A[right+1] to A[high] are >= the pivot
   ```

   Each time around the loop:

   ```
   left is incremented until it "points" to a value > the pivot
   right is decremented until it "points" to a value < the pivot
   if left and right have not crossed each other,
   then swap the items they "point" to.
   ```

4. Put the pivot into its final place.

Here's the actual code for the partitioning step (the reason for returning a value will be clear when we look at the code for quick sort itself):

```
private static int partition(Comparable[] A, int low, int high) {
// precondition: A.length >= 3

    int pivot = medianOfThree(A, low, high); // this does step 1
    int left = low+1; right = high-2;
    while ( left <= right ) {
        while (A[left].compareTo(pivot) < 0) left++;
        while (A[right].compareTo(pivot) > 0) right--;
        if (left <= right) {
            swap(A, left, right);
            left++;
            right--;
        }
    }
    swap(A, left, high-1); // step 4
    return right;
}
```

After partitioning, the pivot is in A[right+1], which is its final place; the final task is to sort the values to the left of the pivot, and to sort the values to the right of the pivot. Here's the code for quick sort (so that we can illustrate the algorithm, we use insertion sort only when the part of the array to be sorted has less than 3 items, rather than when it has less than 20 items):

```
public static void quickSort(Comparable[] A) {
    quickAux(A, 0, A.length-1);
}

