comptSort: A Multi-Algorithm Sorting Function

Jacob Sánchez Pérez

INTRODUCTION

comptSort is a Python library designed to sort sequences of data with one of 5 different sorting algorithms: Bubble Sort, Insertion Sort, Binary Insertion Sort, Merge Sort and Quicksort. It requires Python >=3.9.

USAGE

The public API mainly consists of two functions, one that sorts a list in place, and one that returns a sorted copy of a list.

To import:

```
from comptSort import compSort
from comptSort import compSortInPlace
```

`def comptSort(uData, sort, asc = True):`

Return a sorted copy of a list.

`def comptSortInPlace(uData, sort, asc = True):`

Sort a list in place.

Arguments

uData: `list`

The original, unsorted data.

sort: `str`

A string specifying the sorting algorithm to be used.

asc: `bool`

If true, sort will be performed in ascending order, otherwise, in descending order.

sort can be one of the following: "bubble", "insertion", "bin_insertion", "merge", "quick". Additionally, the `SortingAlgorithm` **enum** can be used instead of a string, which can be imported like so:

```
from comptSort import SortingAlgorithm as Sorting
```

COMMAND LINE INTERFACE

The library includes a command line interface that can be invoked with

```
$ python -m comptSort.cli

usage: comptSort [-h] [-a {bubble,insertion,bin_insertion,merge,quick}] [-d] [-
i] [-o OUTPUT] file

Test the comptSort library by sorting lines of files.

positional arguments:
 file       file containing items to sort

options:
-h, --help
      show this help message and exit

-a, --algorithm
      algorithm to sort with

-d, --descending, --reverse
      order sequence in descending order

-i, --force-integers
      parse lines from files as integers (by default strings)

-o, --output
      write sorted lines to file (otherwise printed to console)
```

This interface allows any user to quickly sort files containing unsorted strings/characters or integers (with the **`-i`** flag) into sorted files.

~~EXAMPLES~~

~~Additionally, some sample files are included in the examples/ directory. To test the library with them, use the CLI, or use the `examples.py` script, which will run through all sorting algorithms in both asc/desc order, with only the 10 integers / characters examples, since fitting all the data on screen is otherwise difficult.~~ Additionally, there a `timing.py` script is also included which can be used for a quick timing test on every algorithm, using a random shuffle of numbers.

COMPATIBLE DATA TYPES

The comptSort and comptSortInPlace as well as all the sorting algorithms implemented are in theory compatible with **any** data type that can be compared with **<** (less-than). The reason for this is that Python is dynamically-typed. Furthermore, all sorting algorithms included in this library are comparison sorts.

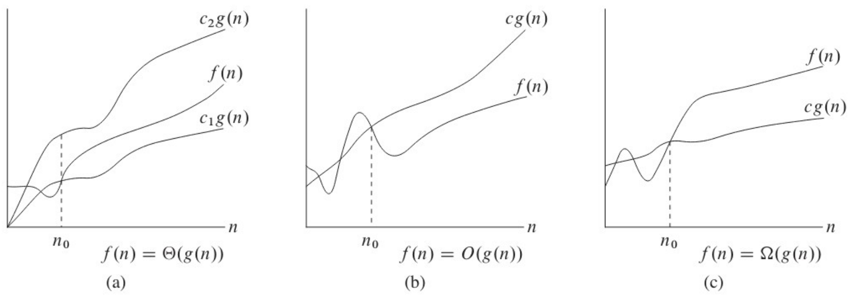
On the other hand, the CLI is only compatible with strings and integers (using the **`-i`** flag).

ASYMPTOTIC NOTATION

Generally speaking, the time an algorithm takes to complete is dependant on the size of its input. Therefore it is common to describe this running time in relation to the input size (Cormen et al., 2022). A useful notation to express these relationships is known as asymptotic notation (Cormen et al., 2022, p. 3). Usually, an algorithm with a lower asymptotic time will be faster except in small inputs (Cormen et al., 2022, p. 43). However, asymptotic notation does not only describe time complexity of algorithms, as it "actually applies to functions" (Cormen et al., 2022, p. 44), and thus can describe just about any other aspect of algorithms, such as their space complexity (Cormen et al., 2022, p. 44).

Asymptotic notation for the most part is only concerned about the most siginificant terms in a mathematical expression (Sedgewick, 1997, p. 45). It suppresses detail, allowing to focus on the most important terms. Even though, as Sedgewick explains, asymptotic notation is not all that matters, it does provide accurate approximations that allow to make informed choices and draw comparisons between algorithms.

Some asymptotic notations which are appropiate to study the running times of algorithms include Θ (Big Theta), O (Big O), and Ω (Big Omega) notations (Cormen et al., 2022, p. 44).



Θ notation can be used to bound a function within constant factors, both above and below, for a sufficiently large n. If the time taken by an algorithm to run is $\Theta(\mathrm{g(n)})$, it means that for a large enough input, the running time of the algorithm only differ from $\mathrm{g(n)}$ by a constant factor (Cormen et al., 2022, p. 47).

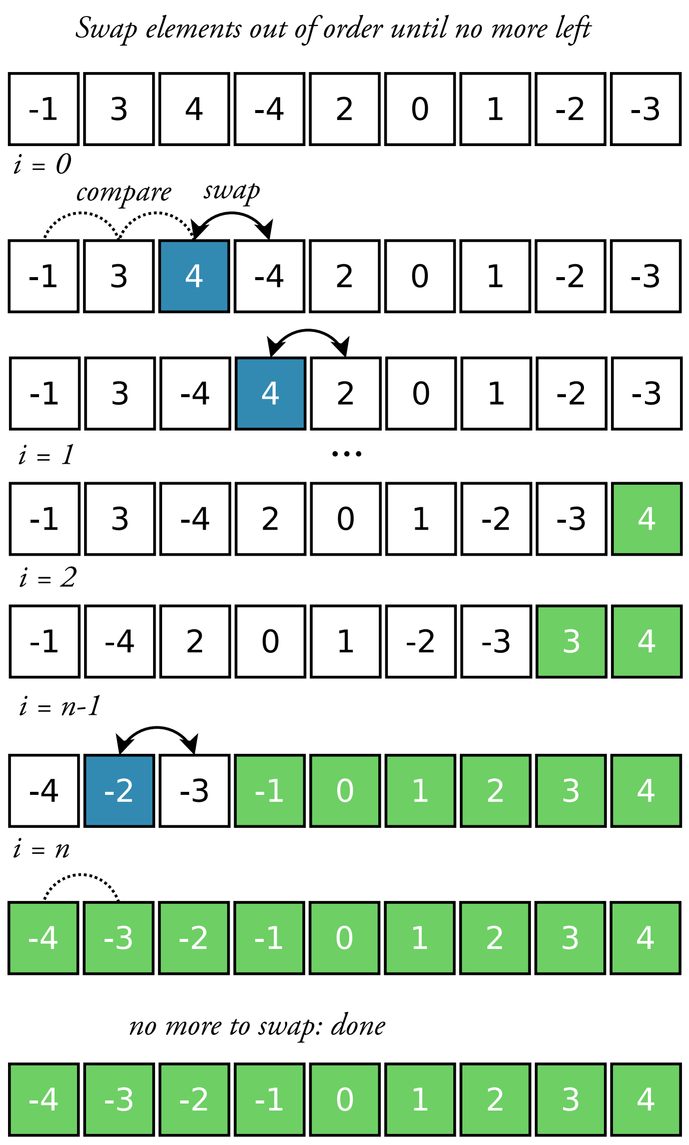
On the other hand, Big O notation (O) will only give an upper bound to a function. This means that while the running time of an algorithm may not necessarily be proportionate to the function denoted by O, it will never exceed it by more than a constant factor (Cormen et al., 2022, p. 48).

