CSE-5311-004
DESIGN AND ANALYSIS OF ALGORITHMS

Implementation of Different Sorting Algorithm

Rutuja Dukhande
UTA ID 1001730748

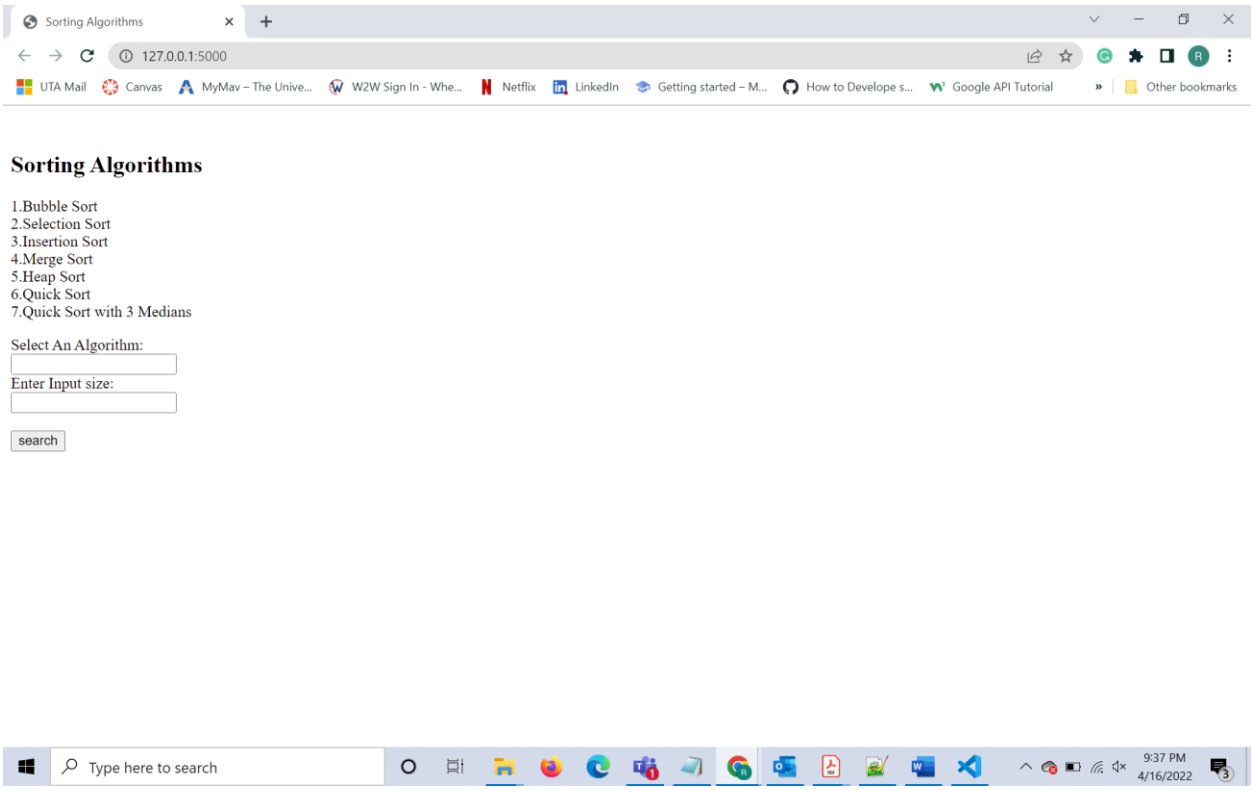
Instructor
Dr. Negin Fraidouni

Introduction:

This project implements and compare the runtime of sorting algorithms based on given input size:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Heap Sort
6. Quick Sort

These algorithms take the various input size from the user and after random array and provide the sorted array.
It calculates the run time required for the algorithm for every sorting algorithm. We can also compare runtime of all sorting algorithms.

User Interface:



1. Bubble Sort

In Bubble Sort, we compare the adjacent pair in the array. If the next element is smaller than the first element, then swap. Keep repeating this step until no swap is required and elements are sorted.

Complexities

If we don't perform any swap in the first iteration. We know that the array is already sorted. Therefore, the time complexity is to be liner O(n). Worst case will occur when array is sorted in the reversed order.

