Contents

1. General information

What is an algorithm? An algorithm describes the steps one has to perform to accomplish a specific task. For example below is one algorithm for making tea.

Making Tea.

1. Boil water
2. Add a teabag to a cup.
3. Pour the boiling water into the cup
4. Remove the teabag when tea is ready (meaning it is steeped to the desired degree – either weak tea, strong tea or tea in between)
5. Add the desired amount of milk to the cup
6. Add the desired amount of sugar to the cup
7. Stir the contents of the cup with a spoon

And tea is ready!

But one may look at this and thinks "Wait a minute that is not how I make tea. I do not put the teabag in the cup, I put teabag into a teapot. And I boil the water and I pour the water into the teapot. And then I add milk into the cup before I pour from the teapot into the cup, because that burns the milk, and it gives the tea a little bit of a different taste." But it was said here is "one" algorithm for making tea, not here is "the" algorithm for making tea.

There can be more than one algorithm for accomplishing a task. There are many ways one can sort data and so there are many sort algorithms. There is not just one sort algorithm, and this is the one and only one way one can sort data. The making tea algorithm makes some assumptions that assume that one uses a teabag and makes the tea directly in the cup, rather than using a teapot. And it is not unusual for an algorithm to make assumptions. Some of the sort algorithms make assumptions about the data they are sorting and some do not. There can be many algorithms that accomplish the same task.

There is one really important distinction that one has to understand – an algorithm is not an implementation, they are not the same thing. An algorithm describes the steps one has to perform; an implementation is the code one writes to perform those steps if it comes to programming. There can be many implementations of the same algorithm. For example, look at the tea making algorithm again. In step one it is not specified how to boil the water, it says "boil water", and so it does not matter if the water is boiled in the kettle or in a saucepan. As long as water was boiled step one was performed. In the same way when an algorithm is implemented it may be possible to implement a step in many different ways. At this report I will describe one implementation of each algorithm that I chose for this project, but there can be, and likely are, other implementations of the algorithm. That is the algorithms; they just describe the steps one has to perform to accomplish a specific task.

It is really helpful to be able to compare the performance of one algorithm against another algorithm. One might think that the one way to do that is to implement one algorithm and then put a line of code that records the start time and run the implementation. Then have a line of code that records the end time and subtract the start time from the end time and to get the running time of the implementation of that algorithm Do the same for another algorithm and just compare the two. It sounds great in theory but unfortunately that's not a good way to do it because of hardware.

Hardware will influence the running time of these algorithms. Think about it, if to run an implementation on a desktop computer that was built in 2020 and compare that to the running time of the exact same implementation on a desktop computer that was built 20 years ago and perhaps is an old Pentium computer, or even go further than that and run the implementation on a Commodore 64 or something like that, it is obviously the implementations are going to run slower on the older hardware. And if to run an

implementation on a super computer, it is going to run really, really fast even though it might be a really inefficient algorithm. A more objective measure is needed than just the straight running time. Look at the number of steps that it takes to execute an algorithm. It is called the time complexity.

There are two types of complexity:
- time complexity which is the number of steps involved to run an algorithm;
- memory complexity which is the amount of memory it takes to run an algorithm.

These days, because memory is so cheap, memory complexity is not such an issue. The main concern is about time complexity. In how many steps does it take to run an algorithm and what will be the worst case? Looking at the best case does not help because the best case is rare. It is possible to take the average case but that will not tell the absolute worst time complexity. If one wants to know what the upper bound is, like what is the absolute worst that one can expect from this algorithm, it is much more helpful to look at the worst case. It is helpful to compare the worst case scenario for one algorithm against the worst case scenario for the other algorithm. Look at the algorithm below - an algorithm for adding sugar to tea.

Add Sugar to Tea

1. Fetch the bowl containing the sugar
2. Get a spoon (assuming it is loose sugar)
3. Scoop out sugar using the spoon
4. Pour the sugar from the spoon into the tea
5. Repeat steps 3 and 4 until you have added the desired amount of sugar (And if you want more than one teaspoon then you have to repeat steps three and four until you have added the desired amount of sugar)

As can be seen from this algorithm the number of steps it will take to add sugar to ones tea will depend on how many sugars one want. If only one spoon of sugar is needed, then this algorithm will run taking four steps. But if two spoons of sugar are needed, it will take six steps because one has to repeat steps three and four. Have a look at what happens for the number of spoons. If it is one spoon, just do four steps. With two spoons, it will take six steps. With three spoons in the tea it will be eight steps because it is necessary to repeat steps three and four three times. With four spoons, repeat steps 3 and 4 four times. It will take 10 steps to put sugar in the tea.
The time complexity give an idea of how an algorithm will perform as the number of items it has to deal with grows. One can see as the number of spoons this algorithm has to add to tea increases, the number of steps required increases.

Number of Sugars	Steps Required
1	4
2	6
3	8
4	10

Table 1: Number of Sugars

Another way of saying this is it tells how well an algorithm will scale. How well will it do when it has to deal with 100 items, versus 1000 items, versus a million items. Some algorithm will scale really well and others not so well. The more items there are, the more algorithm's performance will degrade. The big O notation is a way of expressing the complexity related to the number of items that an algorithm has to deal with. Work out the big O value for the sugar algorithm.
It is conventional to designate the number of items by n.

- Number of desired spoons of sugar = n

- Total number of steps = 2n + 2 (the reason for this is that one have to repeat steps three and four for the number of spoons one wants that is 2n. And then the plus 2 is steps one and two)
- As n grows, the number of steps grows
- The "2" in 2n and the "+2" remain constant; it does not changes as the number of spoons changes, so it does not factor into the time complexity. Because of that do not consider them when come up with the big O value. The value of n determines the growth rate; it is n that is influencing the number of steps.
- Time complexity is O(n)
- Linear time complexity

If go back to Table 1, it is seen that it goes 4, 6, 7, 10 and of course it will keep going 12, 14, 16, 18. That is a linear progression, it is a linear sequence. The time complexity of the adding sugars to tea algorithm is O(n).
See the big O values in Table 2; the order is the best to worst.

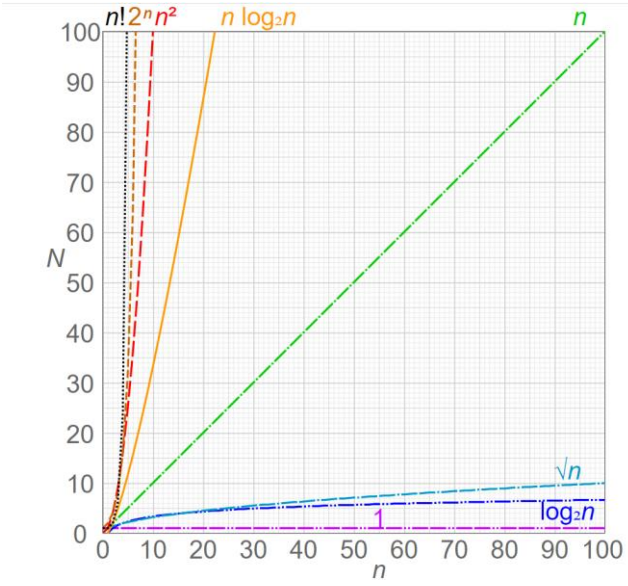


Figure 1: Graphs [2]

Big-O	
O(1)	Constant
O(logn)	Logarithmic
O(n)	Linear
O(nlogn)	n log-star n
O(n²)	Quadratic

Table 2: Big-O

At Wikipedia [2] there is the image (created by MC), this is a graph of some big O values and this represents how an algorithm would degrade.

