

Algorithms, Sorting, and Computational Methods

Timothy
Green

Maths
for IT

5N18396

9th February
2026

Table of Contents

Introduction	1
Abstractions leak	1
Research Documentation	2
Definition of <i>algorithm</i>	2
Example: The Fourier Transform	2
Sorting algorithms	2
Bubble sort	3
Selection sort	5
Quicksort	5
Comparison of sorting algorithms	8
Outline of Solution	10
Source Code	12
Testing & Verification	15
Debug Mode Output	16
PHP Translation	18
Conclusion	21
References	22

Introduction

In this document, three sorting algorithms are explained, and one is implemented. All three algorithms have the same effect: they all sort data. (For the purpose of the demonstrations in this document, the data is an array of integers, but all the algorithms can in principle be applied to any sortable data, such as strings (sorted alphabetically) or floats.)

It is important to understand that there can be many ways to achieve the same effect. There are many different sort algorithms, which work in fundamentally different ways.

Array Indexing

In examples in this document, arrays are indexed from 0, as they are in Java and most other programming languages. (This is not true of *all* programming languages. Lua, for example, is an exception, and indexes its arrays from 1.) That means that in examples in this document, the first number is in position 0, the second number is in position 1, etc.

Abstractions leak

Any higher-level language will have a built-in `sort()` function. A programmer using it does not usually need to know the details of how it's implemented. For example, the documentation for PHP's `sort()` function does not explain in detail how it works, though one may be able to draw some conclusions from these two notes on the page:

If two members compare as equal, they retain their original order. Prior to PHP 8.0.0, their relative order in the sorted array was undefined.

— PHP (2023)

Like most PHP sorting functions, `sort()` uses an implementation of Quicksort. The pivot is chosen in the middle of the partition resulting in an optimal time for already sorted arrays. This is however an implementation detail you shouldn't rely on.

— PHP (2023)

Most PHP developers have no need to know the underlying details. However, the fact that PHP used to use an unstable sort function is the kind of abstraction which occasionally leaks. Should a programmer have needed a stable sort function in PHP prior to version 8.0.0, they would have been required to implement it themselves.

In actual fact, a consultation of the source code reveals that for arrays of 16 or fewer items, PHP uses insert sort instead. (PHP, 2023)

An excellent treatment of leaky abstractions is given in Spolsky (2002), which introduced the term, expanding on the earlier Spolsky (2001). There are times when it pays to understand what's happening “under the hood” in a programming language.

Research Documentation

Definition of *algorithm*

The word *algorithm* has existed in English since the 1690s, originally meaning “the Arabic system of computation” (i.e., algebra) which is solved in a step-by-step process. (Harper and Felix, no date)

The modern meaning of the word follows from that. An algorithm is a set of instructions for completing a task. The word is sometimes applied to the actual code or compiled binary which executes those instructions, but usually it is applied to the instructions themselves, and the code is called an *implementation* of the algorithm.

While any set of instructions can be called an algorithm (a recipe is a common exemplar), the word is most often used specifically of mathematical instructions, as in the original definition (algebra).

An algorithm is a series of mathematical steps, especially in a computer program, which will give you the answer to a particular kind of problem or question.

— HarperCollins (2023)

Example: The Fourier Transform

$$\hat{g}(f) = \int_{-\infty}^{+\infty} g(t)e^{-2\pi ift} dt$$

The Fourier Transform decomposes a complex signal, $g(t)$ into its component parts. (Sanderson, Schloss and Way, 2018) An example application lies in infrared spectroscopy. Covalent bonds will absorb infrared radiation, and each bond has a characteristic absorption frequency. Therefore, a spectral analysis can show which bonds are present in the sample molecule, and thereby help to analyse the composition of a substance. However, most samples will have many bonds, with many overlapping absorption frequencies. The Fourier Transform separates out the underlying individual frequencies, so that the makeup of the sample can better be determined.

There is more than one algorithm to compute the Fourier Transform. The most straightforward has a complexity of $O(n^2)$, but Fast Fourier Transform methods have a complexity of $O(n \log n)$, significantly faster.

Sorting algorithms

There are many ways to sort an unsorted list. In this description, all examples will be for an array of integers (in Java, `int[]`), though the same algorithms would apply to strings (sorted alphabetically), floats, and other data types.

A sorting algorithm may or may not be *stable*. In a stable sorting algorithm, two identical values remain in the same order they start in. In an unstable algorithm, this is not guaranteed. Of course, for a simple `int[]` array, this makes no difference: two identical values are fully identical, and the question of which order they're in is meaningless. However, for other data structures, the distinction does make sense.

A PHP array, for example, has keys, values, and an order. It may be desirable to reorder the array, sorting by either keys or values. In Java, one may store people's names and ages in a data structure such as `LinkedHashMap<Integer, String>`.

42	Arthur Dent	stable sort by age →	17	Susan Sto Helit
17	Susan Sto Helit		42	Arthur Dent
50	Frodo Baggins		42	Ford Prefect
42	Ford Prefect		50	Frodo Baggins

After sorting, Arthur and Ford's relative orders are unchanged (Arthur comes first), as they have the same age. An unstable sorting algorithm does not make this guarantee. (To be clear, an unstable algorithm does not guarantee that the positions *will* change, either. They might; they might not.)

