# Introduction to Sorting

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure

If in a book we want to find anything then we have to search each page but if we have index then it’s easy to find

Sorting is about data arrangement and storing it in ascending or descending order

It’s easy to find pattern and faster to locate items in a sorted list, It helps to compare lists and in multiple list it’s easy to find duplicate.

Sorting algorithms were mostly used during development of user interface, programming and debugging.

**Space Complexity**

When our program run it’s always use some memory to finish the code and memory depend some of the factor

* Variables
* Execution
* Instruction

Calculating the Space Complexity

Memory used by different type of variables and that depend on System configuration

|  |  |
| --- | --- |
| Type | Size |
| bool, char, unsigned char, signed char, int8 | 1 byte |
| int16, short, unsigned short, w char, w char | 2 bytes |
| float, int32,unsigned Int, long, unsigned long | 4 bytes |
| double, int64, long double, long long | 8 bytes |

**Time Complexity of Algorithms**

Total time took to complete the program

The algorithm performance may vary with different types of input data, hence if it takes more time that will we worst- case time complexity.

**In-place sorting**

This transforms input using a data structure with a small, constant amount of extra storage space.it does not need an extra space ,produces an output in the same memory.

#*https://en.wikipedia.org/wiki/Category:Stable\_sorts*

**Stable sorting**

Algorithms maintain the relative order of records with equal keys .whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

**Comparator functions**

This is built in function in python it compares two objects and returns a value in positive, negative or zero

**Comparison sort**

It only reads the list elements through a single abstract comparison operation that determines which of two elements should occur first in the final sorted list.

**Non-comparison-based sorts**

This is always fast algorithm because it did not compare, while for non-comparison based algorithms it’s O (n).

## Sorting Algorithms

Below are the five algorithms I have select.

* Bubble sort
* Selection Sort
* Radix sort
* Insertion Sort
* Merge Sort

**Reference** [**https://www.studytonight.com/data-structures/selection-sorting**](https://www.studytonight.com/data-structures/selection-sorting)

1. **Bubble Sort**

Compares the entire element one by one and sort them based on their values.

The bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element and ….

**Complexity Analysis of Bubble Sort**

(N-1) + (n-2) + (n-3) + ……

Sum = n (n-1)/2

O (n2)

Starting the first element (index = 0), compare the current element with the next element of the array.

If the current element is greater than the next element of the array, swap them.

If the current element is less than the next element, move to the next element. ……….

Example 62735

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | 7 | 3 | 5 |

**6>2 so interchange**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | 7 | 3 | 5 |

**6< 7 No Swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 6 | 7 | 3 | 5 |

**7>3 so interchange**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 6 | 3 | 7 | 5 |

**7>5 interchange**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 6 | 3 | 7 | 5 |

**Second Insertion**

**2<6 no swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 6 | 3 | 7 | 5 |

**6>3 so interchange**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 6 | 7 | 5 |

**6<7 no swap**

**7>5 so interchange**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 6 | 5 | 7 |

**Third Insertion**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 6 | 5 | 7 |

**2<3 no swap**

**3<6 no swap**

**6>5 so interchange**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 6 | 7 |

**6< 7 so no swap**

**Fourth Insertion**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 6 | 7 |

**2<3 no swap**

**3<5 no swap**

**5<6 no swap**

**6<7 no swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 6 | 7 |

**Complexity Analysis of Bubble Sort**

Worst Case Time Complexity O(n2)

Best Case Time Complexity O(n2)

Average Time Complexity O(n2)

Space Complexity O(1)

1. **Selection Sort Algorithm**

Smallest element in the array and swap it with the element in the first position, then it will find the secondsmallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

It starts with first element and searches the small element and replaces the first position,Then move to second element and look small number in sub array, Replace the element at the second position in the original array. This process will continuous till sorted.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Original array** |  | **After pass1** |  | **After pass2** |  | **After pass3** |  | **After pass4** |  | **After pass5** |  |
| |  | | --- | | **4** | | **7** | | **2** | | **9** | | **5** | | **6** | |  | |  | | --- | | **2** | | **7** | | **4** | | **9** | | **5** | | **6** | |  | |  | | --- | | **2** | | **4** | | **7** | | **9** | | **5** | | **6** | |  | |  | | --- | | **2** | | **4** | | **5** | | **9** | | **7** | | **6** | |  | |  | | --- | | **2** | | **4** | | **5** | | **6** | | **7** | | **9** | |  | |  | | --- | | **2** | | **4** | | **5** | | **6** | | **7** | | **9** | |  |

**Complexity Analysis of Selection Sort**

It has two nested for loops so input size of n, following will be the time and space complexity for selection sort algorithm.