On Figure 1 there are graphs of functions commonly used in the analysis of algorithms, showing the number of operations N versus input size n for each function.

Along the X axis there is the input size (n), that is the number of items. And along the Y axis there is the number of steps (N). It is seen a constant algorithm O(1) which is in violet color as the number of items increases the number of steps remains the same. This is just a straight horizontal line. That is the best case. If one can get a constant algorithm that is pretty much the best one can do.

The next best thing is logarithmic, notice that is $\log_2 n$ and this is in a dark blue. As one can see it is like kind of starts of climbing as the number of items increases the number of steps goes up. But then it levels off and it is pretty much almost constant time. If one can achieve $\log_2 n$, that is great.

Then there is a green graph which is more pattern of growth, if there are10 items, it will take 10 steps, 20 items – 20 steps, so it is a linear sequence. This is what it looked like for the adding sugar to tea algorithm. It does not start at zero steps (but if one does not want no sugar, obviously there is zero steps), but our algorithm is for adding sugar to one's tea, so that assumes that there should be sugar added. The minimum number of steps for one sugar is four. One would actually start at four and then go six, eight, ten, et cetera. And it is a linear progression.

The orange line is $n\log_2 n$. Here $\log_2 n$ is multiplied by the number of items. This is worse than linear because the number of steps required climbs much more sharply. Worse than that is quadratic (n^2). This is an even sharper climb. It is the red dashed line. If there are 10 items it will take 100 steps, because that will be 10^2. A quadratic algorithm quickly degrades. A thousand items is already a million steps.

The absolute best is constant time. Next is logarithmic time $\log_2 n$. Linear time is still pretty good, but once it starts getting up into $n\log_2 n$ and n^2 one can see that the number of steps required rises sharply as the number of items increase. In conclusion this is what one have to remember really is a big O notation gives a way of comparing the time complexity of different algorithms

Researching algorithms allows creating useful sets of methods for solving some scientific problems. This way allows to understand which algorithms are most effective in different cases so that it is possible to choose the algorithm that works best for specific programs. An algorithm that performs well on a one set of data may perform terribly with other data, so it's important to know how to choose the algorithm that will be the best match for the scenario.

A good algorithm must have three features [1]: correctness, maintainability, and efficiency. Clearly if an algorithm doesn't resolve the problem for which it was worked out, it's not very helpful. If it doesn't give correct answers, there's little point in using it.

2. Introduction to Sort Algorithms

Sorting is the process of ordering a certain set of elements, on which the order relations >, <, >=, <= are defined (in ascending or descending order). Sorting algorithms have a lot of practical applications. They can be found almost everywhere where it comes to processing and storing large amounts of information. Some data processing tasks are easier to solve if the data is organized.

2.1 Bubble Sort

Let us look at sort algorithms by starting with Bubble sort.
I want to mention right up front that this algorithm's performance degrades quickly as the number of items one need to sort grows, but it is a commonly taught algorithm. I would like to start out by explaining how this algorithm works.
There is an array of length 7, so seven integers can be stored. I will use Bubble sort to sort this array.

The way the Bubble sort works is as the algorithm progresses, it partitions the array into a sorted partition and then an unsorted partition. This is a logical partitioning one does not create completely separate array instances. This array instance contains what have been sorted so far and another array instance contains unsorted numbers. The array is partitioned logically. To start the algorithm for this array, there will be a field that will be called partition index. When one starts this will be index 6, because the entire array is the unsorted partition, because nothing was sorted yet. 6 will be the last index of the unsorted partition. The implementation starts at the left of the array, or at index 0, that is what i is and it is necessary to compare the elemented index 0 with the elemented index 1. If the elemented index 0 is greater than the elemented index 1, swap the elements. If it is less do not do anything, because it is necessary to move larger elements towards the end of the array or towards the right because the sorting is in ascending order. In

this case 20 is less than 35, so do not do any swapping. Then increment i to 1. Look at the element at position 1 and compare it with the element at position 2. In this case 35 is greater than -15, swap them and increment i to 2.

Compare element at index 2 with the elemented index 3. 35 is greater than 7, swap them and increment i to 3.

Compare the elemented index 3 with the elemented index 4. 35 is less than 55 do not do anything. Just increment i to 4 and then compare 55 to 1, that is the elemented position 4 with the elemented position 5. 55 is greater than 1, swap them and increment i to 5.

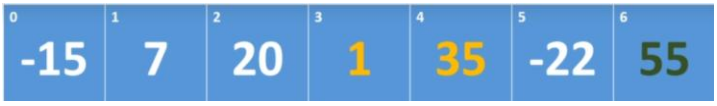
Finally compare 55 to -22 and of course 55 is greater than -22, swap them and at this point i is equal to the last unsorted partition index so stop.

The first traversal of the array has been completed and at the end of that traversal, the last element in the array is in its correct position. At this point the greatest element in the array will be at index array.length minus 1 which for this array is length minus 1 it is 6. Now set the unsorted partition index to 5 because now position 6 is considered to be sorted and the logical partition is going to be between the elemented index 5 and the elemented index 6. Everything from index 5 down to the front of the array is in the unsorted partition. Everything at array index 6 and to the right (in this case there is nothing to the right) is in the sorted partition.
On the next traversal it is seen that 55 is now in the sorted partition and i is sat back to 0, and the unsorted partition index is now 5.

Start all over again, the element at array index 0 is greater than the element at array index 1, swap them and increment i to 1.

Watch if the array element at index 1 is greater than the array element at index 2? Yes it is. Swap them and increment i to 2 .

Is the element at index 2 greater than the elemented index 3? No it is not, so just increment i to 3. Compare the elemented index 3 to the elemented index 4. 35 is greater than 1, swap them and i get incremented to 4.



35 is greater than -22, swap them. I gets incremented to 5 and now i equals to the unsorted partition index so stop.

At this point 35 and 55 are in their correct positions and the unsorted partition index is set to 4. Everything in array from index 0 to array index 4 is in the unsorted partition and everything from array index 5 up to the end of the array is in the sorted partition. Set i back to 0, because the process should be repeated.

I will not go through the other traversals, they will operate the exact same way. Keep incrementing i and comparing the elemented i with its neighbor. Swap the elements if its neighbor is less than the elemented i, until the sorted partition is the entire array and then it is done.
This is called a bubble sort because the larger values in the array will bubble up to the end of the array. Another way of looking at it is to flip the array vertically, take the array and flip it counterclockwise so that array element zero is at the bottom and array element six is at the top. And another way of looking at it is saying that the larger values are bubbling up to the top of the array.