Finally, Ω provides an asymptotic lower bound, which translates to the running time of an algorithm never being lower than the function inside the notation by more than a constant factor (Cormen et al., 2022, p. 48).

Often, an algorithm will have a different time complexity depending on its input. By analysing the worst case input for an algorithm, it is possible to provide an upper bound for any given input (Cormen et al., 2022, p. 27).

SORTING ALGORITHMS

BUBBLE SORT



Time complexity

Best case $O(n)$ (Min, 2010)
When elements are already in order

Worst case $O(n^2)$ (Min, 2010)
When sequence in reverse order

Average case $O(n^2)$ (Min, 2010)

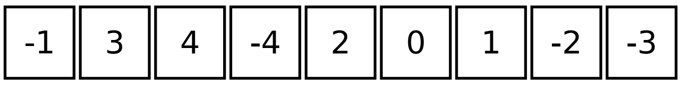
Space complexity $O(1)$ (Min, 2010)
Since it operates in place.

Bubble sort consists in repeatedly passing through the sequence, swapping neighbouring elements that are not in the right order, until no elements are swapped in a single pass (Sedgewick, 1997). It is considered popular and simple to implement (Sedgewick, 1997), but quite inefficient (Cormen et al., 2022) and not *best practice* (Astrachan, 2003). It works in place.

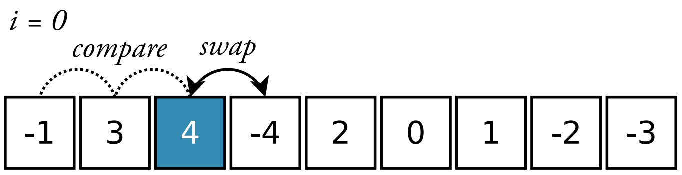
The name *bubble sort* is owing to the fact that large elements "bubble up" until they have reached the right position. However, it has also been known as *exchange selection* or *propagation* (Knuth, 1997).

Walkthrough

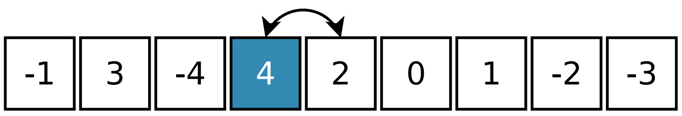
This will be the list to sort (ascending order).



First pass
Compare first element (-1) with second (3). As they are in order nothing happens.
Compare second element (3) with third (4). As they are in order nothing happens.
Compare third element (4) with fourth (-4). Since -4 is less than 4, swap them.

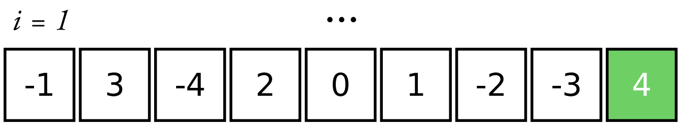


Compare element 4 with 2. Since 2 is less than 4, swap them.

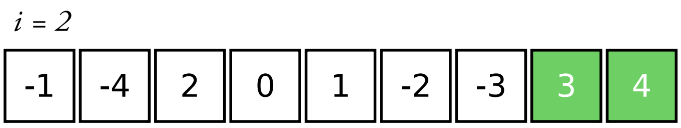


Continue this process until the end of the sequence is reached.

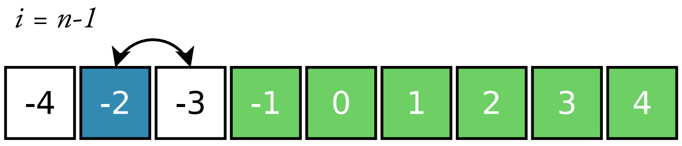
This will be the array at the start of the second pass. Notice how 4 bubbled up to the last index in the array. An efficient implementation of bubble sort will not go over the elements that have settled in the last position after each pass. A more efficient one will also store the index where the last swap ocurred, and stop comparing elements after that, since they have already been compared (Knuth, 1997).



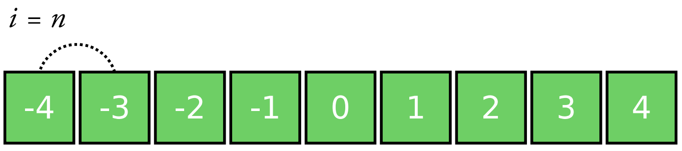
After the second iteration, element 3 has also bubbled up to the right position.



On the second to last iteration, the last swap will be made.



In the last pass, no swaps will be made and thus, the algorithm will terminate.