Worst Case Time Complexity O(n2)

Best Case Time Complexity O(n2)

Average Time Complexity O(n2)

Space Complexity O(1)

1. **Insertion Sort Algorithm**

It start from the index 1and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left

Making the second element of the given array element at index 1, the key (the key will be as new card), Compare the key element with the element before it

If the key element is less than the first element, insert the key element before the first element. If it greater then insert it after the first element.

Now third element of the array as key will compare with elements to left and insert it at the right position. This process will run until it’s sorted.

**Start with second key of element**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6 | 2 | 7 | 3 | 5 | 4 |

**2< 6 change the order**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 6 | 7 | 3 | 5 | 4 |

**7>2, 7>6 no change the order**

**3>2, 3<6, 3<7 3 insert before 6 and after 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 6 | 7 | 5 | 4 |

**5>3, 5<7, 5 will insert before 6 and after 3**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 6 | 7 | 4 |

**4>3, 4<4, 4<6, 4<7, 4 will insert before 5 and after 3**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 | 7 |

**Array sorted**

**Complexity Analysis of Selection Sort**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 | 7 |

Insertion sort is an efficient sorting algorithm as it does not run on preset conditions using for loops but instead it uses one while loop, which avoids extra steps once the array gets sorted.

Worst Case Time Complexity O(n2)

Best Case Time Complexity O(n)

Average Time Complexity O(n2)

Space Complexity O(1)

1. **Merge Sort Algorithm**

Merge Sort follows the rule of Divide and Conquer to sort a given set of numbers/elements, recursively, hence consuming less time.

It break the given array midway original array into 2 smaller subarray, so we will break these subarrays into even smaller subarrays, until we have multiple subarrays with single element in them

A single element is already sorted, so once break the original array into subarrays which has only a single element, successfully broken down, now we have to merge all these subarrays.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 30 | 28 | 40 | 4 | 7 | 76 | 17 |

Divide the array in two parts

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30 | 28 | 40 | 4 |  |  | 7 | 76 | 17 |

Divide the array in two parts

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30 | 28 |  | 40 | 4 |  |  | 7 | 26 |  | 17 |

Break each element again

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30 |  | 28 |  | 40 |  | 4 |  | 7 |  | 26 |  | 17 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 28 | 30 |  |  |  | 4 | 40 |  | 7 | 26 |  | 17 |  |

Sort the element small to large

Sort the element small to large

Merge the divided sorted array together

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 28 | 30 | 40 |  |  | 7 | 17 | 26 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 7 | 17 | 26 | 28 | 30 | 40 |

The array is sorted now

**Complexity Analysis of Merge Sort**

Merge sort is quite fast so time of complexity is O(n\*log n)

Worst Case Time Complexity O(n\*log n)

Best Case Time Complexity O(n\*log n)

Average Time Complexity O(n\*log n)

Space Complexity O(n)

As merge sort always divides the array in two halves and takes linear time to merge.

It takes equal amount of additional space as unsorted array.

1. **Radix Sort Algorithm**

It works by sorting each digit from least significant digit to most significant digit.

Then sort by the digits in the 1's place, then the 10's place, and ….. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 83 | 39 | 97 | 345 | 26 | 72 |

Sorting the given array in **counting sort key is last digit of each number**

Sort by smallest key

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 72 | 83 | 345 | 26 | 97 | 39 |

**Now second digit number as key sort by smallest key**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 026 | 039 | 345 | 072 | 083 | 097 |

**Now first digit number as key sort by smallest key**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 39 | 72 | 83 | 97 | 345 |

Array is sorted now **Complexity Analysis of Radix Sort**

The time complexity is O (kn) and space complexity is O(k + n) . Here n is the number of elements and k is the number of bits required to represent largest element in the array.