Performance of Bubble Sort

- ✓ In-place algorithm
- ✓ $O(n^2)$ time complexity – quadratic
- ✓ It will take 100 steps to sort 10 items, 10 000 steps to sort 100 items, 1 000 000 steps to sort 1 000 items
- ✓ Algorithm degrades quickly
- ✓ Stable sort algorithm

First of all, it is an in-place algorithm. What does it mean? As was mentioned before, the array is logically partitioned. No need to create another array in order to perform this sort. It is an in-place algorithm, if the extra memory that it needs does not depend on the number of sorting items. For example create a few local variables to store the last sorted partition and to store i, but that does not mean it is not an in-place algorithm. If the extra memory one is using does not increase as the number of items in the array increases, then it is still an in-place algorithm. It is OK to use a few extra variables. Bubble sort algorithm has $O(n^2)$ time complexity. It is a quadratic algorithm, that means it will take 100 steps to sort 10 items and so on. As it is seen this algorithm really degrades quickly.

How was $O(n^2)$ got? Absolute worst case is n^2 steps, where N is the number of sorted items. But looking at the code (see implementation part) one could it is not n^2, it is actually less than that because the sorted partition is growing after each iteration of the outer loop and the inner loop does not cross into the sorted partition. So the inner loop is actually doing less work as the algorithm progresses, or at least as this implementation progresses, and that is right. But remember when it comes to determining the time complexity of an algorithm, it is not math. It is not the absolute precise expression. It is some sense of how the number of steps grows as the number of items to be sorted grows. It is kind of a general idea. It does not grow as the number of items grows. The algorithm grows in a quadratic way as the number of items increases. The pattern is not linear, it is not logarithmic, and it is certainly not constant. One tip when it comes to determining time complexity is to look at how many loops there are, because normally each loop corresponds to n. If there is only one loop then it is linear time complexity. If there are two loops, then it is n times n, that is quadratic time complexity. So that was a Bubble sort algorithm – one of the least efficient algorithms.

At this point I would like to introduce the concept of stable versus an unstable sort. When it comes to sort algorithms, there are stable and unstable sorts. What does this mean? Stable versus unstable comes into play when there are duplicate values in the data that should be sorting. In example below an array contains two nines (at position one and three).

Unstable Sort (unsorted)

The question is during the sorting will the original ordering of the two nines be preserved? In other words, in the sorted array will the “white nine” still come before the “black nine” or will their positions have changed so that the “black nine” comes before the “white nine”? If a sort is unstable, then that means the relative ordering of duplicate items will not be preserved. And in an unstable sort, the “black nine” will end up coming before the “white nine”. What will happen is the nines are sorted now, the array is sorted, but the two nines have flipped position. Their relative ordering has not been preserved.

Unstable Sort (sorted)

In the original array, the “white nine” came before the “black nine”. And in the sorted array the “black nine” is coming before the “white nine”. When this happens, when the relative ordering of duplicate items is not preserved, it is considered to be an unstable sort.
By contrast for a stable sort, let us start off with the same array, but after it be sorted, the “white nine” still appears before the “black nine”, so the relative ordering of the duplicate items has been preserved and in this case it is a stable sort.

Stable Sort (unsorted)

Stable Sort (sorted)

All other things being equal, a stable sort preferable to an unstable sort. But one might say “Well, who cares if the relative ordering of the nines is a flip position?” For integers it does not really matter. But if to sort objects it could make a difference depending on how to use it. For example, if to do a sort within a sort, one may first sort based on name and then sort based on age or something else. If the second sort causes the positions one got from the first sort to flip, that will be a problem.

What about Bubble sort, is it a stable or an unstable sort algorithm? Look at the code and think how Bubble sort works. The answer is it is a stable sort algorithm, and why is that? When adjacent elements are compared, they are only swapped if the element at i is greater than the element at i plus 1. And when those two “nines” end up next to each other (and they will eventually), the “white nine” will end up at position i and the “black nine” will end up at position i + 1 and when we compare them, because nine is not greater than nine, than do not swap them, and their positions remains the same, the relative ordering. If the algorithm said or implementation said greater or equals, swap them and that would be an unstable sort but nobody wants to turn a stable sort into an unstable sort, despite the fact that it is really easy to do. There are a lot of codes where a stable sort algorithm has been coded to be unstable. That pesky little equals can make a big difference. Be aware of this writing own code. Make sure that if a sort algorithm is stable that the implementation is not inadvertently changing it to an unstable. In a nutshell, a stable sort algorithm preserves the relative ordering of duplicate items and an unstable sort algorithm does not.

2.2 Selection Sort

Let us look at the Selection sort algorithm. This algorithm divides the array into sorted and unsorted partitions just like Bubble sort does. One traverses the array and looks for the largest element in the unsorted partition. Then swap it with the last element in the unsorted partition. At that point, that swapped element will be in its correct sorted position. Just like with Bubble sort, at the beginning of the algorithm, the entire array is unsorted so the last unsorted index is 6. Like with Bubble sort the sorted partition will be grown from right to left.

Initialize a largest field to 0, at the beginning 20 is the largest element that one knows about so far, whatever is at position zero will be the largest element. Compare the element at position one to whatever is at position zero so start with i equal to one. Use i to traverse the unsorted partition and find the largest element.
Compare 35 to 20; 35 is larger than 20 and because change largest to one and increment i to two, and compare -15 to the largest element which is now at position one. It is obvious that -15 is less than 35, so only increment i to three.
Compare 7 to the largest element which is still at position one. 7 is less than 35, so just increment i to four.
Now compare 55 to 35 and 55 is greater than 35, change largest to four, because the largest element that is found so far in the unsorted partition is at index four and increment i to five.
Compare 1 to 55. 1 is less than 55, just increment i to six.
Then compare -22 to 55. -22 is less than 55, do not do anything. Now i is equal to the last unsorted index and the first traversal of the array is completed. The largest element that was found in the unsorted partition was swapped and that is at position four with the last element in the unsorted partition and that is at position six. And what we are doing is swap 55 and -22 and the first traversal is completed.

Now 55 is in its correct sorted position. Next to is to decrement last unsorted index to five and re-initialize i to one and say that the largest element in the unsorted partition is at position zero.

Repeat the process. Compare 35 against 20. 35 is greater than 20; set largest to one. Increment i to two and compare -15 against 35, because 35 is now the largest element. -15 is less than 35; just increment i to three. 7 is less than 35; increment i to four. -22 is less than 35; increment i to five and of course 1 is less than 35. Now i equals the last unsorted index, the second traversal of the array is completed. the largest element, which is at position one, was swapped with the last element in the unsorted partition, which is at index five, so 35 and 1 was swapped.

35 and 55 are in their correct sorted positions.

Decrement the last unsorted index and now the last unsorted index of the unsorted partition is four. Re-initialize largest to zero and that should be 1. Re-initialize i to one and repeat the process. Compare 1 to 20, largest is going to remain zero, et cetera, until the entire array is sorted. This is how Selection sort works. It is called Selection sort because on each traversal it is selecting the largest element and removing it into the sorted partition.