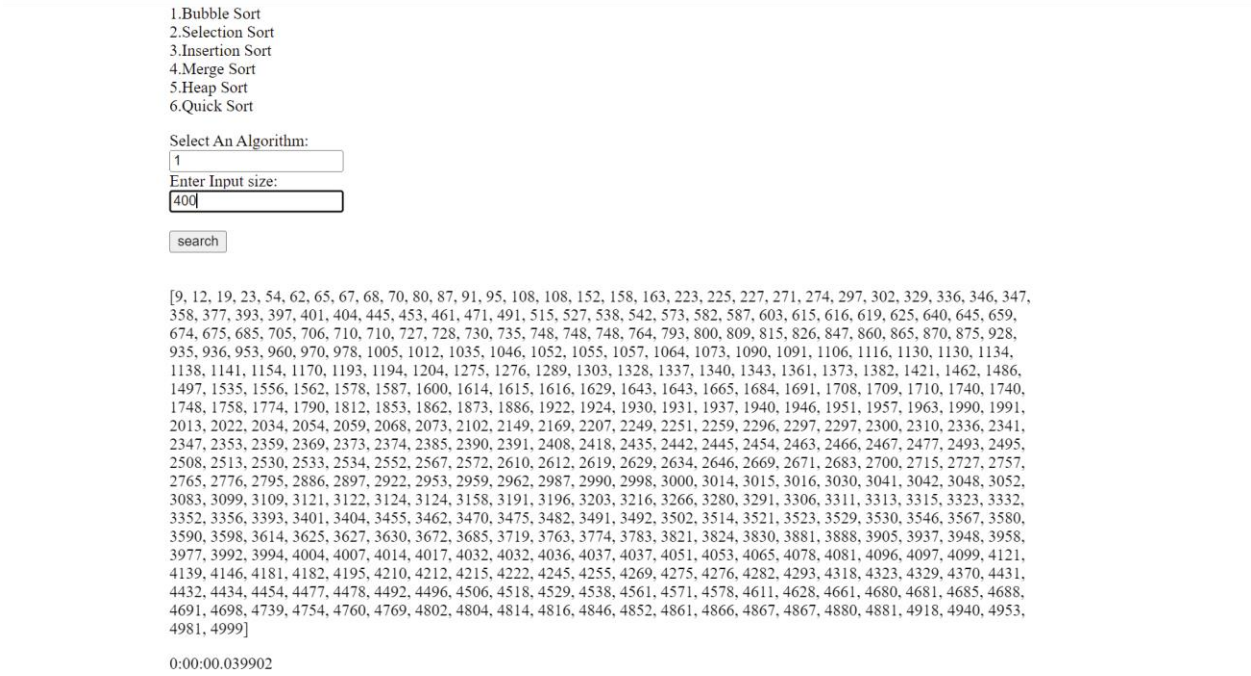
Best case Time Complexity	Average case Time complexity	Worst Case time complexity	Space complexity
O(n)	O(n*n)	O(n*n)	O(n)

Code Snippets:

```
def bblSort(array):
    start_time = datetime.now()
    for i in range(len(array)-1):
        exchange = False
        for j in range(len(array) - 1 -i):
            if array[j] > array[j+1]:
                exchange = True
                t = array[j]
                array[j]=array[j+1]
                array[j+1] = t
    end_time = datetime.now()
    qtime = end_time - start_time
    return array,qtime
```

Data Structure used:

Here, list is used in python as data structure to implement Bubble Sort algorithm.