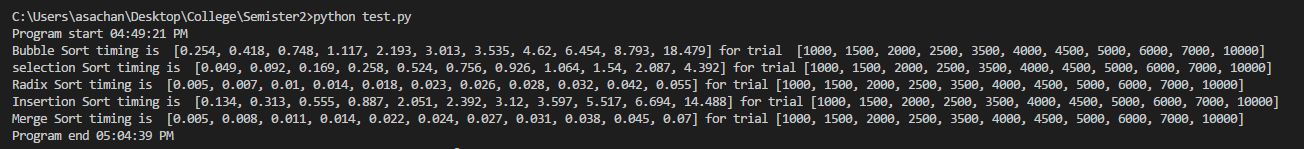
### Implementation & Benchmarking

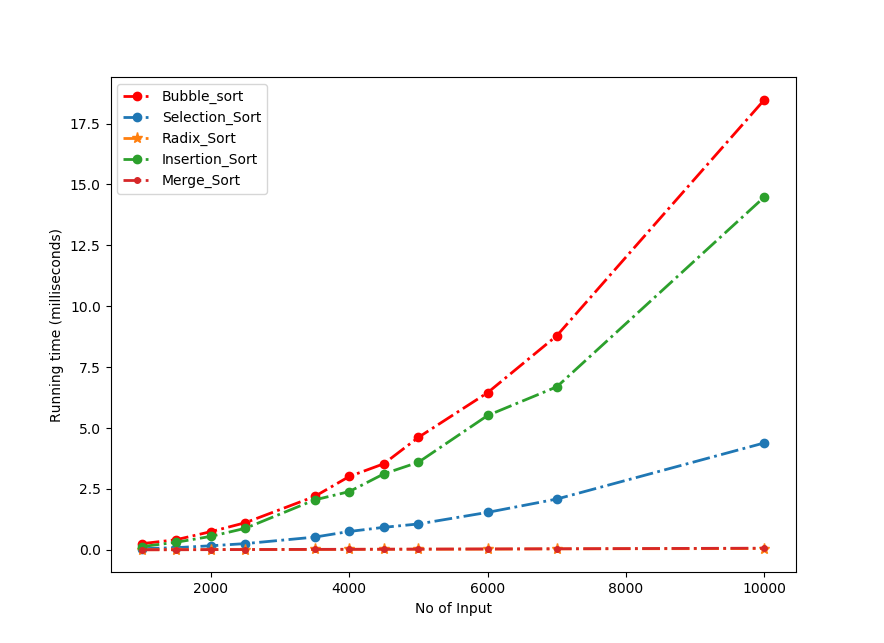
I have selected five algorithms (Bubble sort, Selection Sort, Radix sort, Insertion Sort, Merge Sort) and each algorithm I am generating a random number in count of [1000,1500,2000,2500,3500,4000,4500,5000,6000,7000,100000] and repeating 10 times then I am calculating average time to find a performance.

After running the code I see below result

**Results table – all values are in milliseconds, and are the average of 10 repeated runs**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Trial** | 1000 | 1500 | 2000 | 2500 | 3500 | 4000 | 4500 | 5000 | 6000 | 7000 | 100000 |
| **Bubble Sort timing is** | 0.254 | 0.418 | 0.748 | 1.117 | 2.193 | 3.013 | 3.535 | 4.62 | 6.454 | 8.793 | 18.479 |
| **selection Sort timing is** | 0.049 | 0.092 | 0.169 | 0.258 | 0.524 | 0.756 | 0.926 | 1.064 | 1.54 | 2.087 | 4.392 |
| **Radix Sort timing is** | 0.005 | 0.007 | 0.01 | 0.014 | 0.018 | 0.023 | 0.026 | 0.028 | 0.032 | 0.042 | 0.055 |
| **Insertion Sort timing is** | 0.134 | 0.313 | 0.555 | 0.887 | 2.051 | 2.392 | 3.12 | 3.597 | 5.517 | 6.694 | 14.488 |
| **Merge Sort timing is** | 0.005 | 0.008 | 0.011 | 0.014 | 0.022 | 0.024 | 0.027 | 0.031 | 0.038 | 0.045 | 0.07 |





* While looking the graph I see bubble sort has took lot of time and merge and radix sort took very less time.
* What I see when number of count increase its take more time but we see in count 5000 the time was 4.62 ms but in count 1000 (Double )it took time 4 times more times for bubble sort. Same goes for selection and inserting. (4 to 5 times more time)
* But radix and merge sorts when count increases I don’t see time difference to much.
* To see in more details I have change my count array [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000] and repeating 10 times then I am calculating average time to find a performance. In that scenario I see same things. Only code ruining time got increase.

