Project

Vishnu Bangalore Thirumalesha

The project is done with Python programming. The main Data Structure used for working with the Sorting Algorithms is Python list.

The library random is used for generating random elements for the list based on the size of list user decides. The inputs are not hardcoded and user who runs the code is free to choose the size of input and system generates the shuffled values based on size. All values that will be populated are integer values.

The main program is running in app.py and it calls individual sorting algorithms placed in different .py files with respective algorithm names.

Note: The algorithms are implemented from scratch and the import is just for importing algorithm files to the main program and they are not libraries.

Sample output: 
output.txt

For input size of 50: Execution time is in seconds

Algorithms and their execution time

{'MergeSort': '0.0004954338073730469', 'HeapSort': '0.0019948482513427734', 'InsertionSort': '0.0080761909484863328',

'SelectionSort': '0.0109214782714843 75', 'BubbleSort': '0.0106782913208007 81', 'QuickSort': '0.0026137828826904297', 'QuickSortThreeMed': '0.0019960403442382812'}

0.0004954338073730469

Best Algorithm is :MergeSort with execution time :0.0004954338073730469s

For input size of 500:

Algorithms and their execution time

{'MergeSort': '0.004128932952880859', 'HeapSort': '0.005948543548583984', 'InsertionSort': '0.017136573791503906', 'SelectionSort': '0.015807390213012695', 'BubbleSort': '0.032010555267333984', 'QuickSort': '0.008014440536499023', 'QuickSortThreeMed': '0.00700736045837402 34'}

0.004128932952880859

Best Algorithm is :MergeSort with execution time :0.004128932952880859s

For input size of 1000:

Algorithms and their execution time

{'MergeSort': '0.006983041763305664', 'HeapSort': '0.013200044631958008', 'InsertionSort': '0.05349397659301758', 'SelectionSort': '0.03993868827819824', 'BubbleSort': '0.0964066982269287 1', 'QuickSort': '0.004933595657348633', 'QuickSortThreeMed': '0.03484606742858887'}

0.004933595657348633

Best Algorithm is :QuickSort with execution time :0.004933595657348633s

For input size of 4000:

Algorithms and their execution time

{'MergeSort': '0.03490734100341797', 'HeapSort': '0.03896951675415039', 'InsertionSort': '0.6344397068023682', 'SelectionSort': '0.5924923419952393', 'BubbleSort': '1.4128761291503906', 'QuickSort': '0.017920494079589844', 'QuickSortThreeMed': '0.1736905574798584'}

0.017920494079589844

Best Algorithm is :QuickSort with execution time :0.017920494079589844s

For input size of 10,000:

Algorithms and their execution time

{'MergeSort': '0.06814432144165039', 'HeapSort': '0.07815408706665039', 'InsertionSort': '4.466365814208984', 'SelectionSort': '3.2955877780914307', 'BubbleSort': '10.0802001953125', 'QuickSort': '0.08235669136047363', 'QuickSortThreeMed': '0.7087633609771729'}

0.06814432144165039

Best Algorithm is :MergeSort with execution time :0.06814432144165039s

For input size 25,000:

Algorithms and their execution time

{'MergeSort': '0.1661543846130371', 'HeapSort': '0.278688907623291', 'InsertionSort': '24.47555184364319', 'SelectionSort': '21.339855432510376', 'BubbleSort': '58.7831346988678', 'QuickSort': '0.12153387069702148', 'QuickSortThreeMed': '4.280285358428955'}

0.12153387069702148

Best Algorithm is :QuickSort with execution time :0.12153387069702148s

For input size 50,000:

Algorithms and their execution time

{'MergeSort': '0.3433799743652344', 'HeapSort': '0.5223896503448486', 'InsertionSort': '113.01661825180054', 'SelectionSort': '94.90005397796631', 'BubbleSort': '236.75030040740967', 'QuickSort': '0.4207096099853515 6', 'QuickSortThreeMed': '17.17217755317688'}

0.3433799743652344

Best Algorithm is :MergeSort with execution time :0.3433799743652344s

Instead of function for ease of differentiating between algorithms, they are placed in separate .py files

Algorithm	.py files
Merge Sort	MergeSort.py

Heap Sort	HeapSort.py
Insertion Sort	InsertionSort.py
Selection Sort	SelectionSort.py
Bubble Sort	BubbleSort.py
Quick Sort	QuickSort.py
Quick Sort using 3 Median	QuickSortTM.py

1) Merge Sort: Sequence of numbers is taken as input, and is split into two halves

Afterwards they are recursively sorted.

To combine the sorted array a function merge is called.

The sorted list is returned to the app.py

2) Heap Sort

First the list is made a max heap, after the heap is made. We remove the last element of the list and moved to root and max heap is built again to restore the max value at the root node.

The max value taken out from root node is placed in the last element of the list and is moved to the left in corresponding iteration.

The sorted list is returned to the app.py

3) Insertion Sort:

The array is traversed thru the length of the list. A key element is started from index 1 position and is compared with elements on its left always such that the elements to the left is always in sorted at any given point of iteration. This is done with the help of while loop inside for loop.

The sorted list is returned to the app.py

4) Selection Sort:

The list is traversed for length of list in the outer loop. Inner loop is also traversed but starting from the sorted position to the end of length of list. The min value of the list is selection as the first element and compared with other elements. Whenever a value less than min is found the min index number is saved, later after the inner loop is finished the min value and selected elements are swapped such that the minimum value from elements on the right of min index is found. The elements to the left of min index is always in sorted.

The sorted list is returned to the app.py

5) Bubble Sort:

Initially we loop to access each array element in the outer for loop, so that we compare and sort the elements in the inner loop from start of index to the list index the outerloop is pointing.

As the outerloop increments the sort of elements in the list also increments. The adjacent elements are compared and swapped.

The sorted list is returned to the app.py

6a) Quick Sort:

This is called with list, index 0 and last index position as parameters. Function will consider last element as pivot. Elements smaller than pivot is placed to left of pivot and all greater elements to right of pivot and the pivot element is placed at the correct position in sorted array. Same is broken down to split the list and recursive calls are made for the same

quicksort function. If the list is split based on an index then the elements are separately sorted before and after that split.

When the based case of recursive function is hit, we start returning the sorted list.

In the end the full sorted list is returned to the app.py

6b) Quick Sort using 3 Median:

There pivots are selected, one the first element of list, second the last element of list and third is found as the middle element of list. If the length of list is even then we take the mid element as the floor of the index returned. If odd then the exact half index element is considered [index starts from 0, hence this looks the opposite logic]. Once the 3 pivots are chosen then the median of those 3 are used as pivot. Later using that pivot we call recursive function to the quicksort again to the do the above again.

When the based case of recursive function is hit, we start returning the sorted list.

In the end the full sorted list is returned to the app.py