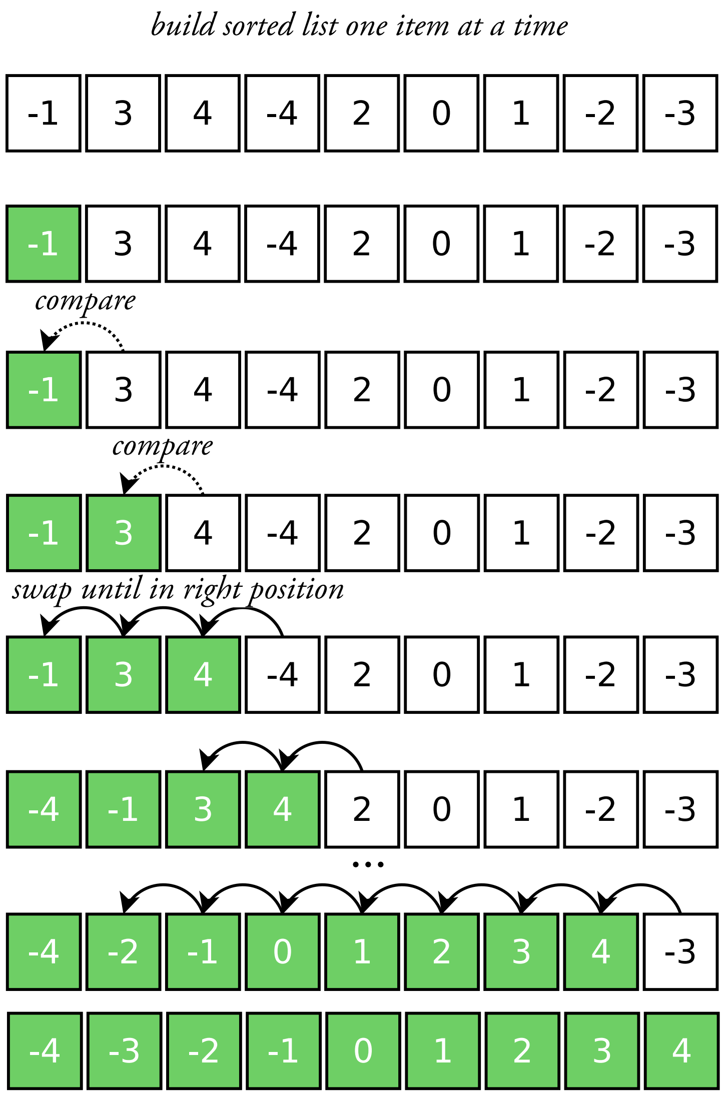
Analysis

According to Knuth (1997), there are three numbers affecting the timing of bubble sort. These are the number of passes, swaps, and comparisons. The minimum values for these properties are 1, 0, and N-1, respectively, while the maximum are N, $((N^2$ - $N)$ / $2)$, and $((N^2$ - $N)$ / $2)$. The maximum of all three happens if the sequence is in reverse order, while the minimum only if the input is already sorted. Using asymptotic notation, this means bubble sort will make $\mathrm{O}(n^2)$ swaps and comparisons in the worst case, while in the best case, it will make $\mathrm{O}(n)$ comparisons, and $\mathrm{O}(1)$ swaps. The average case also has an asymptotic time of $\mathrm{O}(n^2)$ for both swaps and comparisons (Knuth, 1997).

Efficiency

One of the weaknesses of this algorithm is the fact that elements can only move to the left one step per iteration. Using an optimisation called "cocktail shaker sort", which alternates the direction of traversal between passes, slightly reduces the number of comparisons compared to regular bubble sort (Knuth, 1997). However, even after some optimisations, bubble sort is still not as efficient as regular insertion (Knuth, 1997), which is why its inclusion in academic curriculums is often seen as controversial. Indeed, Astrachan (2003) found bubble sort to be as much as 3 times slower as Insertion sort, even though they have the same time complexity in the worst case.

INSERTION SORT



Time complexity

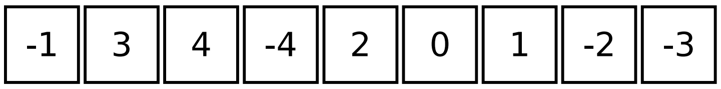
Best case	$O(n)$	(Cormen et al., 2022, p. 26)
Worst case	$O(n^2)$	(Cormen et al., 2022, p. 27)
Average case	$O(n^2)$	(Cormen et al., 2022, p. 28)

Space complexity $O(1)$ (Cormen et al., 2022, p. 148)
Since it operates in place.

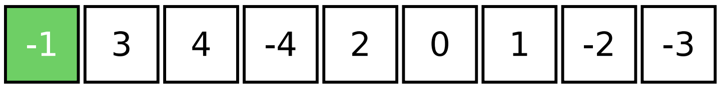
Insertion sort works in a similar fashion to how a hand of cards is usually sorted (Cormen et al., 2022). The input sequence will have a sorted and an unsorted section. The unsorted section starts by being empty. Then, one element at a time, elements are moved from the unsorted section to the right position in the sorted section. This position is found by comparing the new element to each one of the sorted ones, starting from the right. Insertion sort works in place (Cormen et al., 2022).

Walkthrough

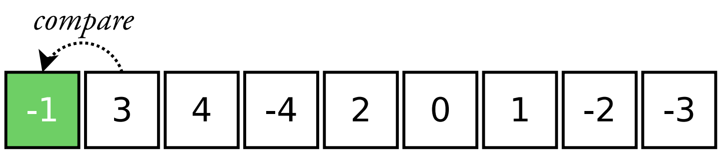
This will be the list to sort (ascending order).



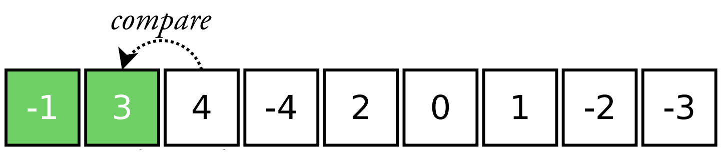
The first element can be skipped, and considered part of the sorted part of the list.



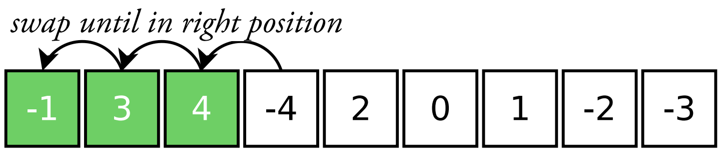
Start with the second element, and find its right position by comparing with the sorted elements one by one from the right, in this case, since 3 is greater than -1, no swaps are needed.



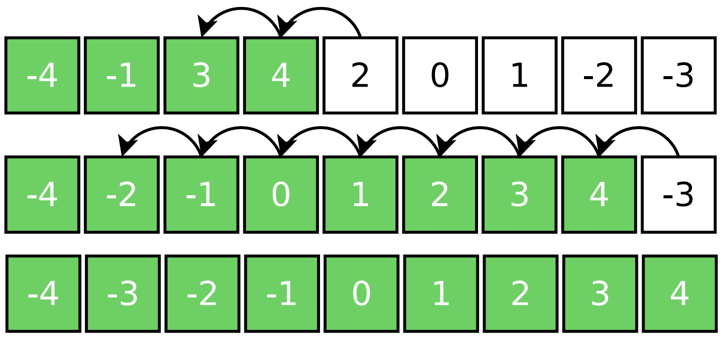
Compare the third element with the last element of the sorted list, since 4 is greater than 3, no swapping is needed.



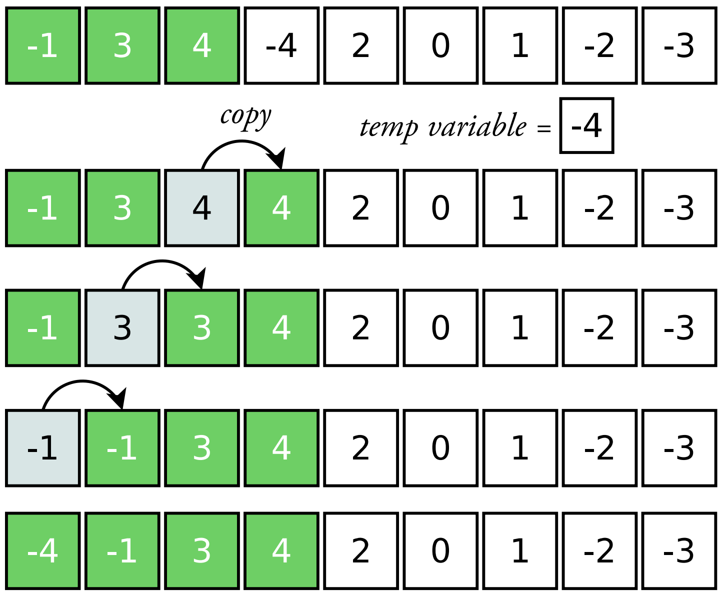
Compare the fourth element (-4) with the last element of the sorted list. Since -4 is less than 4, swap them. Next, compare to the second last element of the sorted list (3). Swap them too. Elements are swapped until the new element is in the right position.



Elements keep getting inserted in the right position, until no unsorted elements are left.



A slightly more efficient version of Insertion sort, as described in Cormen et al. (2022) simply moves values to the left instead of swapping, and stores the new value in a temporary variable. This reduces assignments by half.



Analysis

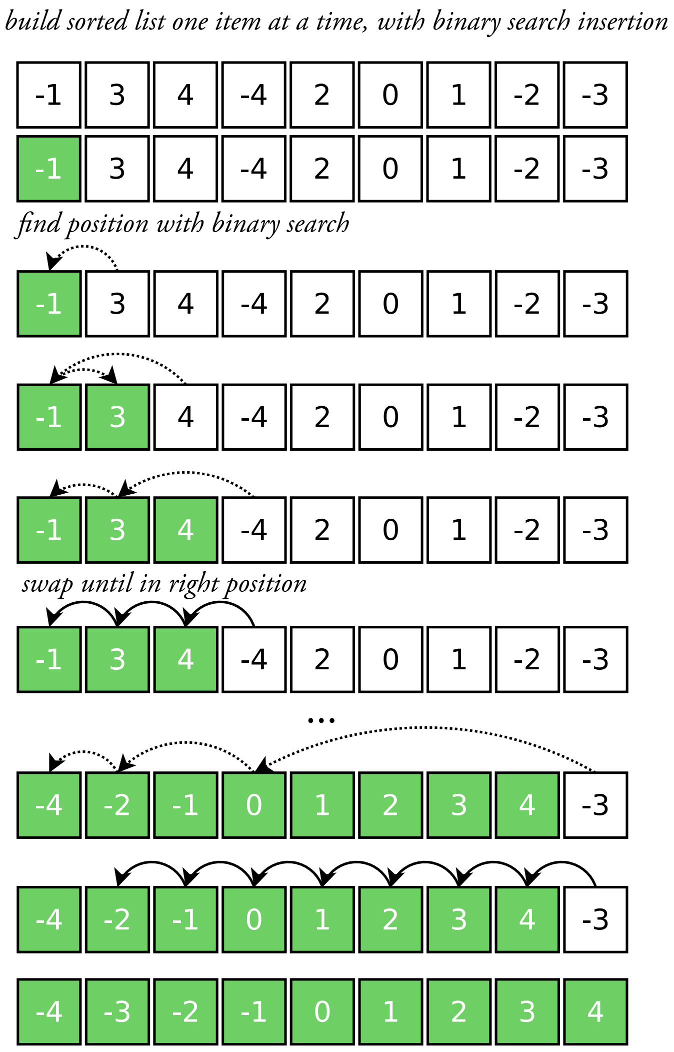
For Insertion Sort, the best case occurs when the input is already sorted state, while the worst case happens when the sequence is sorted backwards (Cormen et al., 2022). For the best case, the running time is linear, or using asymptotic notation $\mathrm{O}(n)$. On the other hand, in the worst case, the algorithm has to compare each element with every every element in the sorted subarray. This means the time complexity is quadratic, or $\mathrm{O}(n^2)$. In the average case, this latter time complexity holds true (Knuth, 1997).

Efficiency

Knuth (1997) concluded that straight Insertion sort was the simplest algorithm to implement, and is efficient for sequences with up to 25 elements. For large inputs, on the other hand, he found it "unbearably slow", unless the sequence was already almost sorted.

Variants of insertion sort that aim to improve timing include Binary Insertion, which uses binary search to located the insertion index of every new element, Two-way Insertion, which builds upon binary search and cuts the running time in half by being able to move elements in two directions to make space for new elements, and more.

BINARY INSERTION SORT



build sorted list one item at a time, with binary search insertion
-1 3 4 -4 2 0 1 -2 -3
find position with binary search
swap until in right position
...
-4 -2 -1 0 1 2 3 4 -3
-4 -3 -2 -1 0 1 2 3 4

Time complexity

Moving elements (swapping) takes more time than comparisons (Knuth, 1970).

Best case $O(n \lg n)$ (Cormen et al., 2022, p. 26)
Since in the best case, binary search still has to make $\lg n$ comparisons.

$O(n \lg n)$ comparisons, $O(n)$ swaps

Worst case $O(n^2)$ (Cormen et al., 2022, p. 27)

$O(n \lg n)$ comparisons, $O(n^2)$ swaps

Average case $O(n^2)$ (Cormen et al., 2022, p. 28)

$O(n \lg n)$ comparisons, $O(n^2)$ swaps

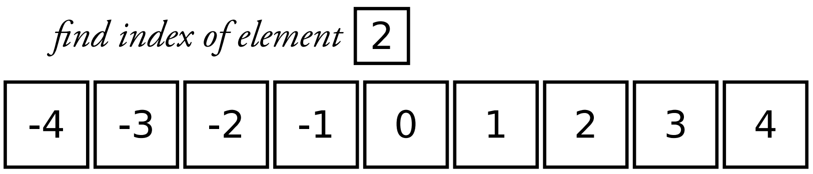
Space complexity $O(1)$ (Cormen et al., 2022, p. 17)
Since it operates in place.

