Computation Thinking Project Report

Introduction

Sorting involves arranging a collection of items according to pre-defined rules. Applications which benefit from sorting include searching for an item in an array, finding out how many times an element appears in an array and calculating various mathematical statistics from an array. Computer scientists and programmers have discovered sets of instructions to sort particular arrays of elements which are not in order, for example an array of numbers or letters of the alphabet. The desired order may also be in different formats, for example numerical ordering for shortest to longest distance or lexicographical ordering for strings of words. These sets of instructions for sorting arrays are known as sorting algorithms. Many sorting algorithms have been invented since merge sort was invented by Von Neumann in 1945. Since then, various comparison-based, non-comparison based and hybrid sorting algorithms have been invented. Tim sort, a hybrid sorting algorithm was invented only 20 years ago in 2002. Interestingly, tim sort is part of Android - a commonly used technology today.

Sorting algorithms have helped speed up operations in computers for example, when a computer is multitasking. There are properties of sorting algorithms which may be described as desirable and some as undesirable. Computer scientists need to know the desirable and undesirable properties of many algorithms because no perfect algorithm exists that is useful for every application. Stability is a desirable property. This means values that have the same value, will keep the same order in the sorted output, that they had in the unsorted collection. Such an algorithm is known as a stable sorting algorithm. An algorithm that does not have this property is known as unstable. Efficiency is another important factor to consider when choosing an algorithm, either in the best case, average or worst case. The efficiency depends on the run-time and space needed. For example, if one wants to know the longest possible time an algorithm could take to run, one would study statistics gathered from the worst case, and could make arrangements to be able to avoid unexpectedly long running times. Or if the best case is found to have a very fast run time, this would be taken into consideration when deciding which algorithm to use for a particular problem. If the run-time efficiency is good in the average case, then most runnings or iterations of the algorithm will be fast. In situations where memory is a concern, one wants an algorithm that uses in-place sorting. These use a fixed amount of working space regardless of the input size of a dataset. The suitability or efficiency of the algorithm is another factor to consider. Some algorithms are only suited to certain types of input classes and run only in certain time frames, or orders of time.

The complexity of an algorithm is a measure of the efficiency of its design, not taking into account the effect a type of computer or machine has on the performance of the algorithm. That is, if one wants to measure complexity he should run the algorithms on the same computer and with the same technology. Both time and space complexity should be taken into consideration. Time complexity is a description of the asymptotic behavior of running time as

input size goes towards infinity. Time complexity is strongly correlated with the size of the input data set. When an algorithm is run it will need some space in a computer's memory. If the space complexity is "1", the space required for the algorithm will equal the size of the input data plus a fixed amount of additional memory which doesn't depend on the size of the input data set. The running time of some algorithms strongly correlates with the number of inversions that must be performed to sort the algorithm. An inversion is an unordered pair of elements in an unsorted array. So, if there are only a small number of inversions needed to be performed on an array, one could say the array is almost sorted. Different factors can affect an algorithm's running time, such as complexity, how many elements need to be sorted, do the elements have the same order relation or in what way are the elements related? Can elements be placed into fast or only in slow memory such as on a disk. This is known as external sorting.

The order of growth of an algorithm is proportional to mathematical algebraic functions.

Comparator functions are functions where two elements can be passed in and it can be determined if one element is either less than, equal to or greater than the other element. One can create his own custom definitions of the comparison operators. This greatly increases the range of tasks and areas of application which sorting algorithms can be applied to.

As previously stated, sorting algorithms can be divided into comparison-based, non-comparison based and hybrid types. A comparison sort can be defined as a sorting algorithm which uses comparsion operators only to determine which of two elements should appear first in a sorted array. Examples of these include bubble sort, insertion sort and quick sort. ((Note: nlogn is the best performance in the average and worst cases))? Non comparison sorting algorithms sort without comparing the elements, these sorts employ the internal character of the elements, and can only be performed with certain input sets. "All sorting problem that can be sorted with non-comparison sort algorithm can be sorted with comparison sort algorithm, but not vice versa."

"Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does the algorithm become incredibly slow? Is it able to maintain its fast run time as the input size grows? You can answer these questions thanks to Asymptotic Notation.

You can't compare two algorithms head to head. It is heavily influenced by the tools and hardware you use for comparisons, such as the operating system, CPU model, processor generation, and so on. Even if you calculate time and space complexity for two algorithms running on the same system, the subtle changes in the system environment may affect their time and space complexity.