Bubble sort

The main benefit of bubble sort is that it is straightforward to explain and easy to implement, so it is frequently used as a coding exercise. It is rarely used in actual practice, as its $O(n^2)$ complexity makes it slow for anything but very small data sets.

Big O Notation

The actual number of operations is not quite n^2 , depending on what is counted as an "operation". An array of n items will be looped over n times: within the first loop there will be $n - 1$ comparisons; within the second loop there will be $n - 2$ comparisons; and so on. Each comparison may or may not be followed by a swap (which is itself three operations). Big O notation captures the magnitude of the operations, not an exact number.

Let's sort an array of four integers, three distinct. As the first step of the first iteration, the first two integers (in positions 0 and 1 in the array) are compared.

42	17	50	42
----	----	----	----

As 42 is larger than 17, the two numbers are swapped.

17	42	50	42
----	----	----	----

42 is not larger than 50, so these two numbers are left as is.

17	42	50	42
----	----	----	----

50 is larger than 42, so these two numbers are swapped, completing the first pass over the array.

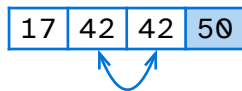
17	42	42	50
----	----	----	----

The first pass is complete.

At the end of the first iteration, there have been three comparisons and two swaps, for a total of five operations. And our task is far from complete. As humans looking at the numbers, we can see that it is now completely in order, but algorithmically all we have proven is that the number now at the extreme right is the largest. All the other numbers have not yet been fully sorted, so further operations are needed.

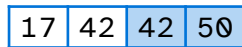
17	42	42	50
----	----	----	----

17 is not larger than 42, so these two numbers are left as is.



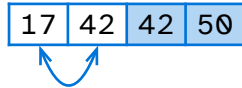
This is an interesting one.

The logic runs as follows: 42 is not larger than 42, so these two numbers are left as is. This is what makes bubble sort a stable sort algorithm.

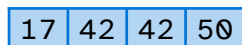


This brings us to the end of the second pass.

Now the third pass begins. There is only one comparison left to make.



17 is not larger than 42, so the items can be left in their current position.



The array is now sorted.

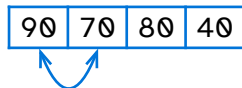
Swapping Values

Bubble sort requires many *swaps* of values. Let's say that we want to swap the value in position *i* with the value in position *i+1*. There is no single operation to do that. Instead, we first store one of the values into a temporary variable, thus:

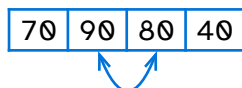
```
int temp = arr[i]; // Store first value into temp.  
arr[i] = arr[i+1]; // Overwrite first value with second.  
arr[i+1] = temp;   // Store temp value into second.
```

Therefore, each swap is three operations. Bubble sort may have many swaps.

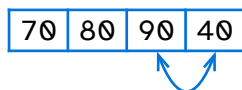
Here's a second example, which demonstrates why multiple iterations are needed. This array is not in the correct order until the final step.



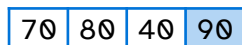
The first comparison of the first pass results in a swap.



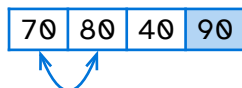
The second comparison of the first pass results in a swap.



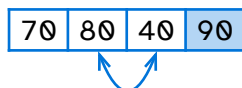
The third comparison of the first pass results in a swap.



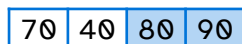
The first pass is complete, and the last item is now fixed in place.



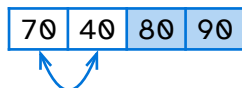
The first comparison of the second pass results in no swap.



The second comparison of the second pass results in a swap.



The second pass is complete, and the last two items are now fixed in place.



The first (and only) comparison of the third pass results in a swap.

40	70	80	90
----	----	----	----

The list is now sorted.

Selection sort

Selection sort approximates the way a human would sort a small list, by finding the first object and moving it to the beginning. Like bubble sort, it swaps elements. Each element is compared in turn with each other element, so once again the complexity is $O(n^2)$. Therefore, as with bubble sort, selection sort is unlikely to be used in practice. It is used mostly for purposes of paedagogy. (Kartik, 2025)

90	70	80	40
----	----	----	----

The first comparison of the first pass results in a swap.

70	90	80	40
----	----	----	----

The new first value, 70, is now compared with the third value, and is not swapped.

70	90	80	40
----	----	----	----

70 is larger than 40, so the values are swapped.

40	90	80	70
----	----	----	----

The first pass is now complete, and the smallest value is in the first position.

40	90	80	70
----	----	----	----

90 is larger than 80, so the values are swapped.

40	80	90	70
----	----	----	----

80 is larger than 70, so the values are swapped.

40	70	90	80
----	----	----	----

The second pass is now complete, and the two smallest values are in place.

40	70	90	80
----	----	----	----

90 is larger than 80, so the values are swapped.