2. Selection Sort

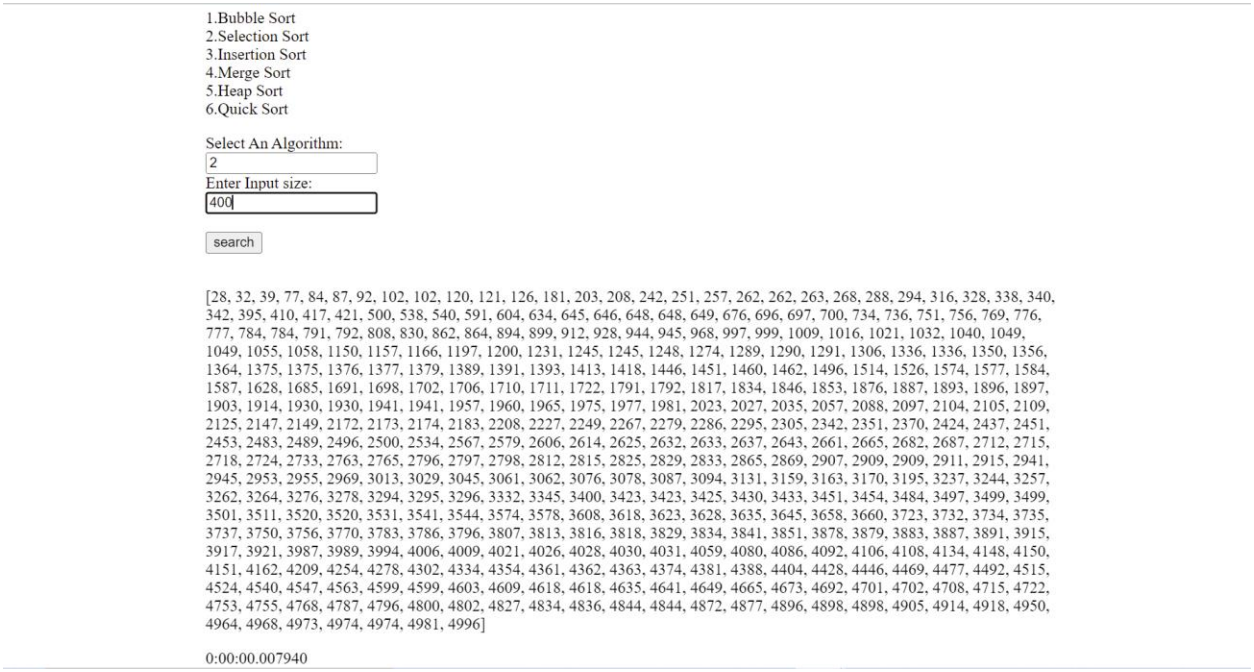
The selection sort algorithm sorts an array by finding the minimum element repeatedly and placing it at the beginning of the list. In every iteration, the minimum element is found from unsorted array and picked and moved to sorted subarray.

Complexities

Best case Time Complexity	Average case Time complexity	Worst Case time complexity	Space complexity
O(n*n)	O(n*n)	O(n*n)	O(1)

Code Snippets:

```
def selectionSort(array):
    start_time = datetime.now()
    for i in range(len(array)-1):
        min = array[i]
        pos = i
        for j in range(i+1, len(array)):
            if array[j]<min:
                min = array[j]
                pos = j
        array[pos] = array[i]
        array[i]=min
    end_time = datetime.now()
    qtime = end_time - start_time
    return array,qtime
```



Data Structure used:

Here, list is used in python as data structure to implement Selection Sort algorithm.

3. Insertion Sort

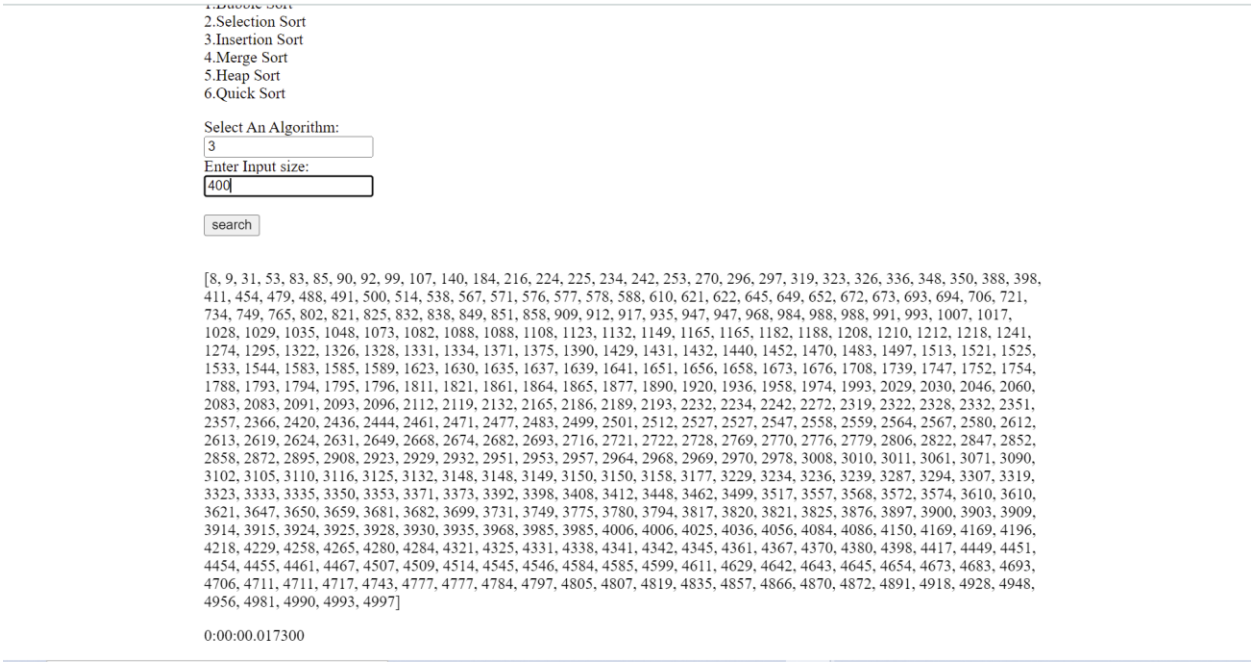
Insertion sort is based on the idea that one element from the list is taken in each iteration to compare its correct position.