As a result, you compare space and time complexity using asymptotic analysis. It compares two algorithms based on changes in their performance as the input size is increased or decreased.

Asymptotic notations are classified into three types: Big-Oh (O) notation Big Omega (Ω notation Big Theta (Θ) notation"(Upadhyay, 2022, para. 15).

The algorithms I am choosing for this project are bubble sort, bucket sort, quick sort, intro sort and tim sort.

Bubble sort

The bubble sort algorithm is one of the comparison-based sorts. It works by iterating through a list or array of numbers and comparing two adjacent elements during each iteration. If the element to the left is greater than the element exactly next to it on the right, the elements are swapped and the program moves to the next index in the array and repeats the process. If the element to the left is less than the element to its right, no swap is made, and the program moves to the next iteration. The process is repeated until the end of the array, and when the entire array has been iterated through, the largest element will be placed at the end of the array. That element can be said to be sorted. The next iteration will leave this element out, and after the next iteration, the largest element in the remaining unsorted part of the array will be placed exactly to the left of the sorted element from the first iteration. The process of iterating through the array will be repeated until the unsorted section of the array is composed of only two elements, and if the smallest element is already on the far left side of the array, no more iterations or swaps are performed and the result is a sorted array or list of numbers. In certain cases, a final iteration may be required to check is the array actually sorted. The worst case time complexity of bubble sort is O(N^2). This happens when the initial array is arranged in descending order if the desired output array is in ascending order, or the other way around. O(N^2) is also the average case and best case time complexity of bubble sort. The best case occurs when the initial array is already sorted. The algorithm only needs to run through the array once and no swaps of elements are required.

Bucket Sort

Bucket sort, also known as bin sort, is a stable sorting algorithm. Also called bin sort. It works by putting the elements into what are known as buckets according to the range of values which the elements belong to. Another sorting algorithm is then applied to each bucket or else the bucket sort algorithm can be recursively applied. It's a generalization of counting sort. Building up each discrete type or distinct type in the input instances. Ranges are chosen for the buckets of similar length, i.e. 0-9, 10-19, 20-29. These buckets have ten keys each. An extra step can partition the elements in each bucket and place them into new buckets that have 5 keys. Then eventually insertion sort may be suitable for very small unsorted arrays. Time complexity is O(n^2) in the worst case and O(n+k) in the best and average cases (where n is the number of elements and k is the number of buckets). Worst case space complexity is O(nk). The best case occurs when the elements of the initial array are evenly distributed, and hence each bucket will

have roughly the same number of elements. The time complexity for filling buckets with the elements is O(n), and when using insertion sort on the buckets the time complexity is O(k), giving bucket sort an overall time complexity of O(n+k) in the best and average cases. The worst case time complexity of O(n^2) occurs when there is a very uneven distribution of elements in the initial array resulting in, for example, some buckets having a lot of elements and other buckets have one or no elements. If the elements are also in descending order in the initial array this further increases the time complexity. Bucket sort is useful when input instances are evenly distributed. Bucket Sort's performance degrades with clustering. That is, when many values occur close together they fall into the same bucket and will take longer to sort than values in other buckets. Procedure: Set up an array of buckets which are initially empty. Iterate through the array, putting each element into its correct bucket. Sort each non-empty bucket. Visit the buckets in order, and place each element back in its correct position. An empty bucket would be a sort of base case, or if it has one element we can assume it's sorted and that it is also part of the base case.

Quick Sort

Quick sort follows the approach known as 'divide and conquer', which means dividing a problem into smaller problems and solving the smaller problems before combing all the solutions together to solve the original problem. A pivot is picked from an unsorted array. Then the array is partitioned into two subarrays around the pivot, or if the pivot is at the fat left or far right of the array, the pivot itself is one of the subarrays. Each element in the left subarray should be smaller than the pivot; and each element in the right subarray should be larger than the pivot. The two subarrays are themselves then partitioned around an element chosen as a pivot and the process is repeated until the subarrays are single elements. The subarrays are then combined to get the final sorted array. The time complexity of quick sort is O(nlogn) in the best and average cases, and is O(n^2) in the worst case. The best case occurs when the pivot element is near to the middle element. The average case occurs as long as the array is not in an ascending or descending order. The worst case occurs when the pivot happens to be either the largest element or the smallest element in the array, or, if the pivot is the last element of the array, and the subarray to the left of the pivot is already sorted in ascending or descending order. "Though the worst-case complexity of quick sort is more than other sorting algorithms such as merge sort and heap sort, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quick sort can be avoided by choosing the right pivot element.(javaTpoint, 2021)"