40	70	80	90
----	----	----	----

The list is now sorted.

Selection sort, unlike bubble sort, is unstable.

42	Arthur Dent	selection sort by age →	17	Susan Sto Helit
50	Frodo Baggins		42	Ford Prefect
42	Ford Prefect		42	Arthur Dent
17	Susan Sto Helit		50	Frodo Baggins

The first step is to swap the positions of Susan and Arthur, putting Arthur after Ford. Later steps do not undo this rearrangement. The algorithm can be modified to make it stable, but then it isn't truly selection sort any more. (Shilimkar, 2013)

Quicksort

Quicksort (it is usually written as a single word) is a more complex algorithm to explain. Unlike bubble sort and selection sort, quicksort does have real-world use, even though its complexity may also be $O(n^2)$. More commonly, though, the complexity is $O(n \log n)$. In the best-case

scenario, verifying an array which is already in order, it's as fast as $O(n)$, which means that the complexity is linearly proportional to the size of the input array.

The first step of the quicksort algorithm is to choose an element as the *pivot*. Different implementations of quicksort make different choices here. Some always pick the first or final element; others take an element near the middle. A discussion of the relative merits of various partition methods can be found in Wild (2013). The version demonstrated here uses the final element as the pivot.

The array is then split into three portions: the pivot element itself, as a single element, an array of all elements smaller than the pivot, and an array of all elements larger than the pivot. To achieve this, each other element must be compared with the pivot element.

This technique is then applied recursively to each of the two sub-arrays.

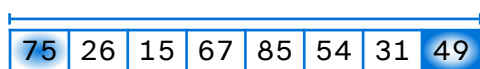
The description above suggests creating multiple smaller and smaller arrays. This would consume a lot of memory. In practice, elements can be moved within the array, and in the recursion step, quicksort can be applied only to *ranges* of the array, so sorting happens in place. When all sub-arrays contain only one element each, the array as a whole is sorted.

Since quicksort splits an array into segments, it is difficult to illustrate with a four-element array: it's too small to show clearly what's happening. For this demonstration, an eight-element array will be used. This example is drawn from Paidipati (2023), with an implementation which always chooses the final element to be the pivot.

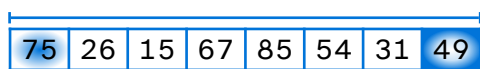
49 The *pivot* is shown with a central white highlight.

75 The *index* is shown with a central blue blur.

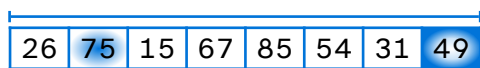
As we iterate through the array, the current item (*i*) is compared with the pivot. If the item is smaller than or equal to the pivot, it is swapped with the index and the index is advanced by one. That means that at the start, the index is the same as *i*, but this may change.



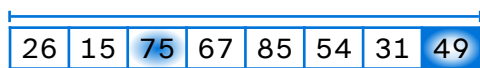
75 is larger than 49, so no change is made.
The index remains in position 0.



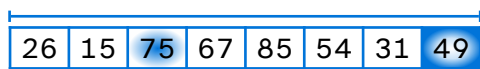
26 is smaller than 49, so is swapped with the index.
The index is then advanced.



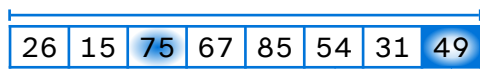
15 is smaller than 49, so is swapped with the index.
The index is then advanced.



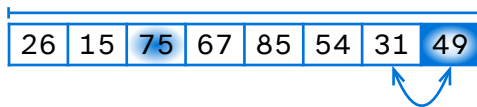
67 is larger than 49, so no change is made.
The index remains in position 2.



85 is larger than 49, so no change is made.
The index remains in position 2.



54 is larger than 49, so no change is made.
The index remains in position 2.



31 is smaller than 49, so is swapped with the index. The index is then advanced.



This is how the array looks now.

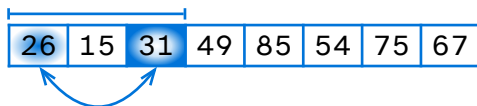


The pivot and the index are then swapped.

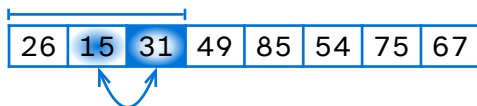
Every value to the left of the pivot is now smaller than the pivot; every value to the right is now larger than or equal to the pivot. However, the array is still far from sorted.

We can now sort just the sub-array of the first three values (i.e., the values to the left of the pivot). In diagrams of quicksort, this is often shown as splitting the array into sub-arrays, and sorting each individually before putting them back together. However, in actuality, this would consume a lot of memory, so instead the sub-array is sorted in place, as part of the larger array.

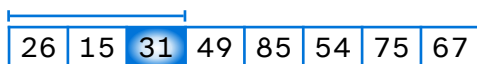
This is interesting to watch, as the values here are both smaller than the pivot.



26 is smaller than 31, so is swapped with the index. But 26 *is* the index, so nothing changes. The index is then advanced.

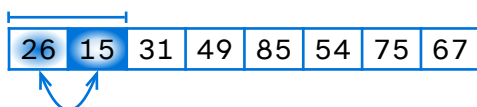


15 is smaller than 31, so is swapped with the index. But 15 *is* the index, so nothing changes. The index is then advanced.

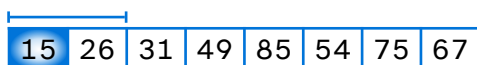


The final step is to swap pivot and index, but once again they're the same number, so no change is made.

We now work on the even smaller sub-array of the first two elements.

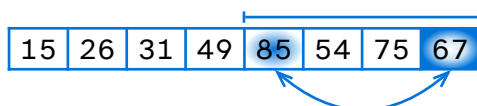


26 is larger than 15, so no change is made. The index remains in position 0.

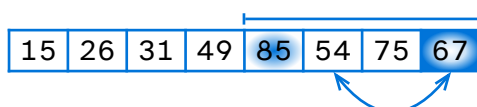


The pivot and the index are then swapped.

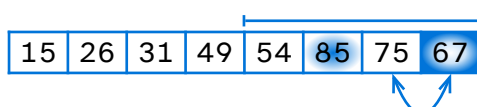
And now work starts on the remaining sub-array of the final four elements, to the right of the original pivot (49).



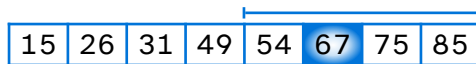
85 is larger than 67, so no change is made. The index remains in position 4.



54 is smaller than 67, so is swapped with the index. The index is then advanced.

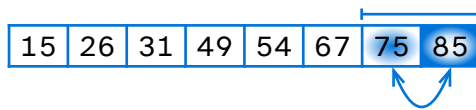


75 is larger than 67, so no change is made. The index remains in position 5.

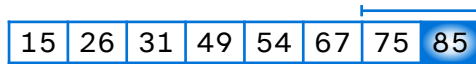


The pivot and the index are then swapped.

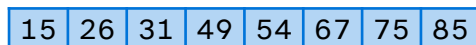
The final step is to sort the sub-array of the last two elements.



75 is smaller than 85, so is swapped with the index. But 75 *is* the index, so nothing changes. The index is then advanced.



The final step is to swap pivot and index, but once again they're the same number, so no change is made.



The whole array is now, finally, sorted.

The inventor of quicksort, Tony Hoare, published it as a recursive Algol program, shown below. (Hoare, 1961)

```

procedure quicksort (A,M,N); value M,N;
      array A; integer M,N;
comment Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
requird. The average number of comparisons made is 2(M-N) ln
(N-M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin      integer I,J;
           if M < N then begin partition (A,M,N,I,J);
                           quicksort (A,M,J);
                           quicksort (A, I, N)
           end
end        quicksort

```

Comparison of sorting algorithms

Big O Notation Is for Comparison

For $n = 4$, both bubble sort and selection sort result in three passes over the array, six comparisons, and up to six swaps, with each swap being three operations. Depending on how one counts this, this equates to 27 operations, even though $4^2 = 16$. Why, then, do we say that the complexity is $O(n^2)$? The reason is that it scales with the square of the size of the input. As n increases, the number of operations increases in square proportion.

For an array which is already sorted, both bubble sort and selection sort will still do the same number of comparisons. However, quicksort will do only $n - 1$ comparisons in that situation. This makes quicksort far more efficient on input which is already in sort order.

This means that bubble sort and selection sort are always of complexity $O(n^2)$, while quicksort is somewhere between $O(n)$ (in the best case) and $O(n^2)$ (in the worst case), and for data in random order is usually approximately $O(n \log n)$.

Bubble sort and selection sort are very rarely used in actual applications, though they are useful for teaching the principles. Quicksort, however, is commonly used.

Outline of Solution

The solution built is quicksort, based on a tutorial from Paidipati (2023). This implementation was modified in many ways from the tutorial example.

The version presented below takes input, so it can sort any data provided by the user.

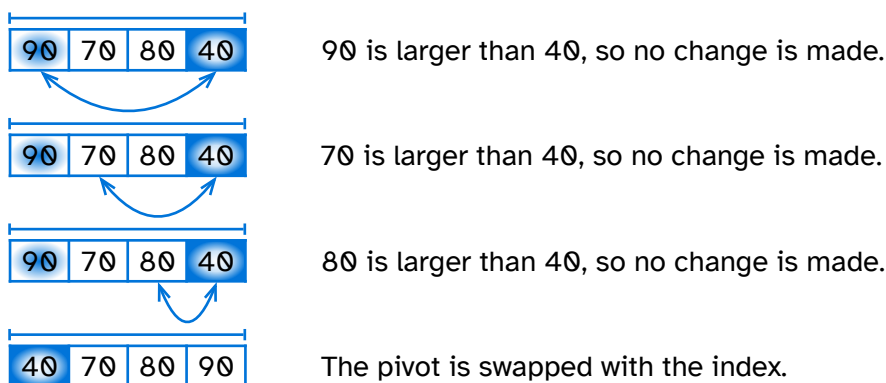
In Java, an array has a fixed length. Drawing on Pombrio (2024), which describes an `ArrayList` as a “growable array” in Java, input is taken in that format and then converted to an array. This makes life easier for the user: they do not need to first count the elements of the array they wish to sort. Conversion of an `ArrayList` to a normal Java array was based on guidance from kartik (2025).

The version in the tutorial may perform redundant swaps. For example, in the original, the three lines of code below would run even where `pIndex` and `end` are the same. This version skips such swaps at the expense of adding a further `if` branch. Whether this increases or decreases the speed of the operation overall is hard to say, and would probably require benchmarking, but it certainly makes the logic easier to follow.

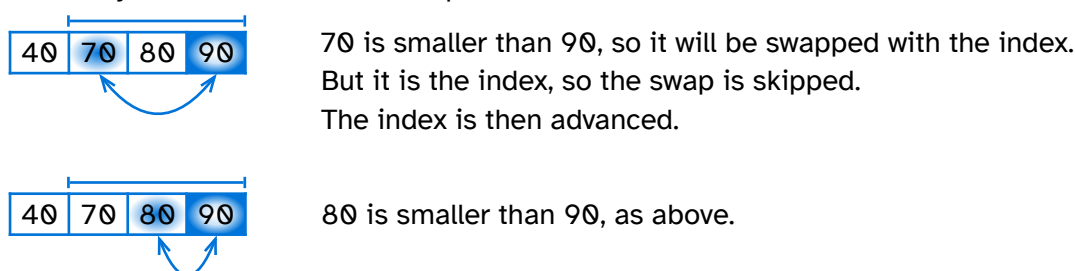
```
int t = A[pIndex];
A[pIndex] = A[end];
A[end] = t;
```

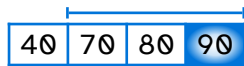
To aid in understanding the algorithm, a debug mode has been added, which prints out the state of the array at each step, and also explains what changes are being made. This was helpful in outlining the process of the algorithm.

A full walkthrough of sorting an eight-element array is given above, but the process for a shorter four-element array is given here.



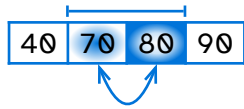
The array will now be sorted from positions 1 to 3.



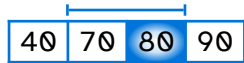


The index is now the pivot, so this swap is also skipped.

The array will now be sorted from positions 1 to 2.



70 is smaller than 80. The swap is skipped.
The index is then advanced.

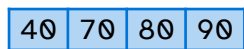


The index is now the pivot, so this swap is also skipped.

The array will now be sorted from positions 1 to 1.



The index is the pivot. No changes are made.



The array is now sorted.

In pseudocode,

```
let array = user.input
# Details of taking input omitted for clarity.

call QuickSort(
  array,
  start: first position in array,
  end: last position in array
)

QuickSort(array, start, end):
  if start is before end:
    # Partition function
    create index equal to start
    step through the array from start to end:
      if value at each point is less than or equal to value at end:
        swap the values at start and index
        advance index by one
    swap value at index with value at end
    call QuickSort(
      array,
      start: same start,
      end: one before the index
    )
    call QuickSort(
      array,
      start: one after the index,
      end: same end
    )
  )
```

It can be seen how the recursive calls to QuickSort split the array into ever-decreasing segments until the full array is sorted.

Source Code

```
/**
 * Based on the tutorial at https://www.csestack.org/quicksort/
 *
 * Changes made from the Tutorial implementation:
 *   Take input, not a hardcoded array.
 *   And translate that input from an ArrayList.
 * Add a debug mode.
 * Skip redundant swaps, to make the implementation more efficient.
 */

/**
 * This import is needed for
 *   the InputMismatchException,
 *   the ArrayList,
 *   and the Scanner.
 */
import java.util.*;

public class MySort {
    static boolean debug = false;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int[] arr = getArrayInput();
        System.out.println("Thank you. Your input list is: ");
        printArr(arr);

        System.out.print("Sort in debug mode (y/n): ");
        debug = sc.nextLine().trim().toLowerCase().equals("y");

        MySort sorter = new MySort();
        sorter.QuickSort(arr, 0, arr.length - 1);
        System.out.println("And the sorted input is: ");
        printArr(arr);
    }

    private static int[] getArrayInput() {
        Scanner sc = new Scanner(System.in);
        List<Integer> input = new ArrayList<>();
        int newInt;
        int i = 0;
        do {
            if (i == 0) {
                System.out.print("Please enter an integer: ");
            } else if (i == 1) {
                System.out.print("Please enter another integer: ");
            } else {
                System.out.print("Enter another integer, or enter the letter X to complete your input: ");
            }
            try {
                newInt = sc.nextInt();
                i++;
                input.add(newInt);
            } catch (InputMismatchException e) {
                if (i > 1) {