private static void quickAux(Comparable[] A, int low, int high) {
    if (high-low < 2) insertionSort(A, low, high);
    else {
        int right = partition(A, low, high);
  quickAux(A, low, right);
  quickAux(A, right+2, high);
    }
}
```

Note: It is important to handle duplicate values efficiently. In particular, it is not a good idea to put all values strictly less than the pivot into the left part of the array, and all values greater than or equal to the pivot into the right part of the array. The code given above for partitioning handles duplicates correctly at the expense of some "extra" swaps when both left and right are "pointing" to values equal to the pivot.

What is the time for Quick Sort?

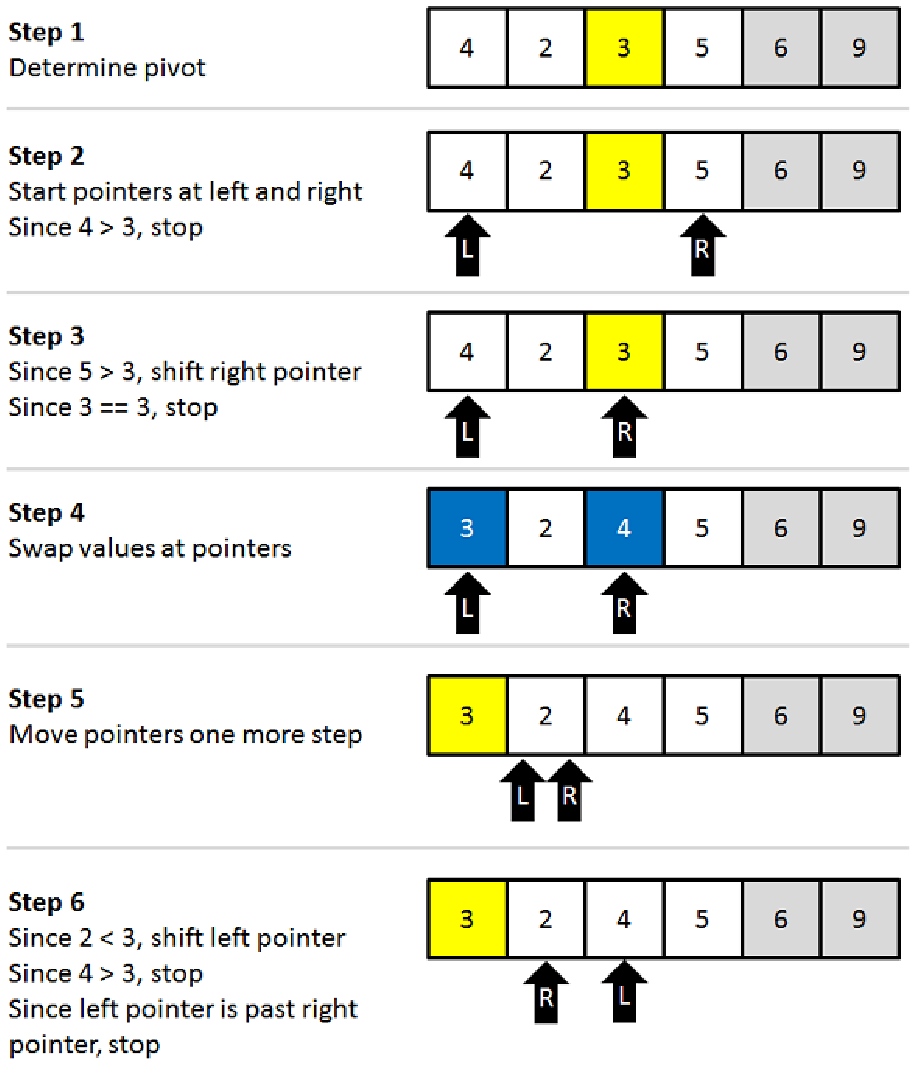
- If the pivot is always the median value, then the calls form a balanced binary tree (like they do for merge sort).

- In the worst case (the pivot is the smallest or largest value) the calls form a "linear" tree.
- In any case, the total work done at each level of the call tree is O(N) for partitioning.

So the total time is:

- worst-case: $O(N^2)$
- in practice: O(N log N)

Note that quick sort's worst-case time is worse than merge sort's. However, an advantage of quick sort is that it does not require extra storage, as merge sort does.



Step 1
Determine pivot

Step 2
Start pointers at left and right
Since 4 > 3, stop

Step 3
Since 5 > 3, shift right pointer
Since 3 == 3, stop

Step 4
Swap values at pointers

Step 5
Move pointers one more step

Step 6
Since 2 < 3, shift left pointer
Since 4 > 3, stop
Since left pointer is past right pointer, stop

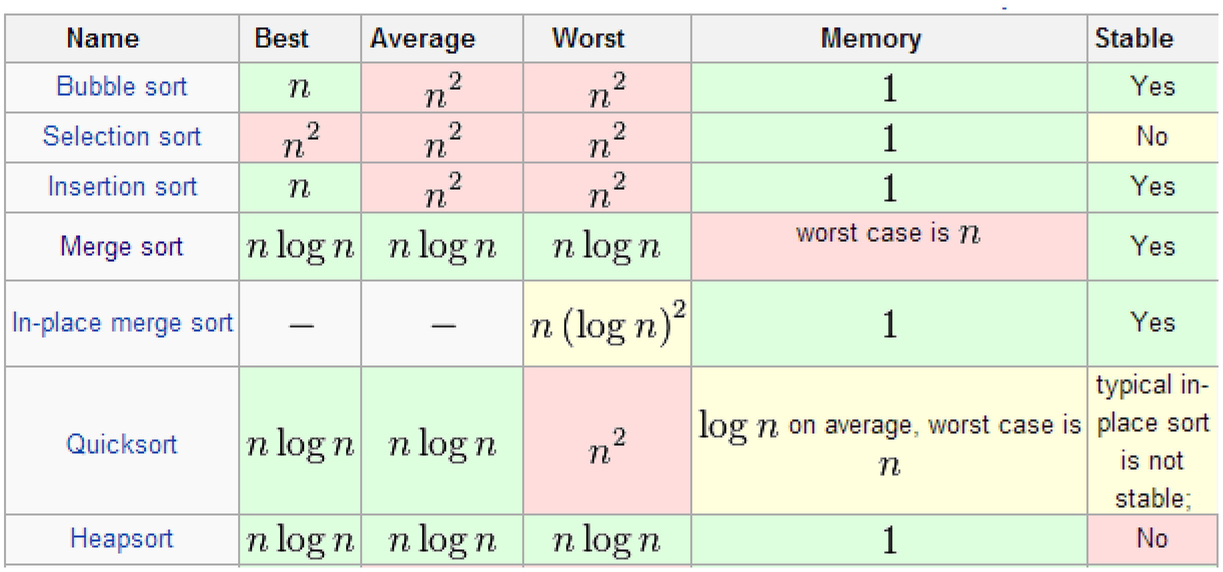
Sorting Summary

- Selection Sort:
  - N passes
    on pass k: find the kth smallest item, put it in its final place
  - always $O(N^2)$
- Insertion Sort:
  - N passes
    on pass k: insert the kth item into its proper position relative to the items to its left
  - worst-case $O(N^2)$
  - given an already-sorted array: O(N)
- Merge Sort:
  - recursively sort the first N/2 items
    recursively sort the last N/2 items
    merge (using an auxiliary array)
  - always O(N log N)
- Quick Sort:
  - choose a pivot value
    partition the array:

    left part has items <= pivot
    right part has items >= pivot

    recursively sort the left part
    recursively sort the right part

  - worst-case $O(N^2)$
  - expected O(N log N)

- 

Name	Best	Average	Worst	Memory	Stable
Bubble sort	n	n^2	n^2	1	Yes
Selection sort	n^2	n^2	n^2	1	No
Insertion sort	n	n^2	n^2	1	Yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	worst case is n	Yes
In-place merge sort	—	—	$n\left(\log n\right)^2$	1	Yes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$ on average, worst case is n	typical in-place sort is not stable;
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No