Count Sort

Proposed by Harold H. Seward in 1954. There are limitations on the type of input instances in non-comparison sorts. Counting sort lets you sort a collection in close to linear time. Assuming an input size, n, where each element has a non-negative integer key with a range of k; best, average and worst time complexity is n + k and space complexity is also n + k. A good

improvement for large input instances. Although this algorithm isn't as widely applicable as comparison-based sorts. You could assign label values with an integer value, that is a mapping which may allow you to apply count sort to certain problem. Counting sort is stable if it's implemented correctly. Procedure: Determine the key range, k, of the input array. Then initialize an array count of size k, that is the number of discrete sort keys that are possible. Then this is used to count the number of times each key value appears in the input instances. There could be multiple instances of the same key value in the input instances. You count how many times each specific key value occurs. Then initialize an array result of size n, which is used to store the sorted output. This gives us n + k space complexity. Then you iterate through the input array and record the number of times each distinct key value occurs in the input instance. This gives a histogram of key frequencies. Then you construct the sorted array based on the histogram of key frequencies stored. Refer to the ordering of keys in the input to ensure stability is preserved. That is, if more than one element has the same key value, their order will be preserved.

Intro Sort

Hybrid sorting algorithm proposed by David Musser in 1997. It can utilize quick sort, heap sort or insertion sort. The algorithm initially adopts quick sort and monitors the recursive depth of the to ensure efficient processing. If the depth of recursion exceeds nlog levels, then intro sort changes to Heap Sort (comparison-based) instead. If the recursion won't exceed that threshold and the partition size isn't too small it the algorithm stays with quick sort until the end. If the partition is small enough, the algorithm will revert from quick sort to insertion sort. Intro sort is comparison-based because the algorithms which it uses are also comparison-based. Best, average and worst case time complexity is O(nlogn). It is also an in-place sorting algorithm It is worth noting that it doesn't have the worst case complexity of quick sort, O(n^2).

Tim Sort

A hybrid sort proposed by Tim Peters in 2002, as a reference sorting algorithm for Python. It is made from a combination of merge sort, insertion sort and other logic. The algorithm finds subsequences of an array that are already ordered, and it can then sort the remainder more efficiently. The more subsets of the array that are already sorted will speed up the algorithm significantly. Tim sort is also used in arrays of non-primitive types in Java SE 7 and in GNU Octave software. The array is divided into blocks known as runs. The runs are then sorted using insertion sort, and merged using merge sort to output the final sorted array. Advantage of tim sort is that it performs better than bubble and selections sort and it is a stable sorting algorithm. A disadvantage of tim sort is that it is not in-place due to extra memory requirements for merging. The best case time complexity occurs when the initial array is already sorted and is of the order O(n). The average and worst case time complexities is of the order O(nlogn). For a number of elements, n, the space complexity is O(n).

Criteria for choosing a Sorting Algorithm

Insertion sort: very good for a small number of items are if elements are almost sorted. Also good if one desires to write as little code as is possible.

Heap sort: the best if one is worried about the worst-case scenario or want a good general performance.

Quick sort: the best if one is interested in good average-case behaviour.

Bucket sort: good choice if there is a relatively uniform distribution among the elements.

Merge sort: if one wants stability to be preserved.

Conclusion: Comparison-based sorts are the most widely applicable, but are limited to nlogn running time in the best case. Whereas, non comparison-based sorts can achieve linear running time in the best case but they aren't as flexible as comparison sorts.

Implementation discussion

As I was using Python programming language, I needed to acquire source code to be able to run each of my chosen algorithms. These I stored in separate .py files on my python-enabled visual studio code (vscode). I also needed to create Python functions to generate arrays of different number elements, n; and also to time each algorithm for a given number, n, of randomly generated arrays. The in-built time method is able to achieve this. I stored various times in named variables that I then used to plot a line graph of these times against the input number, n. Due to different processes that a computer may be running, these can cause a longer run time, a computer's scheduled tasks or if too many processes or programs were running at the same time. To account for this possible anomaly, I wrote code to run each algorithm 10 times and then got the average of these ten runs. For this I used a for loop, and I also used a for loop to store each result (time) in an array. Multiplying each result by a 1000 gave me results in milliseconds which was more suitable for faster algorithms for lower input instances of n. I also outputted a results table containing different times recorded for the five algorithms for various n values up to 10,000. I also created a line graph to see if I could notice the trends as n got larger, and to visually compare the algorithm's results on the same graph. Bubble sort performed much worse than the other 4 (bucket sort, intro sort, quick sort and tim sort) and it was therefore made no sense to output a line graph containing all 5 algorithms together. Considering bubble sort's time complexity of n^2, as opposed to the nlogn and n + k complexities in average cases for the other algorithms, the output largely reflected these time complexities. A quadratic function will take longer than a linearithmic or a linear function.