Performance of Selection Sort

- ✓ In-place algorithm
- ✓ $O(n^2)$ time complexity – quadratic
- ✓ It will take 100 steps to sort 10 items, 10 000 steps to sort 100 items, 1 000 000 steps to sort 1 000 items
- ✓ Does not require as much swapping as Bubble sort
- ✓ Unstable algorithm

Selection sort is an in-place algorithm. It does not use any extra memory. As was mentioned with Bubble sort, it is OK to use a few extra fields as long as the extra memory one is using does not depend on the number of items one is sorting.
Time complexity is $O(n^2)$ because there are n elements in the array and for each element n elements are traversed, so it is quadratic. However, it does not require as much swapping as Bubble sort. There is only one swap per traversal, so Selection sort will usually perform better than bubble sort. It is used 'usually' because there can be an array that is almost sorted, then Bubble sort does not have to swap that much. But in the average case, all other things being equal, generally selection sort will perform better.
However, Selection sort is an unstable algorithm, because if there are duplicate elements there is no guarantee that their original order relative to each other will be preserved, because on each pass the largest element is swapped with whatever is occupying the last position in the unsorted partition, and it is very possible that the second duplicate value could be taken and moved in front of its twin. That is why

Selection sort is an unstable algorithm. If there is necessary to use a stable sort algorithm, then Selection sort should not be used.

2.3 Insertion Sort

Next algorithm is Insertion sort. Like the other discussed algorithms it partitions the array into sorted and unsorted partitions. But this time the implementation grows assorted partition from left to right. So it grows a sorted partition from the front of the array.

firstUnsortedIndex = 1 – this is the first index of the unsorted partition
i = 0 – index used to traverse the sorted partition from right to left
newElement = 35 – the value we want to insert into the sorted partition – array[firstUnsortedIndex]

How does insertion sort work? It starts out by saying that the element position zero is in the sorted partition. And because this sorted partition is of length one, by default, the element is sorted. There is only one element. At the beginning, the elements from position one into the right are in the unsorted partition. Set a first unsorted index field to one. On each iteration, take the first element in the unsorted partition which will be the element at array of first unsorted index, and insert it into the sorted partition. At the end of each iteration this sorted partition should be grown by one. On the first iteration take 35, because that is the first unsorted value. Insert it into the sorted partition. At the end of the iteration, the first two elements in the array will be sorted. So how is each element inserted? Well, compare the inserting value with the values in the sorted partition. Traverse the sorted partition from right to left, and look for a value that is less than or equal to the one that should be inserted. Once the value is found, stop looking, the correct insertion position for the new value that one is trying to insert is found. Remember, when the value is inserting, the work is being done within the sorted partition. If the element at index i in the sorted partition is less than or equal to the element that should be inserted, then all of the values to the left of element i will be less than or equal to the value that should be inserted, because all the values are in sorted order. Looking for the correct insertion position, values are shift in the sorted partition to the right. On the first iteration assign 35 to a new element field, because 35 is the first element in the unsorted partition. Then use i to traverse the sorted partition from right to left. Compare 35 to 20. 35 is greater than 20. 35 is already in its correct position in the sorted partition. It is not in its correct position in the array, but in the sorted partition. After the first iteration, disordered partition has been grown to two, lengths two and the first two elements are in their correct position.

firstUnsortedIndex = 2
i = 1
newElement = -15

Now the first unsorted index is at index two and i is assigned one, because that is the right most index in the sorted partition. Assign -15 to new element. Insert -15 into the sorted partition. Compare -15 against 35. -15 is less than 35. Shift - 15 down the array. But another way of looking at it is to shift 35 to the right to make room for -15. Assign 35 to position two. And do not worry that -15 might be overwritten because it is not saved in a new element field.

firstUnsortedIndex = 2
i = 0
newElement = -15

Compare -15 to 20. -15 is less than 20, shift 20 to the right to make room for -15.

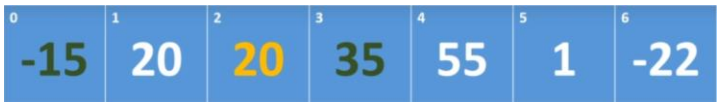
At this point, it hits the front of the array, there is nothing else to compare -15 to. Basically it is the smallest element in the sorted partition and because one has hit the front of the array, this is where -15 is inserted.

firstUnsortedIndex = 3
i = 2
newElement = 7

The second iteration is ended. The sorted partition is grown to three. The first three elements in the array, which is a sorted partition, are in their correct positions in the sorted partition. Now the first unsorted index is at position three. Assign the value of 7 to new element, and compare 7 against 35. 7 is less than 35. We shift 35 to the right.

firstUnsortedIndex = 3
i = 1
newElement = 7

Compare 7 against 20, 7 is less than 20, shift 20 to the right and then compare 7 against -15.



firstUnsortedIndex = 3
i = 0
newElement = 7

7 is greater than -15. The insertion position is found. Remember, the work is within the sorted partition. If there was anything to the left of -15, all those values would be less than -15. There is no need to keep traversing the sorted partition. The moment an element that is less than or equal to the one is hit try to insert, that correct insertion position is found. Insert 7 into position one. Another iteration is completed.

firstUnsortedIndex = 4
i = 3
newElement = 55

The sorted partition is grown by one. All of the elements in the sorted partition are sorted. The next element to insert is 55. Compare 55 against 35, 55 is greater than 35. It means its correct position has already been found. This iteration is completed.

firstUnsortedIndex = 5
i = 4
newElement = 55

The first five elements in the array are in their correct sorted position. Now 1 is needed to be shifted all the way down to position one, and then -22 will be shifted all the way down to position zero. That is the insertion sort – an in-place algorithm.

Performance of Insertion Sort

- ✓ In-place algorithm (do not need to create any temporary arrays)
- ✓ $O(n^2)$ time complexity – quadratic
- ✓ It will take 100 steps to sort 10 items, 10 000 steps to sort 100 items, 1 000 000 steps to sort 1 000 items
- ✓ Stable algorithm

It is stable, because when the picking of elements, is done from left to right. Say there are two nines in the array. Insert the left most nine first. And when one comes to the right most nine (remember, when one is looking for the insertion position, one stops when one hits an element that is less than or equal to the one that one is inserting), one eventually hit the first nine that was inserted into the sorted partition. The second nine will be inserted to the right of the first nine. The relative position of those two nines will be preserved.

2.4 Shell Sort

Insertion sort is a quadratic algorithm (see the previous chapter). But if the sequence of values that one sorts is nearly sorted, then Insertion sort runs in almost linear time. It happens because it does not have to do as much shifting. A computer scientist named Donald Shell realized that if one could cut down on the amount of shifting, than insertion sort would run a lot faster. He came up with something that is known as the Shell sort algorithm. How does Shell sort work? Well, it is a variation of Insertion sort. Insertion sort chooses which element to insert using a gap value of one. Every time Insertion sort runs, it picks off the first unsorted value and compares that value to its neighbor and it keeps shifting the neighbors to the right until it finds the correct insertion point for the element that it is inserting. Shell sort starts out using a larger gap value. Instead of comparing elements to their neighbors it compares elements that are further apart from each other in the array. As the algorithm runs it reduces the gap that it is using. The goal is to reduce the amount of shifting required. As the algorithm progresses the gap value is reduced. Shell sort traverses the array with a certain gap value and after it has done its first traversal with the initial gap value, it decreases the gap and it does it again, this is very important. It keeps reducing the gap value until the gap value is one. When the gap value is one, essentially an Insertion sort is being done. The last iteration of the gap value will actually perform an Insertion sort. But at that point, the array will be more

sorted than it was at the beginning. Essentially what Shell sort does – it does some preliminary work using gap value that is greater than one, and preliminary work puts the elements in the array and perhaps closer to their sorted positions. At the very last iteration when the gap value becomes 1 it does an Insertion sort. That final Insertion sort will work with values that have had some preliminary sorting done on them. Because of that, there will be a lot less shifting required.

What is used for the gap value? How to figure out what to start with and how to reduce it? There is a tonne of theories about this. At the Wikipedia article [3] about Shell sort there is a table with different initial gap values and how to reduce the gap, what sequence of gap values to use. There are a number of ways to calculate the gap value. The important thing to note is that the way to calculate the gap, can influence the time complexity. Depending on what gap is used the time complexity is different.

One really common sequence that is used for the gap value (is also called the interval) is the Knuth sequence. (in implementation the Knuth sequence is not used)

k	Gap (interval)
1	1
2	4
3	13
4	40
5	121

Table 3: Knuth Sequence

The Knuth sequence says that

- Gap is calculated using $(3^k - 1) / 2$
- k is based on the length of the array
- The gap should be as close as possible to the length of the array one wants to sort, without being greater than the length

In the implementation below something simpler is done. The gap is based on the array's length. Initialize the gap to array.length / 2. On each iteration divide the gap value by 2 to get the next gap value.

i = gap = 3
j = i = 3
newElement = intArray [i] = 7
Compare intArray [j – gap] with newElement

There are only seven elements in the present array. The first gap value will be at 7 / 2, which is 3, the second gap value will be 3 / 2 which is 1. Actually this array will have only two iterations. On the first iteration, a gap value of three is used. On the final iteration, this is always true for Sell sort, the gap value will be 1. At that point an Insertion sort start. But because a previous iteration was done and some of the elements were moved around, some of the elements will be closer to their sorted positions. Let's go through this. Initialize i to the gap and j to i. As with Insertion sort, save the value that you want to work with into newElement. Then compare the element at index j minus gap, that will be 3 minus 3 is 0 with newElement. The gap is three, start with element, we basically want compare it. Because the gap is three, compare it to the element that it three positions away, so we compare 7 with 20. 7 is less than 20, assign 20 to where 7 was. Instead of doing what was done with Insertion sort (that was comparing to the neighbours and shifting up one), in Shell sort compare using a gap of three and shift by three. 20 has been shifted up three places.

i = gap = 3
j = j – gap = 0
newElement = intArray [i] = 7

Then set j to j minus gap which is 0. At this point hit the front of the array. Assign newElement to position zero.

What was basically done it is kind of Insertion sort, but with a gap of three. Next move i to 4, and j become i which is 4.
i = 4
j = i = 4
newElement = intArray[i] = 55
Compare intArray [j-gap] with newElement

The newElement is 55 and compare 55 to 35 because that is three elements away. 55 is greater than 35, do not do anything. Everything remains as it is. Now i will be 5, j will be 5, we compare 1 to -15 (-15 is three elements away).

i = 5
j = i = 5
newElement = intArray [i] = 1
Compare intArray [j – gap] with newElement

There is nothing to do because 1 > -15. Move to -22 and assign that to newElement and compare it to the element that is three positions away from it.
i = 6
j = i = 6
newElement = intArray [i] = - 22
Compare intArray [j – gap] with new Element

-22 is less than 20, assign 20 to position six. At this point subtract the gap from j and compare -22 against what is at position zero.

i = 6
j = j – gap = 3
newElement = intArray [i] = -22
Compare intArray [j – gap] with newElement

-22 is less than 7 so shift into position three. At this point hit the front of the array.

i = 6
j = j – gap = 0
newElement = intArray [i] = -22
Compare intArray [j – gap] with newElement

There are no more elements to compare -22 against. Assign -22 to position zero. At this point hit the end of the array and finish the first iteration with gap equal to three.

Notice that the array is more sorted now than it was before. -22 was moved all the way to the front of the array and it was done with one assignment. Also 20 was moved closer to its sorted position. 20 started at the front of the array from where it usually ends up in the sorted array. One can see how doing this preliminary work before get to Insertion sort will cut down on how much shifting one has to do.
Now update the gap. We divide 3 by 2 it is 1 and do an Insertion sort, because the gap is going to be one everything to its neighbours will compared. Shifting will be up by 1 and do an Insertion sort, but it is an Insertion sort on an array that has had some preliminary work done on it, so there will be a lot less shifting and that is what Shell sort is all about.

Performance of Shell Sort

- ✓ In-place algorithm (like Insertion sort)
- ✓ Difficult to define the time complexity because it will depend on the gap (on the method that one is using to choose the gap). Worst case: $O(n^2)$, but it can perform much better than that
- ✓ Does not require as much shifting as Insertion sort, so it usually performs better than insertion sort
- ✓ Unstable algorithm (but Insertion sort is stable)

Shell sort is unstable because in the pre-insertion sort phase when we are comparing elements that are far away from each other, it is very possible that if there is a duplicate element, the rightmost element will be shifted in front of the leftmost element. The fact that one exams elements that are further away from each other can lead to the relative positions of duplicate items not being preserved.
Shell sort can be applied to Insertion sort and Bubble sort and both algorithms are improved that way. It would be the same type of idea – using a gap interval. In this case rather than just shifting elements, they would actually be swapped. With Insertion sort the number of shifting are cut down. With Bubble sort the number of swaps is cut down.

2.5 Merge Sort

- Divide and conquer algorithm (because it involves splitting the array one wants to sort into a bunch of smaller arrays)
- Recursive algorithm
- Two phases: Splitting and Merging (The sorting is done at the merging phase.)
- Splitting phase leads to faster sorting during the Merging phase
- Splitting is logical. We do not crate new arrays (Indices are used to keep track of where the arrays has been split.)

The Divide and Conquer strategy is based on breaking down the task into smaller chunks.

Merge Sort – Splitting Phase

- Start with an unsorted array
- Divide the array into two arrays, which are unsorted. The first array is the left array, and the second array is the right array (Generally just divide the array down to the middle.)
- Split the left and right arrays into two arrays each
- Keep splitting until all the arrays have only one element each – these arrays are sorted

Merge Sort – Merging Phase (say it is an array of four elements)

- Merge every let/right pair of sibling arrays into a sorted array
- After the first merge, there will be a bunch of 2-element sorted arrays
- Then merge those sorted arrays (left/right siblings) to end up with a bunch of 4-element sorted arrays
- Repeat until there is a single sorted array
- Not in-place. Uses temporary array

Basically take the bunch of one-element arrays, keep merging them into two-element, four-element, eight-element, et cetera and at each merge, be sure that the resulting arrays are sorted until all the arrays are merged back into one array.