```

```

String text = sc.nextLine();
if (text.trim().toLowerCase().equals("x")) {
    break;
} else {
    System.out.println("Please enter an integer, or the letter X to complete your
input.");
}
} else {
    System.out.println("Sorry. We can cope only with integers.");
    System.out.println("Please try again.");
    sc.nextLine(); // We need to read and discard this.
}
}
} while (true);

int[] arr = input.stream().mapToInt(j -> j).toArray();

return arr;
}

void QuickSort(int A[], int start, int end) {
    if (start <= end) {
        // calling partition function
        int pIndex = Partition(A, start, end);
        QuickSort(A, start, pIndex - 1);
        QuickSort(A, pIndex + 1, end);
    }
}

int Partition(int A[], int start, int end) {
    int pivot = A[end]; // pivot here is the actual value
    if (debug) {
        System.out.println("Sorting the array from positions " + start + " to " + end + ".");
        System.out.println("The pivot is at position " + end + " and has the value " + pivot +
".");
    }

    // set the partition index as start initially
    int pIndex = start;
    for (int i = start; i < end; i++) {
        // swap if element is smaller than pivot
        if (debug) {
            printArr(A);
            System.out.println("The value at position " + i + " is " + A[i] + ".");
        }
        if (A[i] <= pivot) {
            if (debug) {
                System.out.println("This value is smaller than or equal to the pivot, so it will be
swapped with the index, which is at position " + pIndex + ".");
            }
            if (i == pIndex) {
                if (debug) {
                    System.out.println("However, the position is the index, so there's nothing to
swap.");
                }
            }
            else {
                int temp = A[i];
                A[i] = A[pIndex];
                A[pIndex] = temp;
            }
            pIndex = pIndex + 1;
        }
    }
}

```

```
}  
// Perform swapping operation with  
// the element at the partition index  
if (debug) {  
    System.out.println("And now the pivot will be swapped with the index.");  
}  
if (pIndex == end) {  
    if (debug) {  
        System.out.println("However, the index is the pivot, so there's nothing to swap.");  
    }  
} else {  
    int t = A[pIndex];  
    A[pIndex] = A[end];  
    A[end] = t;  
}  
return pIndex;  
}  
  
private static void printArr(int[] arr) {  
    System.out.println(Arrays.toString(arr));  
}  
}
```

Testing & Verification

The program includes an option to run in debug mode, which outputs the full state of the array after each step. This was intended as an aid to understanding. The full output from a run in debug mode is not shown in these screenshots, as it is too bulky, but one is included below.

```
Please enter an integer: 4
Please enter another integer: 2
Enter another integer, or enter the letter X to complete your input: 42
Enter another integer, or enter the letter X to complete your input: 78
Enter another integer, or enter the letter X to complete your input: 9
Enter another integer, or enter the letter X to complete your input: x
Thank you. Your input list is:
[4, 2, 42, 78, 9]
Sort in debug mode (y/n): n
And the sorted input is:
[2, 4, 9, 42, 78]
```

```
Please enter an integer: 47
Please enter another integer: 74
Enter another integer, or enter the letter X to complete your input: x
Thank you. Your input list is:
[47, 74]
Sort in debug mode (y/n): n
And the sorted input is:
[47, 74]
```

```
Please enter an integer: 1785825
Please enter another integer: 12555487
Enter another integer, or enter the letter X to complete your input: 89554
Enter another integer, or enter the letter X to complete your input: 784554519
Enter another integer, or enter the letter X to complete your input: 3
Enter another integer, or enter the letter X to complete your input: x
Thank you. Your input list is:
[1785825, 12555487, 89554, 784554519, 3]
Sort in debug mode (y/n): n
And the sorted input is:
[3, 89554, 1785825, 12555487, 784554519]
```

Test inputs above are of array sizes 5, 2, and 5. Below (in debug mode) are of array sizes 7 and 3.

PHP Version

As an additional form of verification, the program was re-implemented by hand in PHP, a language with which the author is more familiar. This was intended mostly as a personal aid to understanding, but is appended in case it might be of interest.