Leaving out bubble sort, I was better able to see the trends in the line graph for the other algorithms. From the tabular data I outputted into a text file, bucket sort performance degraded significantly for n equal to 16500 compared to intro, quick and tim sort, this is despite remaining only on average about twice as slow as those algorithms. As bucket sort requires fewer comparisons than quick sort, the decrease in performance for a high n could be put down to the time cost involved in creating and or filling buckets. Another reason could be the effect of clustering, that is, a large number of elements going into some buckets which would increase the run time for the algorithm. A surprising result was a zero time output for input instance of n equal to 120 for bucket and intro sorts, despite recording the results in milliseconds. In the case of intro sort, a small input n causes this algorithm to revert to insertion sort, so I can only assume insertion sort is exceptionally fast for a small n. In the case of bucket sort, it is possible the elements were very evenly distributed and or were already sorted after being put into buckets. Over the range of n inputs, intro sort could be said to have performed the best. Although tim sort performed better than intro sort for certain input instances.

Bubblesort example with diagram:

The primary array below is the initial unsorted array. We begin by comparing the first two elements in the array, i.e. the elements in positions index 0 and 1. In this case, 11 is greater than 1 so a swap is performed. The element at position index 2 is also less than the element at index 1 so a swap is performed again. The element at position index 3 is greater than its adjacent element on the left so no swap is performed here and the program leaves the array as it is, and now compares the elements at indexes 3 and 4. A swap is once again performed. As 29 happens to be the largest element in the array, this element will be swapped on all subsequent comparisons with the remaining elements until 29 is left at the end of the array, as the first sorted element.

11	1	4	29	21	10	15	2

1	11	4	29	21	10	15	2

1	4	11	29	21	10	15	2

1	4	11	21	29	10	15	2

. .

. .

1	4	11	21	10	15	2	29

Now the second iteration will move through n – 1 elements, where n is the number of elements in the array. In the above array, no swap will be made until the program reaches elements at positions index 3 and 4, because the element at index 3 (21) is greater than the element at index 4 (10). The element 21

happens to be the largest element in this array of length n –1, so it will be swapped with the remaining elements and end up at index n – 2 as shown below:

1	4	11	10	15	2	21	29

Quicksort example with diagram:

In the initial unsorted array, a pivot is chosen from any of the elements. The elements less than the value of the pivot are placed to the left of the pivot, and the values that are greater than the pivot are placed to the right of the pivot. The two subarrays are then also partitioned using the same method. This method is continued until the subarrays become single elements. Finally they are combined to form a sorted array. If the pivot is chosen as the far left element, the algorithm begins at the far right and moves towards the left. Since the pivot element is greater than the last element on the far right of the array (R), the algorithm swaps them and so the pivot is now at position R. As the pivot is at the far right (R), now the algorithm starts from the far left and will move to the right. Since 7 is less than the pivot element (11), the algorithm moves one position to the right. Since 1 is also less than 11, the algorithm again moves one position to the right, to the element at index 2 of the array, which is 4. Since 4 is also less than 11, the algorithm moves one position to the right once again. Now, 29 is greater than 11 so the algorithm does a swap. The algorithm again begins from the element at the far right position and works its way left until it reaches an element that is less than the pivot value and then performs a swap. Below the algorithm will perform a swap with the pivot when it gets to element 10 as show in the last two arrays:

11	1	4	29	21	10	34	42	7

Pivot

7	1	4	29	21	10	34	42	11

↑ Pivot

7	1	4	29	21	10	34	42	11

↑ Pivot

7	1	4	29	21	10	34	42	11

↑ Pivot

7	1	4	11	21	10	34	42	29

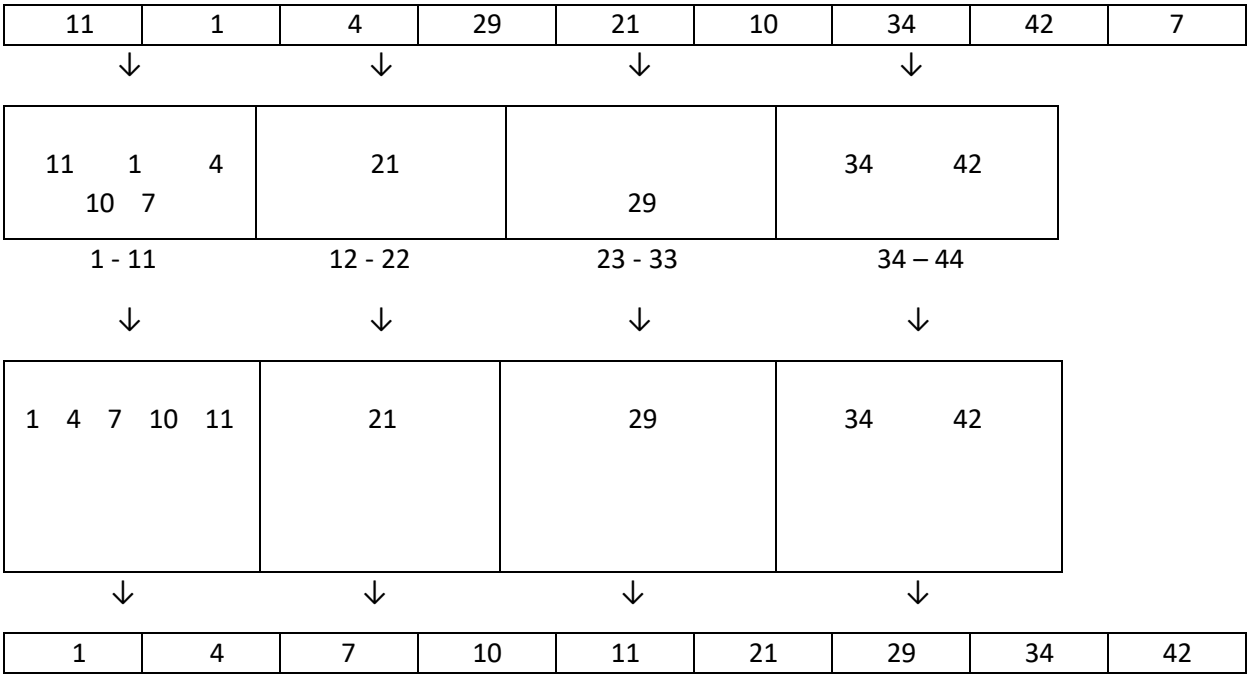
Pivot

7	1	4	10	21	11	34	42	29

Pivot

Bucketsort example with diagram:

A number, N, of empty buckets are created initially. Buckets is a loosely worded term to describe a container to temporarily hold elements from an array. The elements of an unsorted array are put into one of these N buckets according to what range of the elements the bucket is designated to hold, and the corresponding range the element belongs to. Another sorting algorithm or method could achieve this. The elements in each non-empty bucket are then sorted; another sorting algorithm such as insertion sort can be used to achieve this. Lastly, the elements are taken out of the buckets in order, left to right and joined together to create a new sorted array.



Introsort example with diagram:

Initial unsorted array

↓

quick sort is performed first

↓

Subarrays of small partition size

↓

Algorithm revets to insertion sort

↓

Final sorted array

Initial unsorted array

↓

quick sort is performed first

↓

Subarrays not too small or big

↓

Algorithm continues to with quick sort

↓

Final sorted array

Subarrays with a larger partition size

↓

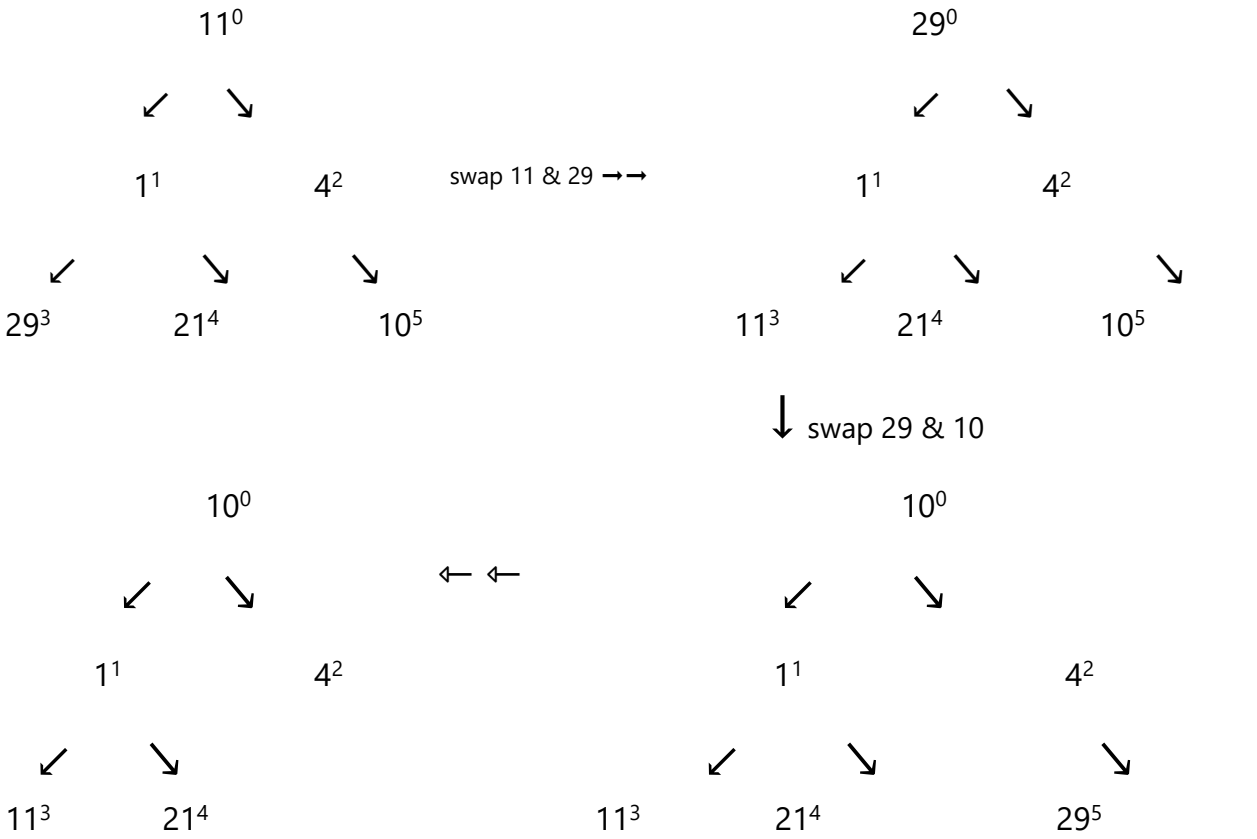
Apply quick sort – partition around the pivot recursively until the subarrays are small enough to apply insertion sort. If the recursive depth goes beyond nlogn, the algorithm adopts heap sort.

↓

11	1	4	29	21	10

↓

Heap sort process with example heaps taken from the above array:



The largest element from the initial array is now removed and placed at the last position in the array, position 5. The other elements in the unsorted section of the array are again heapified, swaps are made again and the largest element removed until the heap contains only one element and it is removed to be placed at the first position, resulting in the final sorted array. Note that intro sort will revert to insertion sort from heap sort once the partition is small enough.

Timsort example with diagram:

A min-run size is first established of that is a multiple of 2, for example 16 or 32. A min-run size of 64 or higher will slow down the insertion sort part of the algorithm. The min-run size ensures all runs found will be at least this size. The program's counter begins at array[0] and notes the size of the element, it then moves to array[1] and compares it with element at array[0].

11	1	4	29	21	10	34	42

↓ run size of 4

11	1	4	29

↓ First iteration, 11 & 1 are swapped

1	11	4	29

↓ Second iteration 4 & 11 are swapped to leave a sorted run

1	4	11	29

↓ Third iteration, the progam moves to the next position of the array, array[3] which is element 29 but doesn't have to do a swap as the program has reached the last element of the run.

1	4	11	29

Run 2:

21	10	34	42

↓ First iteration 21 & 10 are swapped, the program does not have to perform any other swaps.

10	21	34	42

The 2 sorted runs are now merged using merged sort into the final sorted array.

↓

1	4	10	11	21	29	34	42

References

https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity#:~:text=Asymptotic%20Notations%20are%20programming%20languages,as%20an%20algorithm's%20growth%20rate.

https://www.javatpoint.com/quick-sort

https://www.mycareerwise.com/programming/category/sorting/intro-sort