Complexities

Best case Time Complexity	Average case Time complexity	Worst Case time complexity	Space complexity
O(n)	O(n*n)	O(n*n)	O(1)

Code Snippets:

```
def insertionSort(array):
    start_time = datetime.now()
    for i in range(1,len(array)):
        val = array[i]
        j=i-1
        while j>=0 and val < array[j]:
            array[j+1] = array[j]
            j-=1
        array[j+1] = val
    end_time = datetime.now()
    qtime = end_time - start_time
    return array,qtime
```



Data Structure used:

Here, list is used in python as data structure to implement Insertion Sort algorithm.

4. Merge Sort

Merge Sort is a divide-and-conquer algorithm. This algorithm divides the array into two halves and performs the mergeSort() on a subarray.

Complexities

Best case Time Complexity	Average case Time complexity	Worst Case time complexity	Space complexity
O(n*logn)	O(n*logn)	O(n*logn)	O(n)

Code Snippets:

```
def mergeSort(array):
    start_time = datetime.now()
    if len(array) > 1:
        m = len(array)//2 #dividing the array
        lb = array[:m]  #dividing the array element
        rb = array[m:]
        mergeSort(lb) #sorting the 1st half
        mergeSort(rb) #sorting the 2nd half
        i = j = k = 0
        while i < len(lb) and j < len(rb): #comparing both arrays
            if lb[i] < rb[j]:
                array[k] = lb[i]
                i +=1
            else :
                array[k] = rb[j]
                j +=1
            k +=1
        while i < len(lb):
            array[k] = lb[i]
            i +=1
            k +=1
        while j < len(rb):
            array[k] = rb[j]
            j +=1
            k +=1
    end_time = datetime.now()
    qtime = end_time - start_time
    return array, qtime
```

1.Bubble Sort
2.Selection Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort

Select An Algorithm:

Enter Input size:

search

[25, 34, 56, 69, 72, 73, 83, 103, 111, 117, 120, 141, 154, 172, 199, 213, 267, 271, 308, 309, 311, 330, 332, 339, 347, 350, 358, 379, 394, 410, 433, 439, 473, 482, 490, 500, 503, 516, 518, 520, 526, 598, 603, 603, 605, 613, 617, 640, 664, 681, 697, 706, 717, 736, 740, 741, 782, 839, 850, 865, 878, 900, 914, 916, 960, 965, 976, 980, 986, 1014, 1017, 1040, 1064, 1068, 1077, 1085, 1114, 1117, 1118, 1130, 1157, 1158, 1177, 1183, 1205, 1207, 1210, 1210, 1219, 1221, 1233, 1237, 1238, 1270, 1274, 1275, 1306, 1307, 1311, 1324, 1353, 1357, 1360, 1361, 1362, 1371, 1371, 1382, 1383, 1386, 1402, 1410, 1430, 1453, 1462, 1465, 1487, 1495, 1526, 1553, 1556, 1572, 1584, 1586, 1592, 1600, 1614, 1618, 1644, 1650, 1650, 1725, 1748, 1762, 1768, 1798, 1818, 1821, 1826, 1848, 1856, 1860, 1873, 1875, 1879, 1879, 1893, 1938, 1957, 1961, 1966, 1974, 1976, 2008, 2033, 2044, 2044, 2086, 2101, 2107, 2109, 2112, 2121, 2124, 2140, 2156, 2166, 2168, 2187, 2196, 2199, 2200, 2217, 2227, 2253, 2289, 2299, 2307, 2346, 2354, 2360, 2375, 2388, 2392, 2418, 2420, 2457, 2461, 2462, 2468, 2473, 2481, 2489, 2533, 2557, 2560, 2581, 2588, 2607, 2639, 2646, 2650, 2682, 2689, 2693, 2704, 2705, 2721, 2730, 2731, 2737, 2757, 2760, 2764, 2764, 2785, 2789, 2792, 2818, 2820, 2843, 2850, 2861, 2872, 2891, 2897, 2909, 2912, 2928, 2961, 2971, 2977, 2983, 2985, 2991, 3003, 3007, 3012, 3043, 3044, 3048, 3051, 3054, 3063, 3076, 3083, 3088, 3088, 3114, 3134, 3153, 3154, 3202, 3224, 3227, 3228, 3229, 3256, 3266, 3267, 3270, 3297, 3329, 3333, 3351, 3372, 3379, 3383, 3400, 3410, 3412, 3430, 3445, 3456, 3476, 3485, 3510, 3538, 3539, 3546, 3556, 3559, 3574, 3585, 3593, 3610, 3611, 3619, 3652, 3684, 3692, 3694, 3700, 3715, 3731, 3734, 3746, 3757, 3759, 3774, 3779, 3788, 3810, 3826, 3832, 3840, 3856, 3862, 3864, 3918, 3963, 3972, 3987, 4006, 4049, 4056, 4061, 4063, 4068, 4100, 4114, 4120, 4132, 4134, 4155, 4175, 4194, 4220, 4223, 4241, 4264, 4280, 4283, 4315, 4316, 4334, 4336, 4337, 4350, 4358, 4376, 4386, 4389, 4397, 4399, 4425, 4428, 4440, 4445, 4478, 4485, 4490, 4500, 4501, 4509, 4515, 4520, 4531, 4552, 4555, 4571, 4577, 4591, 4616, 4623, 4624, 4641, 4656, 4662, 4667, 4705, 4718, 4730, 4736, 4745, 4759, 4765, 4790, 4797, 4818, 4832, 4836, 4857, 4869, 4891, 4894, 4919, 4922, 4923, 4929, 4929, 4937, 4950, 4953, 4955, 4957, 4963, 4988, 4995, 4998]