Let's analyze Merge Sort with this example

start = 0, end = 7 (array.length)
midpoint = (start + end) / 2 = 3

Elements 0 to 2 will go into the left array, end elements 3 to 6 will go into the right array
The first time the array is split elements zero, one, and two will go into the left array, and elements three, four, five, and six will go into the right array.

Let us split the left array.

start = 0, end = 3
midpoint = (start + end) / 2 = 1

Elements 0 to 0 will go into the left array, end elements 1 to 2 will go into the right array

For the left array, the start index is zero, and the end index is three. The end index is always one greater than the index of the last element in the array. The midpoint is taken by adding start to end and dividing by two, it is 3/2 which is 1. That means that the first element goes into the left array, and elements one to two go into the right array.

Now 20 is split. It can not be split any further because it is a one-element array. The right array (positions one and two) should be split.
start = 1, end = 3
midpoint = (1 + 3) / 2 = 2
Elements 1 to1 will go into the left array, end elements 2 to 2 will go into the right array.

The left array is completely split into 1-element arrays. One and two are both colored in a green shade because they are sibling arrays. The first merge that should be done on the left side will be merging 35 with -15. The last split is always merged.

Let us look at the right array.

start =3, end = 7
midpoint = (start + end) / 2 = 5
Elements 3 to 4 will go into the left array, end elements 5 to 6 will go into the right array.

Let us work on the left array the one that has 7 and 55.

start =3, end = 5
midpoint = (start + end) / 2 = 4
Elements 3 to 3 will go into the left array, end elements 4 to 4 will go into the right array.

The left array on the right side is split now. So split the right array on the right side.

start =5, end = 7
midpoint = (start + end) / 2 = 6
Elements 5 to 5 will go into the left array, end elements 6 to 6 will go into the right array.

The right array is split into 1-elements array, and once again, 7 and 55 are both shaded gold/ yellow/ brown (how one wants to look at this). They are shaded the same because they are sibling arrays. They would be merged together first, and the same applies to 1 and -22.
Now merge these arrays back together, but remember, when a merge step is performed, a sort step is actually done. Every time a merge step is performed, the resulting array is sorted.

The splitting phase is completed.

35 and -15 are in sibling left/right arrays (they will get merged)
7 and 55 are in sibling left/right arrays (they will get merged)
1and -22 are in sibling left/right arrays (they will get merged)
Every left/right array is sorted (consist of 1 element)
20 will not get merged right now because it does not have a sibling at the moment.

Because of the recursive nature of the implementation, it is important to note that the work should be done with the entire left side of the array, before the work with the right side of the array will start. The method is called to partition the left side, then the method is called to partition the right side, but that method will call itself recursively.

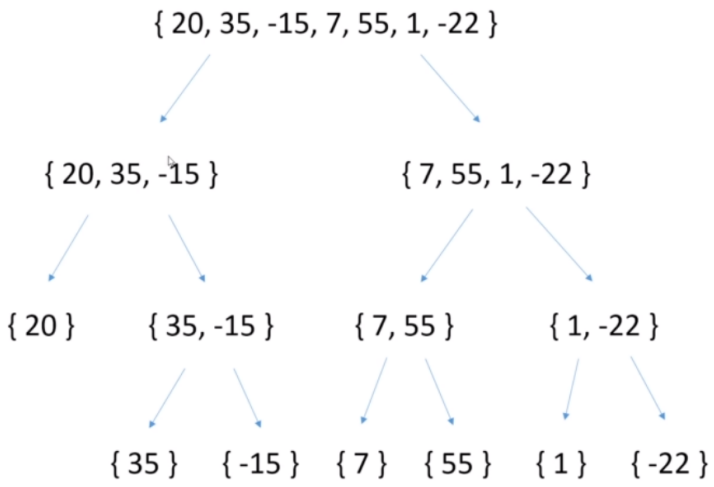


Figure 2: Splitting phase

Move to the merging step now. When the merge step is performed, the merge goes backwards, so merge bottom up. Merge sibling arrays, start by merging 35 with -15, then 7 with 55, and then 1 with -22 and only after 35 and -15 was merged back into a 2-element sorted array then merge 20 with the result, et cetera.

Now merge all these 1-element arrays

- Always merge sibling left/right arrays
- Each merge array will be sorted
- 20 does not have a sibling (it will not get merged on the first round)

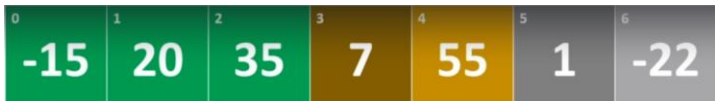
Merging process

- Merge sibling left and right arrays
- Create a temporary array large enough to hold all the elements in the arrays that are merged (On the first round, temporary arrays will be of length two, because two 1-element arrays will be merging)
- Set i to the first index of the left array, and j to the first index of the right array (two arrays were merging)
- Compare left[1] to right[j]. If the value in the left array is smaller, copy it to the temporary array and increment i by 1. If the value in the right array is smaller, copy it to the temporary array and increment j by 1. (if keep doing that, that temporary array will contain the values in sorted order)
- Repeat this process until all the elements in both arrays have been processed
- At this point, the temporary array contains the merged values in sorted order
- Copy this sorted temporary array back to the original input array, at the correct positions
- If the left array is at positions x to y, and the right array is at positions y + 1 to z, then after the copy, positions x to z will be sorted in the original array (i.e. overwrite what is there in the original array with the sorted values).

- Start by merging the two siblings on the left (35 and -15)
- Create a temporary 2-element array
- i will be initialized to 1, and j to 2
- Compare array[i] to array[j]. -15 is smaller than 35
- Copy 35 to the temporary array
- At this point, the temporary array is {-15, 35}
- Copy this temporary array back into positions 1 and 2 in the original array

The two sibling arrays have now been merged

- On the left there are two arrays instead of 3, both arrays are sorted (they are not in sorted position in the array, but the merged array is sorted and the two sibling arrays have now been merged.)
- The {20} array has a sibling {-15, 35}, merge them
- Create a temporary array of length 3
- i will be initialized to 0, and j to 1
- Compare array[i] to array[j]. -15 is smaller than 20, so it is copied to the temporary array and j is incremented by 1
- 20 is smaller than 35, so it will be copied next
- Only 35 remains, so it is copied last
- The temporary array is {-15, 20, 35}
- Copy the temporary array back to position 0 to 2 in the original array



The merging of the left sub-array is completed, it is in sorted order

- Repeat this process with the right part of the array
- 7 and 55, and 1 and -22 are two sets of siblings

- Start by merging 7 and 55, i.e. create temporary array of length 2
- i will be initialized to 3 and j to 4, compare it
- 7 is smaller than 55, so copy it to the temporary array first, and then copy 55
- The temporary array will be {7, 55}
- Copy the temporary array back to positions 3 and 4 in the original array

The left array of the right part of the original array is merged and 7 and 55 are in sorted order

- Now left array consists of 7 and 55
- Repeat this process with 1 and -22
- The temporary array will be {-22, 1}
- Copy it back into positions 5 and 6

Now there are two sorted sibling arrays of length two on the right that need to be merged

- Create a temporary array of length 4
- i will be initialize to 3, and j to 5
- -22 is less than 7, so -22 will be copied to temporary array and j will be incremented to 6
- 1 is less than 7, so 1 will be copied to the temporary array and j will be incremented to 7
- Only elements in the left array are left, and they are sorted, because all the merged arrays are sorted; just copy them in order into the temporary array
- Temporary array is {-22, 1, 7, 55}
- We copy it into the original array at position 3 to 6

All that remains now is to merge the final left and right arrays

- Create a temporary array for 7 elements
- i is initialized to 0 and j is initialized to 3
- -22 is less than -15, we copy -22 to the temporary array and increment j to 4 {-22}
- -15 is less than -1, we copy -15 to the temporary array and increment i to 1 {-22, -15}
- 1 is less than 20, we copy 1 to the temporary array and increment j to 5 {-22, -15, 1}
- 7 is less than 20, we copy 7 to the temporary array and increment j to 6 {-22, -15, 1, 7}
- 20 is less than 55, we copy 20 to the temporary array and increment i to 2 {-22, -15, 1, 7, 20}
- 35 is less than 55, we copy 35 to the temporary array and increment i to 3 {-22, -15, 1, 7, 20, 35}
- Only 55 is left. Copy it to the temporary array
- Copy the temporary array back into positions 0 to 6 of the original array, because all the elements have been checked

The array is sorted.

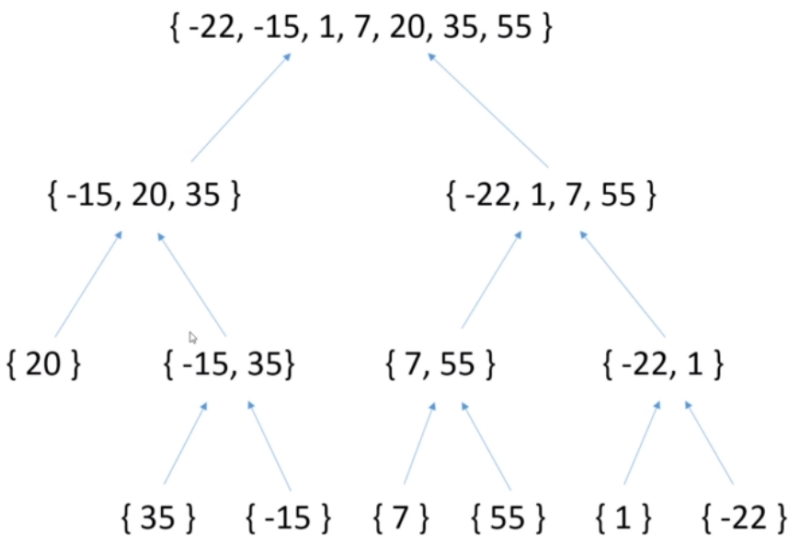


Figure 3: Merging Phase

Performance of Merge Sort

- ✓ NOT an in-place algorithm (The splitting phase is in-place, but during the merging phase a temporary array to merge each pair of sibling array is used)
- ✓ O(n log n)$_2$. Repeatedly dividing the array in half during the splitting phase, this is a logarithmic algorithm.
- ✓ Stable algorithm (during the merging phase, one checks whether the element in the right array is greater than the element in the left array, and if it is not, if it is equal to it, the element in the left array will be the one that is copied into the temporary array first, in this case the relative ordering of duplicate items will be preserved)

3 Benchmarking Results

This chapter describes sorting algorithms that were chosen for comparison. Each algorithm was executed ten times for each size. To measure the running time System.nanotime() was used, and nanoseconds were converted to milliseconds. These results can be seen in Table 4.

The results also depend on hardware and software factors, such as: application language, processor, Operating system, RAM

The running time (in milliseconds)

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
BubbleSort	0.427	1.079	0.852	1.154	1.849	2.832	11.401	31.421	40.804	64.359	97.637	131.536	181.954
SelectionSort	0.172	0.658	0.545	0.432	0.665	1.327	3.600	8.842	17.487	26.543	38.042	53.623	63.476
InsertionSort	0.114	0.861	0.294	0.386	0.150	0.232	0.813	2.167	3.909	6.898	9.128	15.553	20.463
ShellSort	0.198	0.191	0.700	0.408	0.209	0.143	0.315	0.478	0.809	1.325	1.081	1.099	1.302
MergeSort	0.158	0.082	0.206	0.356	0.370	0.364	1.347	1.337	1.588	1.399	1.540	2.312	1.942

Table 4: Benchmark output Table

These results were then plotted on a line graph to visualize the comparative analysis results shown in Figure 4.

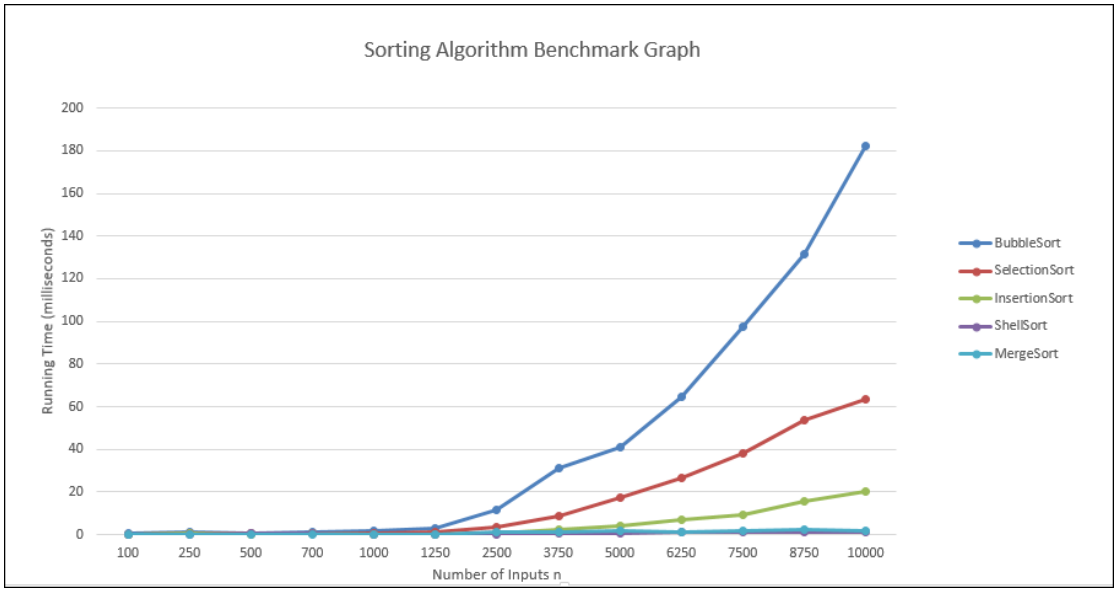


Figure 4: Sorting Algorithm Benchmark Graph

From Figure 4 it is visible that at the beginning all algorithms are quite fast, but when the number of inputs reaches 1250 some of them become slower; and becomes slower with the growing number of inputs. In general, the results turned out to be quite predictable.
Bubble, Selection and Insertion sorts have quite fast run time for reasonably small arrays. They are also relatively simple algorithms. But sometimes they may be faster than more complicated algorithms with very small arrays. The slowest from these three algorithms is Bubble sort and it is the simplest algorithm. Then goes Insertion sort. And though they have the same quadratic time complexity - $O(n^2)$, Selection sort is unstable algorithm it is a minus, a stable sort is preferable to an unstable sort. Bubble sort is mostly used for education purposes.