#https://www.hackerearth.com/practice/algorithms/sorting/radix-sort/tutorial/

#https://www.quora.com/What-are-the-uses-of-different-sorting-algorithms-like-bubble-selection-insertion-shell-merge-heap-quick-tree-radix-counting-and-bucket-sort-in-real-life-scenarios

* **Merge Sort**: used in database scenarios because external results don’t all fit in memory, Useful in distributed scenarios where additional data arrive during or after sorting.
* **Radix sort:** Used in suffix array construction algorithms like Manber algorithm and DC3 algorithm. EBay allows you to sort listings by the current Bid amount leveraging radix sort.
* **Selection Sort:** K12 education portal allows sorting list of pupils alphabetically through selection sort.
* **Bubble Sort:** Bubble sort is used in programming TV remote to sort channels on the basis of longer viewing time.

Program start 09:40:40 AM

Bubble Sort timing is [0.001, 0.012, 0.05, 0.108, 0.173, 0.25, 1.005, 2.382, 4.133, 6.643, 9.419, 12.958, 17.138] for trial [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]

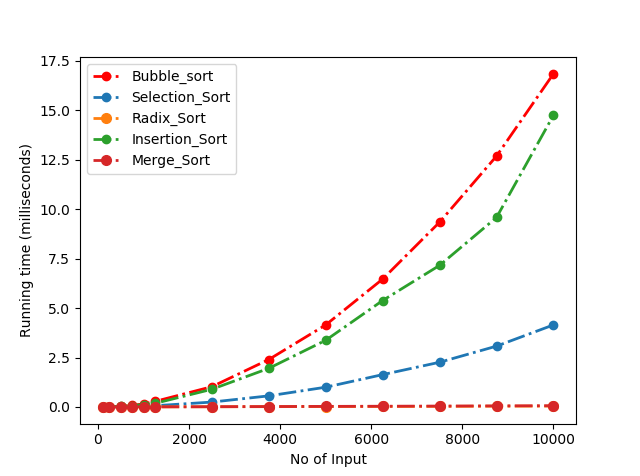
selection Sort timing is [0.0, 0.002, 0.01, 0.023, 0.041, 0.063, 0.254, 0.564, 1.015, 1.698, 2.246, 3.091, 4.477] for trial [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]

Radix Sort timing is [0.001, 0.001, 0.003, 0.003, 0.005, 0.005, 0.01, 0.02, 0.025, 0.032, 0.057, 0.053, 0.043] for trial [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]

Insertion Sort timing is [0.001, 0.007, 0.034, 0.069, 0.119, 0.186, 0.753, 1.954, 3.671, 5.265, 8.562, 10.712, 16.785] for trial [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]

Merge Sort timing is [0.0, 0.001, 0.002, 0.004, 0.005, 0.006, 0.013, 0.021, 0.029, 0.04, 0.045, 0.055, 0.063] for trial [100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000]

Program end 09:58:10 AM



**Results table – all values are in milliseconds, and are the average of 10 repeated runs**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Trail** | 100 | 250 | 500 | 750 | 1000 | 1250 | 2500 | 3750 | 5000 | 6250 | 7500 | 8750 | 10000 |
| **Bubble Sort** | 0.001 | 0.012 | 0.05 | 0.108 | 0.173 | 0.25 | 1.005 | 2.382 | 4.133 | 6.643 | 9.419 | 12.958 | 17.138 |
| **Selection Sort** | 0.000 | 0.002 | 0.01 | 0.023 | 0.041 | 0.063 | 0.254 | 0.564 | 1.015 | 1.698 | 2.246 | 3.091 | 4.477 |
| **Radix Sort** | 0.001 | 0.001 | 0.003 | 0.003 | 0.005 | 0.005 | 0.010 | 0.020 | 0.025 | 0.032 | 0.057 | 0.053 | 0.043 |
| **Insertion Sort** | 0.001 | 0.007 | 0.034 | 0.069 | 0.119 | 0.186 | 0.753 | 1.954 | 3.671 | 5.265 | 8.562 | 10.712 | 16.785 |
| **Merge Sort** | 0.000 | 0.001 | 0.002 | 0.004 | 0.005 | 0.006 | 0.013 | 0.021 | 0.029 | 0.04 | 0.045 | 0.055 | 0.063 |