0:00:00.009118

Data Structure used:

Here, list is used in python as data structure to implement Merge Sort algorithm.

5. Heap Sort

heapify function:

The process of heapify function is it rearrange a binary tree into a heap data structure.

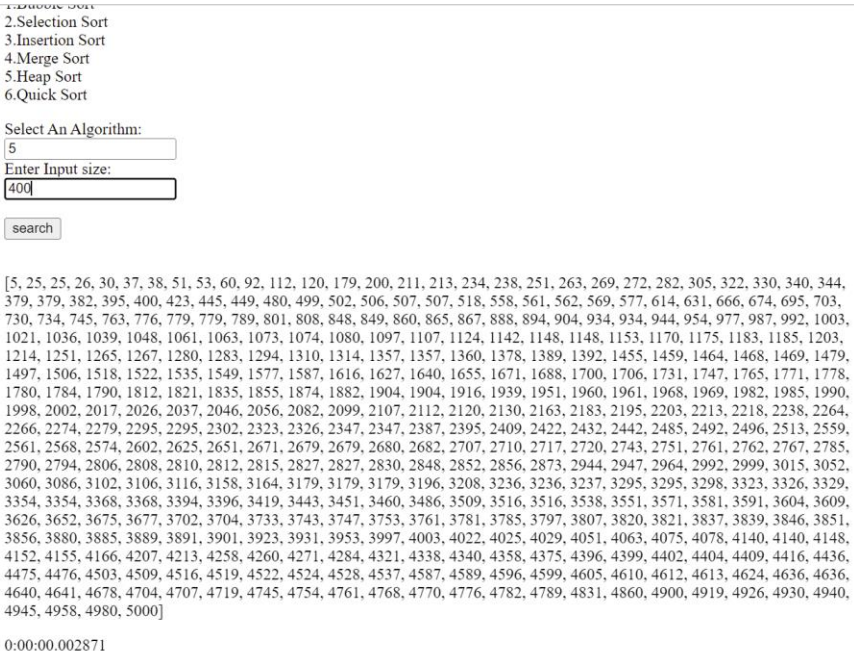
Complexities

Best case Time Complexity	Average case Time complexity	Worst Case time complexity	Space complexity
O(n*logn)	O(n*logn)	O(n*logn)	O(1)

Code Snippets:

```
def heapfiy(array, n, i):
    leftNode = (2*i) + 1
    rightNode = (2*i) +2
    if(leftNode < n and array[leftNode]> array[i]):
        largeElement = leftNode
    else:
        largeElement = i
    if(rightNode < n and array[rightNode]>array[largeElement]):
        largeElement = rightNode
    if(largeElement !=i):
        array[i], array[largeElement] = array[largeElement], array[i]
        heapfiy(array, n, largeElement)
```

```
def buildHeap(array):
    start_time=datetime.now()
    for i in range (len(array), -1, -1):
        heapfiy(array, len(array), i)
    for i in range(len(array)-1, 0, -1):
        array[0], array[i] = array[i], array[0]
        heapfiy(array, i, 0)
    end_time=datetime.now()
    qtime = end_time-start_time
    return array, qtime
```



Data Structure used:

Here, list is used in python as data structure and Heap Data structure to convert into binary tree to implement Heap Sort algorithm.