Binary Insertion sort is a variant of Insertion sort that uses binary search to find the position of each new element (Knuth, 1970). This brings the number of comparisons made for each element close to the theoretical minimum. However, although this reduces the running time of the algorithm, moving elements in memory still takes $\mathrm{O}(n)$ time, which means the overall time complexity remains the same. (Knuth, 1970).

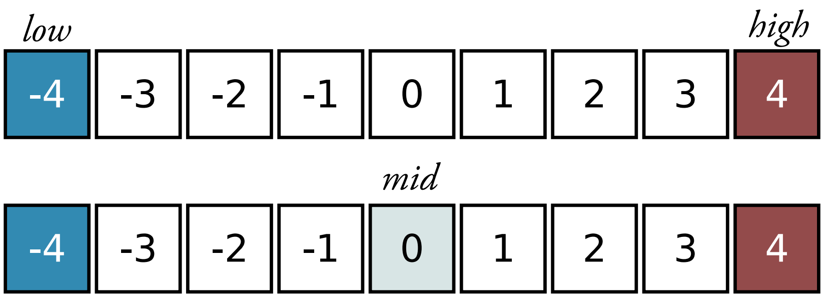
Binary search walkthrough

Binary search is an algorithm to find the index of an element within a sorted sequence. It is also known as *bisection* or *logarithmic search* (Knuth, 1997). To understand Binary Insertion Sort, it is a good idea to first understand binary search.

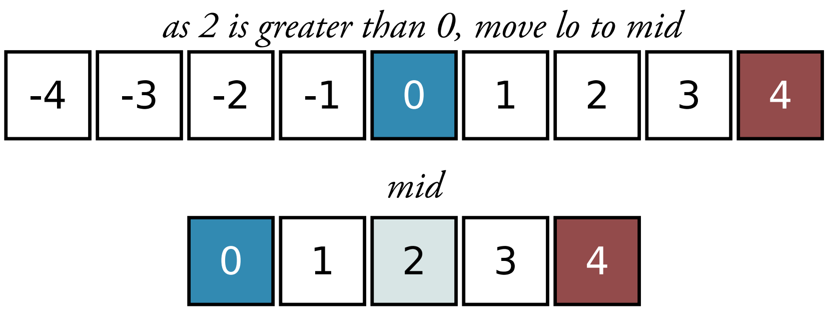
In this example, it is necessary to find the position of the element with value 2 in this sorted array. A sorted sequence is necessary to use binary search.



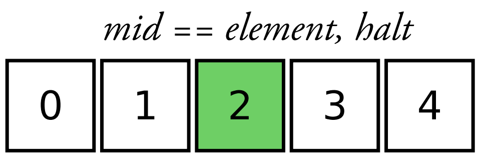
Binary search starts by storing the lowest index (0) in a variable, named lo in this example, and the highest index in another variable, named hi. Then, the middle point is found by averaging both indexes.



Then, the value to find (2) is compared to the value in the middle. As 2 is greater than 0, the lo variable is assigned the index that previously was in the middle. Had the value to find been smaller than the middle value, the hi variable would have to be changed instead. Next, the middle point is calculated again.

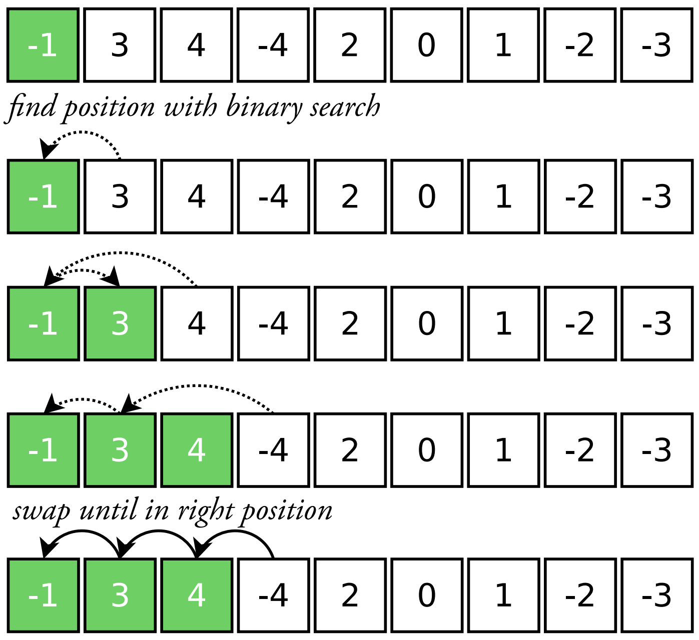


Since the value to find matches the one in the middle, the algorithm returns this index.



The algorithm for finding the insertion point for a new element works slightly different, and halts when the lo and hi variables cross each other.

With this in mind, the Binary Insertion Sort algorithm is one that simply replaces the linear comparison of regular Insertion Sort with a binary search. However, as mentioned previously, elements still have to be swapped linearly.



-1 3 4 -4 2 0 1 -2 -3
find position with binary search
swap until in right position

Analysis

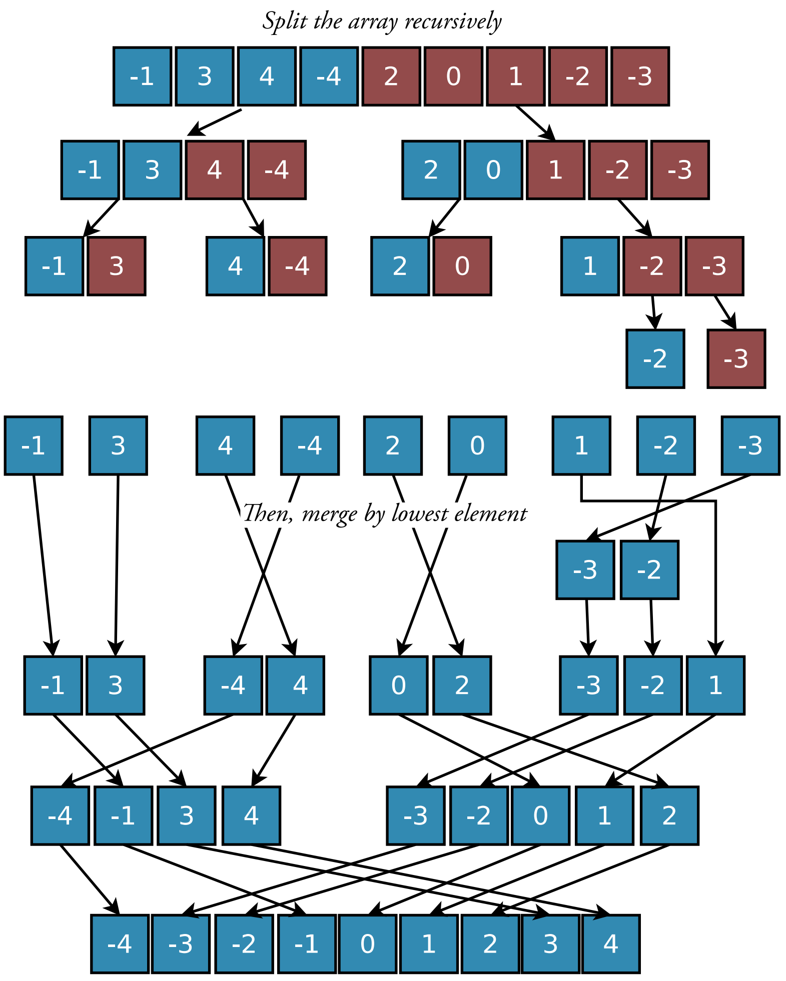
Using binary search to find the right position for a new j^{th} element brings the number of comparisons per element to $\lg j$. Therefore, the total number of comparisons is around $N \lg N$, which represents a significant improvement over the comparisons made in regular insertion, which is around $(N^2)/4$ (Knuth, 1997).