In relation to Shell and Merge sorts, it looks like they both go hand in hand with alternating success. For greater clarity, Shell and Merge sorts have been plotted on a separate graph with a larger scale. (See Figure 5). From graph it is seen that Shell sort has a little bit better performance as compare with Merge sort.
Merge sort requires additional memory, approximately equal to the size of the original array. Memory is consumed by the recursive call and the constant creation of subarrays. This is a big disadvantage of this algorithm. It can be problematic where the input data is huge or the available memory is limited. Also on almost sorted arrays, it works as long as on chaotic ones. In other words, Merge sort is a relatively memory intensive but efficient and stable algorithm.

Shell sort is simple and does not require additional memory, but it is unstable, so can not be used in some cases. The computational complexity of Shell sort highly depends on the array size and the chosen gap. There are many improvements to the gap sequence reduction rule. Sedgewick's version for even and odd cases, created in 1986, is widely known, but it is difficult to implement. The series compiled by Ciura in 2001 is considered the most perfect; it allows processing arrays up to several thousand elements in size. From the mathematically generated sequences, the most recent one was proposed by Tokuda in 1992.

So Merge and Shell sorts are the fastest from all the five analyzed algorithms.

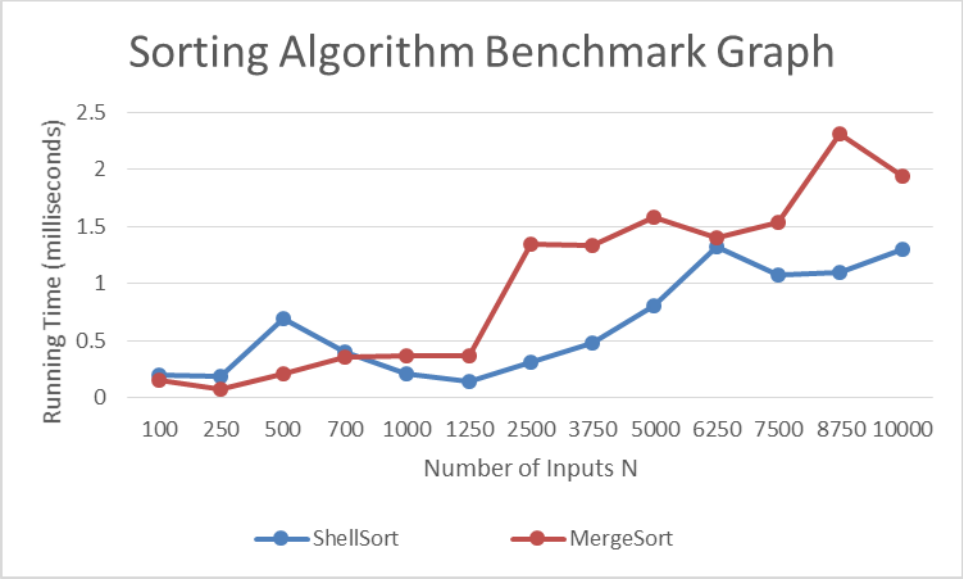


Figure 5: Benchmark Graph for Shell and Merge Sorts

4. Conclusion

In conclusion there is no better or worse sorting algorithm. Each algorithm is useful for certain applications. It is possible to get a general idea, and for each case it is necessary to select its own algorithm. Behavior of algorithms depends on the size of the input data, terms of complexity, speed, stability, memory requirements, and other parameters. However, almost every algorithm is the most convenient in any particular situation. Even very slow algorithms are used for educational purposes, which is due to their simplicity.
There are a lot of other sorting algorithms. Some of them are minor modifications of the discussed algorithms, while others take radically different approaches.
An algorithm that performs great on one set of data may perform horribly on other data, so it's important to know how to choose the algorithm that works best for any specific scenario.

5 Benchmarking Results for Additional Algorithms

Finishing the project I was interesting, how some other algorithms will perform, so I have added some extra algorithms to my implementation.

QuickSort	0.122	0.397	1.636	1.169	0.147	0.142	0.271	0.330	0.450	0.535	0.788	1.022	1.244
RadixSort	0.159	0.256	0.408	1.002	1.074	1.072	1.927	1.535	3.293	2.380	2.391	2.683	3.194
CountingSort	0.017	0.067	0.128	0.131	0.206	0.157	0.277	0.091	0.096	0.079	0.057	0.050	0.062

Table 5: Benchmark output Table

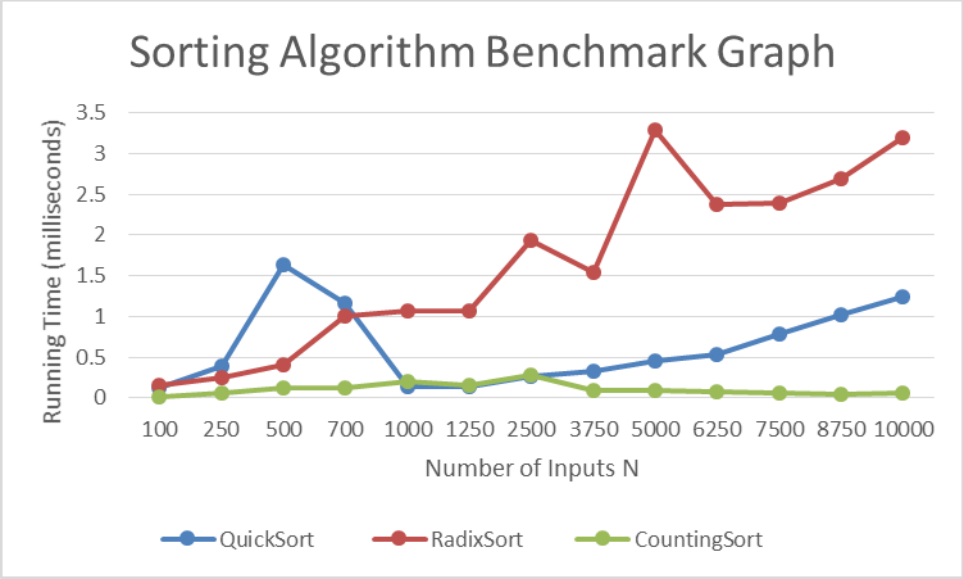


Figure 6: Benchmark Graph for Quick, Radix and Counting Sorts

References

1. Essential Algorithms: A Practical Approach to Computer Algorithms - Rod Stephens. Copyright © 2013 by John Wiley & Sons, Inc., Indianapolis, Indiana. Published simultaneously in Canada

2. https://en.wikipedia.org/wiki/Big_O_notation#/media/File:Comparison_computational_complexity.svg

3. https://en.wikipedia.org/wiki/Shellsort