6. Quick Sort

Quick sort is a divide and conquer approach. An array is divided into subarray by selecting a pivot element.
I have selected last element as a pivot.
The key process in quicksort is partition(). Target of partition() function is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

- Always pick first element as pivot.
- Always pick last element as pivot
- Pick a random element as pivot
- Pick median as pivot.

Complexities

Best case Time Complexity	Average case Time complexity	Worst Case time complexity	Space complexity
O(n*logn)	O(n*logn)	O(n*n)	O(logn)

6.1- Quick Sort with last element as pivot

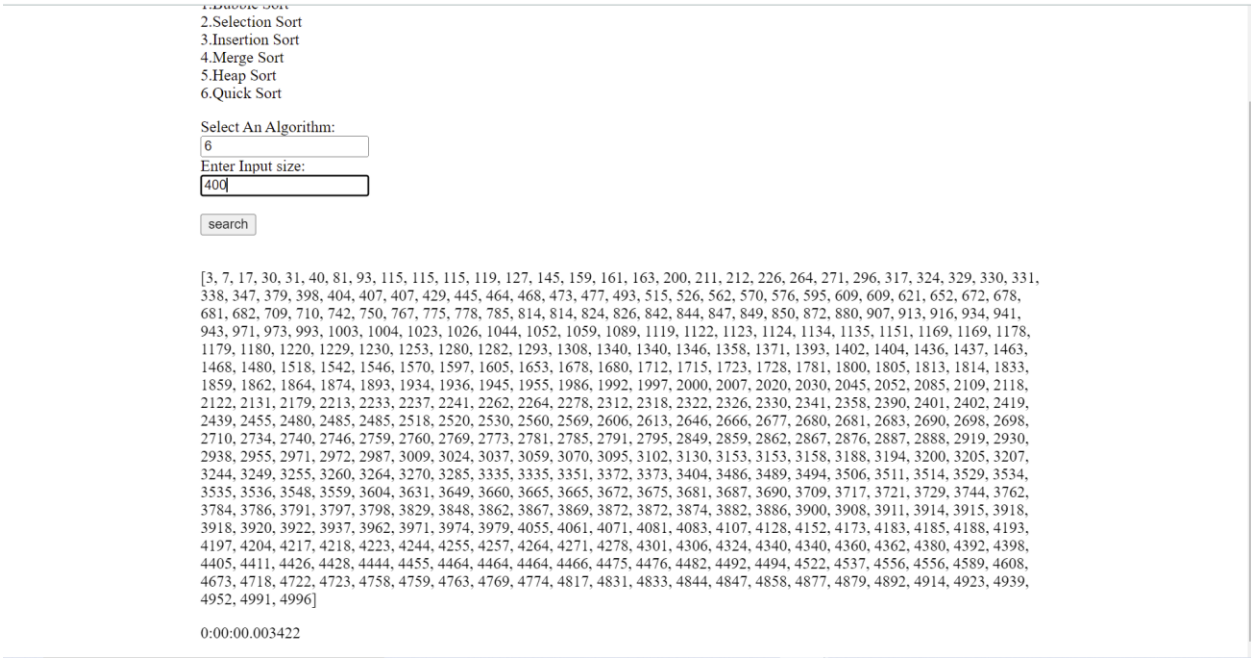
Code Snippets:

```
def partition(array, low, high):
    p = array[high] #pivot
    i=low-1 #smaller element index

    for j in range(low, high):
        if array[j] <= p:
            i=i+1
            array[i], array[j]=array[j], array[i]

    array[i+1], array[high] = array[high], array[i+1]
    return(i+1)
```

```
def quickSort(array, low, high):
    start_time=datetime.now()
    if len(array) == 1:
        return array
    if low<high:
        p1 = partition(array, low, high)
        quickSort(array, low, p1-1)
        quickSort(array, p1+1, high)
    end_time=datetime.now()
    qtime = end_time-start_time
    return array, qtime
```



Data Structure used:

Here, list is used in python as data structure to implement Quick Sort algorithm.