Unfortunately, regardless of how quickly the position for the new element is found, the bigger issue is moving it to that position in memory. As half of the sorted elements have to be moved on average, the running time remains $\mathrm{O}(n^2)$.

Efficiency

Binary Insertion improves the comparisons made by regular insertion and it does not require a much more complicated program to achieve so. However, due to limitations inherent to moving elements in memory, its time complexity remains quadratic (Knuth, 1997). There are ways to mitigate this limitation, such as "two-way" insertion (Knuth, 1997). This means it is not suitable for large inputs, as there are algorithms that can make as few comparisons without having to move as many elements in memory, such as Merge sort (Knuth, 1970).

MERGE SORT



Time complexity		
Best case	$\Omega(n \lg n)$	(Cormen et al., 2022, p. 125)
Worst case	$\Theta(n \lg n)$	(Cormen et al., 2022, p. 150)
Average case	$\Theta(n \lg n)$	(Cormen et al., 2022, p. 150)
Space complexity	$O(n)$	(Sedgewick, 1997)

The merge sort algorithm is based on the **divide-and-conquer** approach (Cormen et al., 2022, p. 30). This approach consists of breaking down a problem into smaller ones using **recursion**, solving these, and then combining them to build the full solution to the original problem.

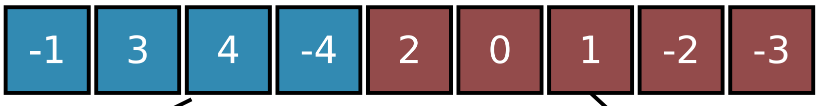
The merge sort algorithm follows this by first **dividing** an array into smaller ones, splitting it in half. Next, it sorts these using merge sort. Finally, these sorted arrays are combined into one, single sorted array (Cormen et al., 2022, p. 30).

Merge sort
0. If length is 1, return.
1. Split array in half.
2. Sort both arrays using merge sort, using recursion.
3. Merge both arrays into a single one.

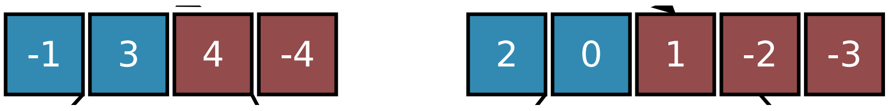
The algorithm eventually splits the array into single elements, at which point they are sorted (since they only have one element), and can be merged (Cormen et al., 2022, p. 30).

Walkthrough

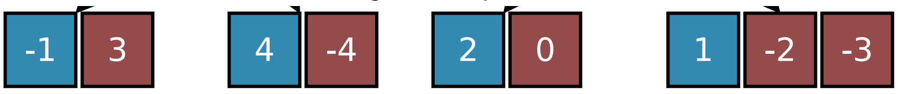
The initial array is divided in two.



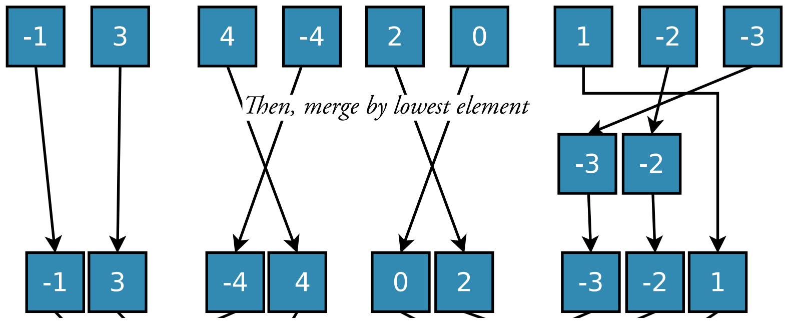
Merge sort is called on each of these. Therefore, they, too, are divided in two.



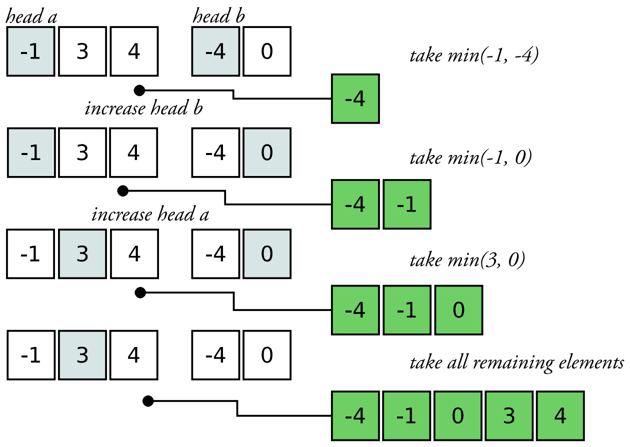
Merge sort is called on each of these four. Again, they are divided in two.



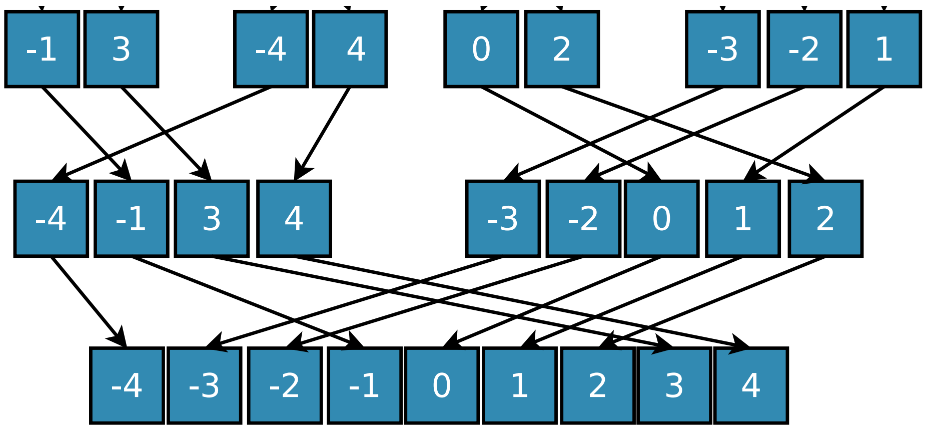
This recursive splitting continues until an array cannot be split any further, that is, it has reached the size of a single element and is already sorted. In that case, merge sort returns immediately.



Once merge sort returns for unit-sized arrays, it can start merging back up. The way this is achieved is by comparing the first element of both arrays and copying the smaller one to the new array, until both arrays are empty.



Arrays are merged back up until the first call to merge sort terminates.



Analysis

The merge operation, where two sorted sublists are merged into a single one, has a time complexity of $\mathrm{O}(n)$, since at most n comparisons are made (Cormen et al., 2022). Since the algorithm is recursive, its running time is best described with a ***recurrence*** (Cormen et al., 2022). For a small enough input, the solution takes constant time ($\mathrm{O}(1)$), which in the case of Merge sort, this small input corresponds to the single-element array. However, for all other cases, the timing will be dictated by the times taken to split the sequence, sort the sub-arrays, and merge them. Dividing the array takes constant time ($\mathrm{O}(1)$), merging the sub-arrays takes $\mathrm{O}(n)$ time (Cormen et al., 2022). If the recurrence describing the time complexity of merge sort is T, it will take $2T(n/2)$ to sort the sub-arrays. The recurrence can be solved to reveal an overall running time of $\mathrm{O}(n \lg n)$.

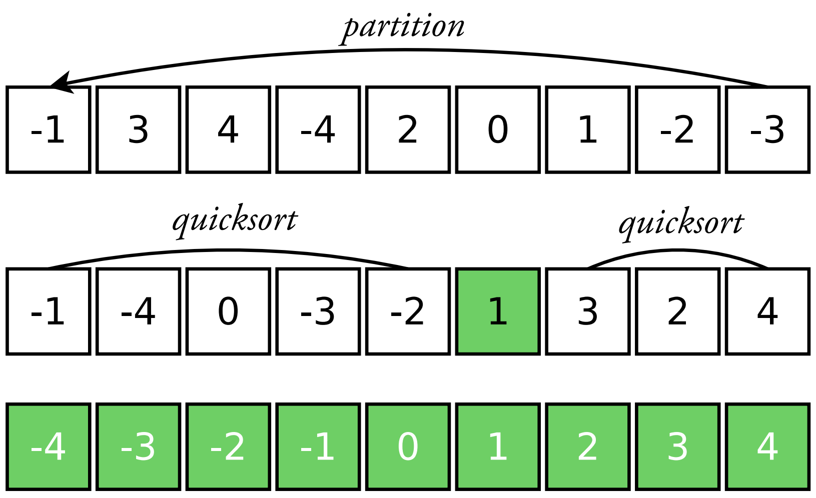
$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

A comparison sort has to make at least $\Omega(n \lg n)$ comparisons. Therefore, merge sort is asymptotically optimal (Cormen et al., 2022).

Efficiency

Merge sort is an optimal comparison sort. It is guaranteed to be fast even in the worst case (Knuth, 1997). However, the constant factors in Merge sort can mean that other sorting algorithms with a greater time complexity can outperform it in a small enough input. One such algorithm is Insertion sort (Cormen et al., 2022, p. 39). This is the reason some implementations of Merge sort switch to a different sorting algorithm for a small enough input (Cormen et al., 2022, p. 39). Furthermore, it is often slower than Quicksort due to their constant factors.

QUICK SORT



Time complexity

Best case	$\Omega(n \lg n)$	(Cormen et al., 2022, p. 125)
Worst case	$\Theta(n^2)$	(Cormen et al., 2022, p. 150)
Average case	$\Theta(n \lg n)$	(Cormen et al., 2022, p. 150)

Space complexity

Average case	$O(\lg n)$	(Sedgewick, 1997)
Worst case	$O(n)$	(Sedgewick, 1997)

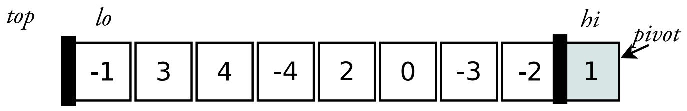
Invented by Antony Hoare in 1960 (Sedgewick, 1997), Quicksort is an in-place, divide-and-conquer sorting algorithm with little overhead and an optimal average case running time (Cormen et al., 2022, p. 170).

The idea behind Quicksort is taking an element and moving it to its final position within the sorted sequence. In the process of finding this position, every other record will be rearranged, such that no element to the right of it will be smaller, and no element to the left of it will be greater. After this, the problem is reduced to sorting both subarrays to the left and right of the element just inserted, which can be done using the same method (Knuth, 1997, p. 113). There exist multiple methods to achieve a partitioning into left and right sub-arrays (Knuth, 1997).

Quicksort
0. Check array limits are valid
1. Partition array, get pivot
2. Quicksort left subarray (before pivot)
3. Quicksort right subarray (after pivot)

Walkthrough of Partition

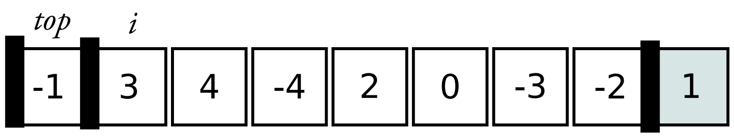
The partition algorithm starts with an array, and its hi and lo indexes. The former will also be known as the pivot. Another variable (here called top) is created with the initial value of lo.



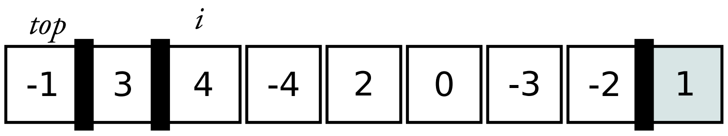
A simple optimisation to Quicksort is to select a random pivot every time, which can help increase the likelihood of a balanced split on average (Cormen et al., 2022, p. 179). This can be achieved by selecting a random index in the array, and exchanging it with the rightmost element before partitioning. A further optimisation is to select the median of three randomly selected elements in the array, also called **Median-of-3 partition** (Cormen et al., 2022, p. 188). In this example the pivot was randomly selected and swapped before starting the partition.