Debug Mode Output

```
Please enter an integer: 7
Please enter another integer: 56
Enter another integer, or enter the letter X to complete your input: 25
Enter another integer, or enter the letter X to complete your input: 56
Enter another integer, or enter the letter X to complete your input: 14
Enter another integer, or enter the letter X to complete your input: 17
Enter another integer, or enter the letter X to complete your input: 19
Enter another integer, or enter the letter X to complete your input: x
Thank you. Your input list is:
[7, 56, 25, 56, 14, 17, 19]
Sort in debug mode (y/n): y
Sorting the array from positions 0 to 6.
The pivot is at position 6 and has the value 19.
[7, 56, 25, 56, 14, 17, 19]
The value at position 0 is 7.
This value is smaller than or equal to the pivot, so it will be swapped with the index,
which is at position 0.
However, the position is the index, so there's nothing to swap.
[7, 56, 25, 56, 14, 17, 19]
The value at position 1 is 56.
[7, 56, 25, 56, 14, 17, 19]
The value at position 2 is 25.
[7, 56, 25, 56, 14, 17, 19]
The value at position 3 is 56.
[7, 56, 25, 56, 14, 17, 19]
The value at position 4 is 14.
This value is smaller than or equal to the pivot, so it will be swapped with the index,
which is at position 1.
[7, 14, 25, 56, 56, 17, 19]
The value at position 5 is 17.
This value is smaller than or equal to the pivot, so it will be swapped with the index,
which is at position 2.
And now the pivot will be swapped with the index.
Sorting the array from positions 0 to 2.
The pivot is at position 2 and has the value 17.
[7, 14, 17, 19, 56, 25, 56]
The value at position 0 is 7.
This value is smaller than or equal to the pivot, so it will be swapped with the index,
which is at position 0.
However, the position is the index, so there's nothing to swap.
[7, 14, 17, 19, 56, 25, 56]
The value at position 1 is 14.
This value is smaller than or equal to the pivot, so it will be swapped with the index,
which is at position 1.
However, the position is the index, so there's nothing to swap.
And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
Sorting the array from positions 0 to 1.
The pivot is at position 1 and has the value 14.
[7, 14, 17, 19, 56, 25, 56]
The value at position 0 is 7.
This value is smaller than or equal to the pivot, so it will be swapped with the index,
which is at position 0.
However, the position is the index, so there's nothing to swap.
And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
Sorting the array from positions 0 to 0.
The pivot is at position 0 and has the value 7.
```

And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
Sorting the array from positions 4 to 6.
The pivot is at position 6 and has the value 56.
[7, 14, 17, 19, 56, 25, 56]
The value at position 4 is 56.
This value is smaller than or equal to the pivot, so it will be swapped with the index, which is at position 4.
However, the position is the index, so there's nothing to swap.
[7, 14, 17, 19, 56, 25, 56]
The value at position 5 is 25.
This value is smaller than or equal to the pivot, so it will be swapped with the index, which is at position 5.
However, the position is the index, so there's nothing to swap.
And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
Sorting the array from positions 4 to 5.
The pivot is at position 5 and has the value 25.
[7, 14, 17, 19, 56, 25, 56]
The value at position 4 is 56.
And now the pivot will be swapped with the index.
Sorting the array from positions 5 to 5.
The pivot is at position 5 and has the value 56.
And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
And the sorted input is:
[7, 14, 17, 19, 25, 56, 56]

```
Please enter an integer: 17
Please enter another integer: 14
Enter another integer, or enter the letter X to complete your input: 23
Enter another integer, or enter the letter X to complete your input: x
Thank you. Your input list is:
[17, 14, 23]
Sort in debug mode (y/n): y
Sorting the array from positions 0 to 2.
The pivot is at position 2 and has the value 23.
[17, 14, 23]
The value at position 0 is 17.
This value is smaller than or equal to the pivot, so it will be swapped with the index, which is at position 0.
However, the position is the index, so there's nothing to swap.
[17, 14, 23]
The value at position 1 is 14.
This value is smaller than or equal to the pivot, so it will be swapped with the index, which is at position 1.
However, the position is the index, so there's nothing to swap.
And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
Sorting the array from positions 0 to 1.
The pivot is at position 1 and has the value 14.
[17, 14, 23]
The value at position 0 is 17.
And now the pivot will be swapped with the index.
Sorting the array from positions 1 to 1.
The pivot is at position 1 and has the value 17.
And now the pivot will be swapped with the index.
However, the index is the pivot, so there's nothing to swap.
And the sorted input is:
[14, 17, 23]
```

PHP Translation

```
<?php

function getInput() {
    $handle = STDIN;
    $arr = [];
    $i = 0;

    do {
        if ($i === 0) {
            $query = 'Please enter an integer: ';
        } elseif ($i === 1) {
            $query = 'Please enter another integer: ';
        } else {
            $query = 'Enter another integer, or enter the letter X to complete your input: ';
        }
        echo $query;
        $i++;

        $input = trim(fgets($handle));

        if (ctype_digit($input)) {
            $arr[] = (int)$input;
        } elseif (($input === 'x') && ($i > 1)) {
            break;
        } elseif ($i > 1) {
            echo 'Please enter an integer, or the letter X to complete your input.' . PHP_EOL;
        } else {
            echo 'Sorry. We can cope only with integers.' . PHP_EOL;
            echo 'Please try again.' . PHP_EOL;
        }
    } while (true);

    return $arr;
}

function flattenArr($arr) {
    return '[' . implode(', ', $arr) . ']' . PHP_EOL;
}

function manage() {
    $arr = getInput();

    echo 'Thank you. Your input list is:' . PHP_EOL;
    echo flattenArr($arr);

    echo 'Sort in debug mode (y/n): ';
    $handle = STDIN;
    $debug = strtolower(trim(fgets($handle))) === 'y';

    $sorter = new MySort($debug);
    $sorter->QuickSort($arr, 0, count($arr) - 1);
    echo 'And the sorted input is: ' . PHP_EOL;
    echo flattenArr($arr);
}

class MySort {
    private
        $debug = false;
```

```

function __construct($debug) {
    $this->debug = $debug;
}

function QuickSort(&$arr, $start, $end) {
    if ($start <= $end) {
        // calling partition function
        $pIndex = $this->Partition($arr, $start, $end);
        $this->QuickSort($arr, $start, $pIndex - 1);
        $this->QuickSort($arr, $pIndex + 1, $end);
    }
}

function Partition(&$arr, $start, $end) {
    $pivot = $arr[$end]; // pivot here is the actual value
    if ($this->debug) {
        echo 'Sorting the array from positions ' . $start . ' to ' . $end . ' . ' . PHP_EOL;
        echo 'The pivot is at position ' . $end . ' and has the value ' . $pivot . ' . ' . PHP_EOL;
    }

    // set the partition index as start initially
    $pIndex = $start;
    for ($i = $start; $i < $end; $i++) {
        // swap if element is smaller than pivot
        if ($this->debug) {
            echo flattenArr($arr);
            echo 'The value at position ' . $i . ' is ' . $arr[$i] . ' . ' . PHP_EOL;
        }
        if ($arr[$i] <= $pivot) {
            if ($this->debug) {
                echo 'This value is smaller than or equal to the pivot, so it will be swapped with the index, which is at position ' . $pIndex . ' . ' . PHP_EOL;
            }
            if ($i == $pIndex) {
                if ($this->debug) {
                    echo 'However, the position is the index, so there\'s nothing to swap.' . PHP_EOL;
                }
            } else {
                $temp = $arr[$i];
                $arr[$i] = $arr[$pIndex];
                $arr[$pIndex] = $temp;
            }
            $pIndex++;
        }
    }

    // Perform swapping operation with the element at the partition index
    if ($this->debug) {
        echo 'And now the pivot will be swapped with the index.' . PHP_EOL;
    }
    if ($pIndex == $end) {
        if ($this->debug) {
            echo 'However, the index is the pivot, so there\'s nothing to swap.' . PHP_EOL;
        }
    } else {
        $t = $arr[$pIndex];
        $arr[$pIndex] = $arr[$end];
        $arr[$end] = $t;
    }
    return $pIndex;
}

```

```
}
```

```
manage();
```

Conclusion

This was a fascinating exercise looking “under the hood” of the higher-level languages with which we are more familiar, and a useful reminder that computers are, fundamentally, bit-twiddling devices, not magic.

The benefit of higher level languages is that they abstract away a lot of this complexity, allowing us to work on more complex systems. It is good, however, to be at least somewhat aware of what’s going on at lower levels of the stack. The abstractions can — and occasionally do — leak.

References

- Harper, D. and Felix, T. (no date) "algorithm (n.)". Available at: <https://www.etymonline.com/word/algorithm> (Accessed: October 22, 2025).
- HarperCollins (2023) "algorithm". Available at: <https://www.collinsdictionary.com/dictionary/english/algorithm> (Accessed: January 24, 2026).
- Hoare, T. (1961) "Algorithm 64: Quicksort," *Communications of the ACM*, 4(7), p. 321. Available at: <https://dl.acm.org/doi/pdf/10.1145/366622.366642> (Accessed: January 30, 2026).
- kartik (2025) *ArrayList to Array Conversion in Java : toArray() Methods*. Available at: <https://www.geeksforgeeks.org/java/arraylist-array-conversion-java-toarray-methods/> (Accessed: February 2, 2026).
- Kartik (2025) *Selection Sort*. Geeks for Geeks. Available at: <https://www.geeksforgeeks.org/dsa/selection-sort-algorithm-2/> (Accessed: January 30, 2026).
- Paidipati, P. (2023) *QuickSort Complete Tutorial | Example | Algorithm | Programming | Complexity*. CSEstack.org. Available at: <https://www.csestack.org/quicksort> (Accessed: January 30, 2025).
- PHP (2023) *PHP: sort*. Available at: <https://www.php.net/manual/en/function.sort.php> (Accessed: February 3, 2026).
- PHP (2023) *Zend sort*. Available at: https://github.com/php/php-src/blob/master/Zend/zend_sort.c#L248 (Accessed: February 3, 2026).
- Pombrio, J. (2024) "Typst as a Language," *Justin Pombrio*, 30 November. Available at: <https://justinpombrio.net/2024/11/30/typst.html> (Accessed: October 22, 2025).
- Sanderson, G., Schloss, J. and Way, R. (2018) *But what is the Fourier Transform? A visual introduction*. Available at: <https://www.3blue1brown.com/lessons/fourier-transforms> (Accessed: January 25, 2026).
- Shilimkar, P. (2013) *Why Selection sort can be stable or unstable*. Available at: <https://stackoverflow.com/a/20761529> (Accessed: February 9, 2026).
- Spolsky, J. (2001) *Back to Basics*. Joel on Software. Available at: <https://www.joelonsoftware.com/2001/12/11/back-to-basics/> (Accessed: February 3, 2026).
- Spolsky, J. (2002) *The Law of Leaky Abstractions*. Joel on Software. Available at: <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/> (Accessed: February 3, 2026).
- Wild, S. (2013) *Quicksort Partitioning: Hoare vs Lomuto*. Available at: <https://cs.stackexchange.com/q/11458> (Accessed: January 30, 2026).