6.2-Quick Sort using 3 medians

Code Snippets:

```
def quickSort2(L, ascending= True):
    quicksorthelp(L, 0, len(L), ascending)

def quicksorthelp(L, low, high, ascending = True):
    result = 0
    if low < high:
        pivot_location, result = Partition(L, low, high, ascending)
        result += quicksorthelp(L, low, pivot_location, ascending)
        result += quicksorthelp(L, pivot_location + 1, high, ascending)
    return result
```

```
def Partition(L, low, high, ascending = True):
    result = 0
    pivot, pidx = median_of_three(L, low, high)
    L[low], L[pidx] = L[pidx], L[low]
    i = low + 1
    for j in range(low+1, high, 1):
        result += 1
        if (ascending and L[j] < pivot) or (not ascending and L[j] > pivot):
            L[i], L[j] = L[j], L[i]
            i += 1
    L[low], L[i-1] = L[i-1], L[low]
    return i - 1, result
```

```
def median_of_three(L, low, high):
    mid = (low+high-1)//2
    a = L[low]
    b = L[mid]
    c = L[high-1]
    if a <= b <= c:
        return b, mid
    if c <= b <= a:
        return b, mid
    if a <= c <= b:
        return c, high-1
    if b <= c <= a:
        return c, high-1
    return a, low
```

Sorting Algorithms

1.Bubble Sort
2.Selection Sort
3.Insertion Sort
4.Merge Sort
5.Heap Sort
6.Quick Sort
7.Quick Sort with 3 Medians

Select An Algorithm:

Enter Input size:

search

[28, 31, 47, 69, 81, 85, 104, 111, 124, 127, 134, 139, 152, 159, 170, 189, 204, 220, 227, 231, 247, 255, 259, 267, 269, 280, 296, 339, 366, 378, 378, 380, 385, 417, 419, 435, 438, 459, 471, 471, 476, 492, 496, 499, 511, 511, 540, 596, 608, 612, 613, 621, 630, 645, 680, 688, 695, 699, 701, 704, 708, 718, 726, 738, 748, 770, 780, 780, 781, 786, 788, 801, 813, 824, 846, 848, 856, 865, 870, 877, 882, 899, 921, 933, 949, 952, 963, 963, 972, 973, 975, 985, 987, 1004, 1008, 1022, 1061, 1113, 1124, 1151, 1165, 1172, 1186, 1212, 1215, 1260, 1267, 1322, 1334, 1344, 1352, 1367, 1418, 1419, 1431, 1432, 1480, 1496, 1524, 1536, 1558, 1567, 1598, 1600, 1639, 1645, 1692, 1693, 1701, 1708, 1717, 1721, 1723, 1737, 1755, 1785, 1786, 1802, 1806, 1820, 1825, 1829, 1829, 1847, 1849, 1851, 1851, 1855, 1867, 1898, 1900, 1902, 1916, 1934, 1943, 1953, 1978, 1981, 1999, 2005, 2011, 2013, 2033, 2053, 2063, 2067, 2068, 2071, 2076, 2084, 2088, 2107, 2132, 2147, 2161, 2200, 2207, 2207, 2221, 2222, 2224, 2227, 2232, 2233, 2246, 2251, 2253, 2256, 2272, 2274, 2284, 2293, 2317, 2328, 2344, 2357, 2374, 2389, 2392, 2395, 2418, 2419, 2437, 2442, 2445, 2466, 2481, 2486, 2486, 2512, 2514, 2524, 2526, 2540, 2572, 2598, 2627, 2629, 2629, 2631, 2645, 2679, 2687, 2687, 2689, 2693, 2697, 2699, 2725, 2731, 2756, 2759, 2780, 2798, 2811, 2822, 2832, 2866, 2872, 2884, 2895, 2904, 2908, 2909, 2921, 2934, 2942, 2981, 2989, 2992, 3044, 3044, 3048, 3049, 3064, 3090, 3096, 3154, 3161, 3189, 3197, 3212, 3216, 3234, 3241, 3251, 3263, 3263, 3276, 3280, 3299, 3303, 3307, 3325, 3356, 3384, 3384, 3392, 3396, 3409, 3431, 3471, 3481, 3491, 3495, 3514, 3515, 3517, 3536, 3552, 3554, 3557, 3562, 3563, 3564, 3586, 3591, 3628, 3636, 3640, 3677, 3689, 3689, 3697, 3705, 3710, 3713, 3714, 3722, 3727, 3729, 3729, 3730, 3731, 3751, 3755, 3760, 3764, 3773, 3775, 3778, 3788, 3793, 3802, 3831, 3837, 3893, 3896, 3899, 3906, 3923, 3926, 3962, 3964, 3967, 3994, 4074, 4088, 4091, 4097, 4098, 4109, 4119, 4175, 4192, 4193, 4204, 4213, 4261, 4299, 4302, 4303, 4319, 4319, 4333, 4341, 4343, 4365, 4383, 4396, 4399, 4400, 4403, 4412, 4436, 4446, 4466, 4477, 4497, 4503, 4526, 4533, 4564, 4566, 4620, 4623, 4625, 4629, 4648, 4650, 4653, 4665, 4680, 4688, 4697, 4710, 4719, 4745, 4767, 4841, 4842, 4849, 4869, 4893, 4910, 4913, 4926, 4954, 4957, 4986]