The algorithm will iterate through elements $\mathrm{lo}...\mathrm{hi}$. It will create two partitions, one with elements lower than the pivot, the other one with elements which are greater than it. The former will span from 0 to top, the latter from top to the current i.

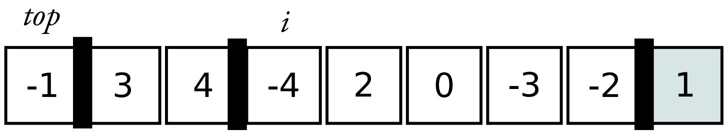
The first iteration starts and the element (-1) is compared with the pivot, as it is smaller, it swaps positions with top (since top is in the same position it remains in the same place) and top is increased by one.



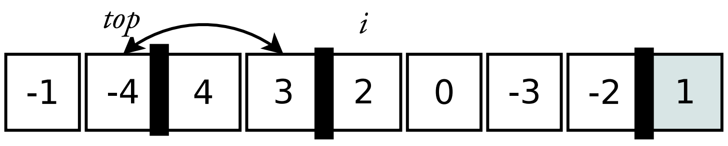
Then, the next element (3) is compared to the pivot, as it is greater than it, nothing else happens.



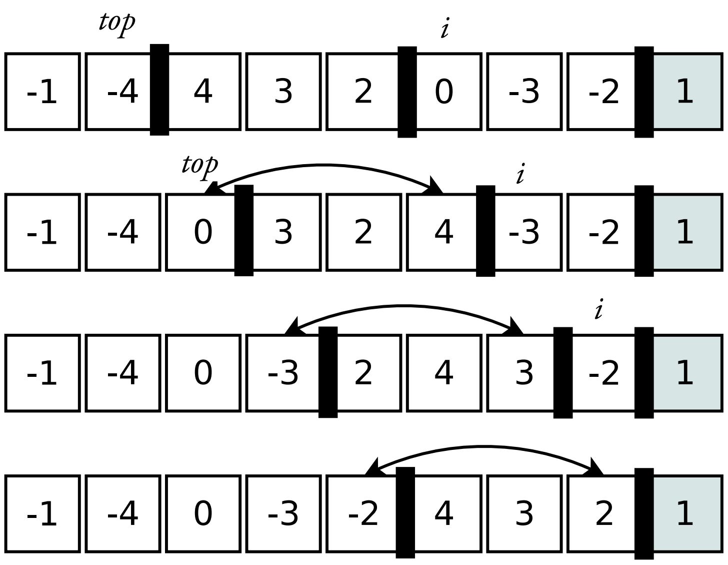
The same happens on the next iteration.



The next element (-4) is smaller than the pivot, therefore the new element swaps positions with top and top is increased by one.

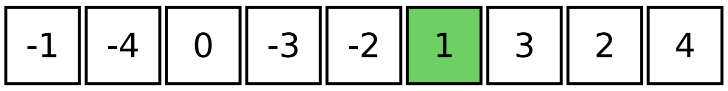


This process continues until the end of the sequence is reached.

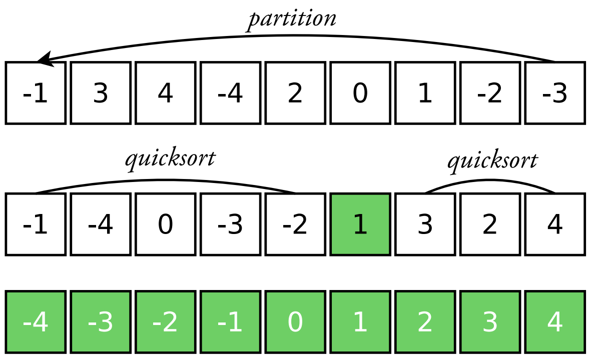


After this process two sub-arrays have been created, one with elements which are smaller than the pivot, the other one with elements which are greater.

To complete the algorithm, the pivot exchanges positions with top, finally resting in its right place.



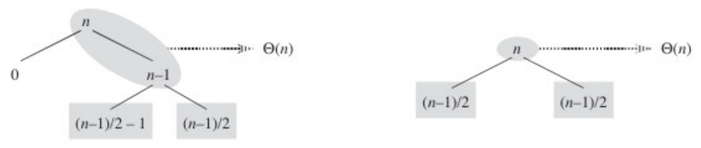
As mentioned above, Quicksort is simply calling partition recursively, settling elements one by one on their place, using divide-and-conquer.



There are different variants of this partition process, some more efficient than others.

Analysis

The performance of Quicksort depends strongly on the partitioning being balanced or unbalanced, which can bring its time complexity down to that of merge sort, or up to that of insertion sort, respectively. According to Cormen et al. (2022), the worst case in partitioning presents itself when the routine causes one of the partitions to have 0 elements, with the other one having as many as the previous iteration (minus the element just inserted). If this is the case on every recursive level, the time complexity of Quicksort becomes $\Theta(n^2)$. Moreover, this is the case on a sorted array on a non randomised Quicksort (Cormen et al., 2022). On the complete opposite case, the best partitioning occurs with a split as even as possible, which makes the time complexity $\Theta(n \lg n)$.



Efficiency

Quicksort was described by Knuth (1997) as the "most useful general-purpose technique for internal sorting", for its efficient memory use and its low average running time. It is said to the "best practical choice for sorting" (Cormen et al., 2022, p. 170), and "probably" the most widely used (Sedgewick, 1997, p. 303). It has an extremely small inner loop (Sedgewick, 1997, p. 303).

A good implementation and version of Quicksort is probable to be faster than any other sorting algorithm on any given computer (Sedgewick, 1997, p. 304).

REFERENCES

Astrachan, O. (2003). Bubble sort. *ACM SIGCSE Bulletin*, *35*(1), 1–5. https://doi.org/10.1145/792548.611918

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Knuth, D. E. (1970). The analysis of algorithms. *Actes Du Congres International Des Mathématiciens (Nice, 1970)*, *3*, 269–274.

Knuth, D. E. (1997). *The art of computer programming* (Vol. 3). Pearson Education.

Min, W. (2010). Analysis on Bubble Sort Algorithm Optimization. *2010 International Forum on Information Technology and Applications*, 208–211. https://doi.org/10.1109/IFITA.2010.9

Sedgewick, R. (1997). *Algorithms in C, parts 1-4*. Addison Wesley.

APPENDIX A: TESTING THE LIBRARY

Pipenv is required, it can be installed with

```
$ pip install --user pipenv
```

Install the development dependencies with Pipenv.

```
$ pipenv install --dev
```

Then, simply run the pytest command.

```
$ pipenv run pytest
```

Alternatively, run the tox command to test with multiple python versions.

```
$ pipenv run tox
```