0:00:00.000491

Runtime Comparison

- For comparing the Runtime between all sorting algorithms, I took various input size such as 50, 100, 500, 1000, 5000, 10000 on x-axis and runtime for each sorting algorithm on Y-axis.
- Bubble Sort is quicker for small data sets(input size 50) than any other algorithms.
- I have observed one thing, for big input size(for example, input size 10000) the quick sort is more efficient(runtime is around 0.53972) and Bubble sort and Selection sort takes more time (around 2-3 secs).

test1.py > ...

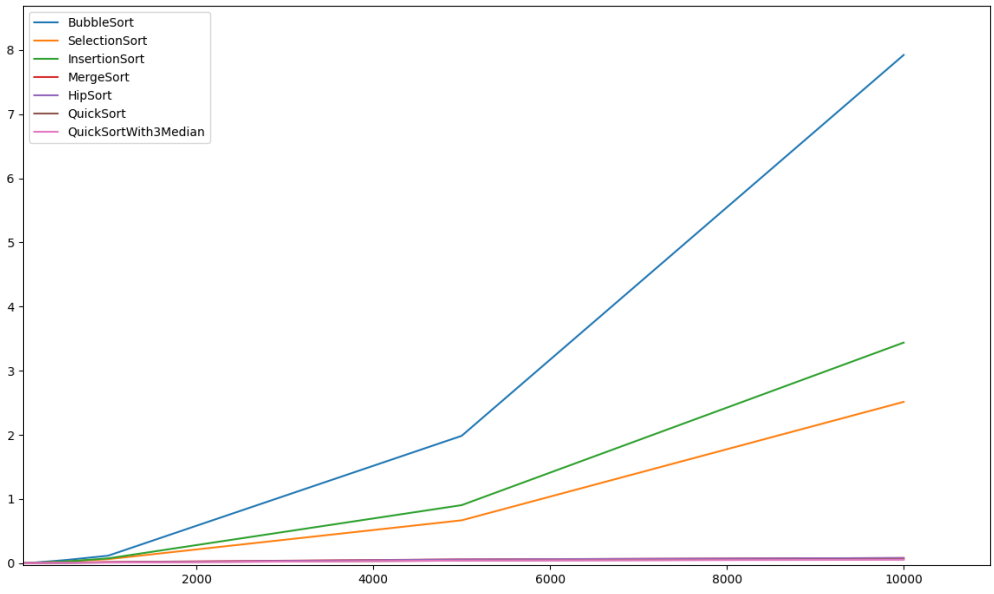
```python
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([50, 100, 500, 1000,5000,10000])
ypoints = np.array([.000603,.002309,.046131,.116293,1.984896,7.921630])
y2 = np.array([.000244,.000931,.024447,.062882,.668152,2.514627])
y3 = np.array([.000325,.001189,.025219,.074623,.904756,3.438328])
y4 = np.array([.000633,.001656,.007157,.017398,.059195,.080384])
y5 = np.array([.000372,.000865,.006320,.012559,.057591,.081131])
y6 = np.array([.000402,.000812,.004316,.011832,.040166,.061523])
y7 = np.array([.000361,.000631,.003773,.009068,.040349,.053792])


plt.plot(xpoints, ypoints)
plt.plot(xpoints,y2)
plt.plot(xpoints,y3)
plt.plot(xpoints,y4)
plt.plot(xpoints,y5)
plt.plot(xpoints,y6)
plt.plot(xpoints,y7)


plt.legend(["BubbleSort", "SelectionSort","InsertionSort","MergeSort","HipSort","QuickSort","QuickSortWith3Median"])
plt.show()
```

Graph:


BubbleSort
SelectionSort
InsertionSort
MergeSort
HipSort
QuickSort
QuickSortWith3Median
0
1
2
3
4
5
6
7
8
2000
4000
6000
8000
10000

References:

https://www.programiz.com/dsa/merge-sort
https://coderslegacy.com/python/heap-sort-algorithm/
https://www.programiz.com/dsa/heap-sort
https://www.geeksforgeeks.org/python-program-for-quicksort/
https://www.programiz.com/dsa/quick-sort
https://stackabuse.com/quicksort-in-python/
https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms