Project Computational Algorithms

Author: Silvio Dunst
Student No: G00388192

Introduction of Sorting Algorithms

Sorting Algorithms are importing steps of many computational algorithms for instance in computer graphics, phone books, web searches like Google, bank transaction etc. The sorting algorithms puts elements of a list's/array's or collections in a certain order. Sorting algorithm has pre-defined ordering rules. The most frequently used orders are numerical and lexicographic order. The numerical order is a mathematical order and the lexicographic order is an alphabetical order. Many computational tasks are simplified by pre-sorting data's in advance e.g. find duplicate data's, find the frequency of data, find the maximum ,minimum and median values of a data set etc. A collection of data deemed to be sorted when the data are less or equal to his successor. The output of the sorted data must be a permutation. It means the reordering of the data collection, but retaining all the originally elements of the input. Duplicate data's must be in a contiguous order no other element can be between them.
Many sorting algorithms comparing items in a collection. If the items are numerical values it can be sorted by less or greater than the previous numerical value. If the items are lexicographic values the items can be sorted by the string characters. To sort lexicographic values, the ASCII character table can be used. The comparison sorting algorithms determine which item or element of two items should appear first. For comparison algorithms we can use comparator functions what de-terms what data in a collection are lower or higher.
Comparator function takes two arguments or inputs and contains logic to it to decide their relative order for the sorted output. We could use one comparator function if value "a" is lower than value "b", we could write -1 (-1 if a<b) or if value "a" is equals to value "b", we could write 0 (0 if a =b). Another comparator function would be if value "a" is greater than value "b", we could write 1 (1 if a>b).
A comparator function deemed to be stable if two elements with the same keys or values appear in the same order in the sorted output as they appear in the sorted input array. Comparator functions deemed to be stable as well if a value "a" is smaller than value "b" before sorting and the sorting algorithm maintains the same order after sorting. If this can't be achieved the sorting algorithm deemed to be unstable. What can happens if using an unstable sorting algorithm that an already sorted input will be changed and written in the output. Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values) and we like to sort these objects by keys[4].
The comparisons algorithms are the most widely used sorting algorithms. Many of the well know sorting algorithms like Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort
and Heap Sort are comparisons based. In analysing comparison algorithms a very important result is that no comparison algorithm can do better than n (linear) or log n (logarithm) performance in the best, average or worst case scenario.
Under some special condition it is possible to design different type of non-comparison algorithms what can have better time worst case scenarios. Some examples would be Counting Sort, Radix Sort and Bucket Sort algorithm. Non-comparison sort algorithm uses the internal character of the values to be sorted.

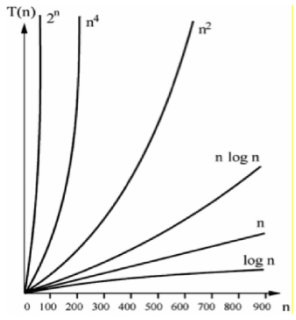
Non comparison sorting algorithm

It is important to know about comparison/non-comparison sort algorithms. If we use a comparison sort algorithm then on each comparison we will split the set of possible outcomes roughly in half (because the output is binary) thus the best complexity we can possibly have is O(log(n!)) = O(n*log(n)). This restriction does not hold for non-comparison sorts.

For analysing different sorting algorithms a good indicator is to use the concept of time complexity. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. The maximum amount of time what is needed for a certain input of data collection to be sorted is called worst case time complexity. A less common used is the average time complexity. The average time complexity describes the average time it takes to run the sorting algorithm for a certain amount of input data. The average case is mostly the hardest to quantify. It relies on advanced mathematical techniques and estimations. It assumes that may be the inputs already partially sorted. We have as well the best time complexity where the shortest time is needed to run a certain collection of data. Normally we are only interested on worst case time complexity. Another important factor is the so called “Space complexity”. Space complexity described the amount of extra memory (space) needed for an algorithm to run. Not included in the space complexity is the memory needed for the storing the input data themselves. Different sorting algorithms require different memory space. If a sorting algorithm requires a fixed amount of additional working space independent of the input size when we called it in-place sorting. Some algorithms need additional memory space, the amount is often related to the input size n. If the availability of the memory space is a concern in-place sorting is a welcome property.

The less time and less extra memory space is required to run a sorting algorithm the better is the performance.
Therefore, the time and space complexity is commonly expressed using big O notation. We have different big O notation performances like O(n) (linear) or O(log n) (logarithm) or O(2^n) (exponential), where n is the input size in units of bits needed to represent the input [3]. See picture below

Running time $T(n)$	
is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential



Sometimes it can be impractical to use certain sorting algorithms for instance an Insertion sort with a large amount of input data’s. No sorting algorithm is the best for all situations. It is important to understand the strengths and weaknesses of the algorithms.

Some examples of sorting algorithms

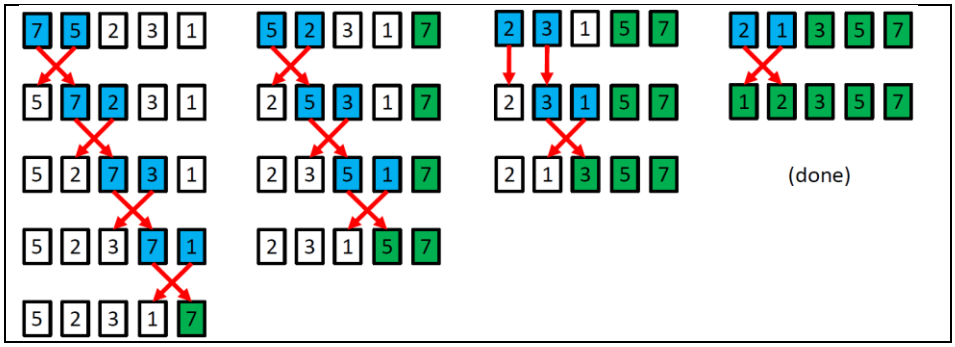
Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$O(n)$	Yes
Quicksort	$n\ log\ n$	n^2	$n\ log\ n$	n (worst case)	No*
Heapsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Timsort	n	$n\ log\ n$	$n\ log\ n$	n	Yes
Introsort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$	$log\ n$	No

Others factors what can influence the running time of a sorting algorithm are the computer hardware like internal computer memory, cache size. If the computer memory and cache size is too low the sorting might has to happen on the slower hard drive so called external sorting. Another factor is the amount of items what has to be sorted or are the items already pre-sorted or have they already a certain order [1],[2].

Bubble Sort

Bubble sort algorithm is one of the simplest sorting algorithms. It became his name when using larger sample size the values in samples “bubble up” at the end of the sorting. It was the first time analysed in the 1950’s. Bubble sort algorithm is a comparison sorting algorithm with best case time complexity O notation of n(linear) for small sample sizes and worst and average time complexity O notation of n^2(squared) for bigger sample sizes. Bubble sort is a in place sorting algorithm with a space complexity O(1) order of 1. It uses constant amount of memory space in addition to the memory space is required for the data inputs. Bubble sort is a stable sort algorithm.

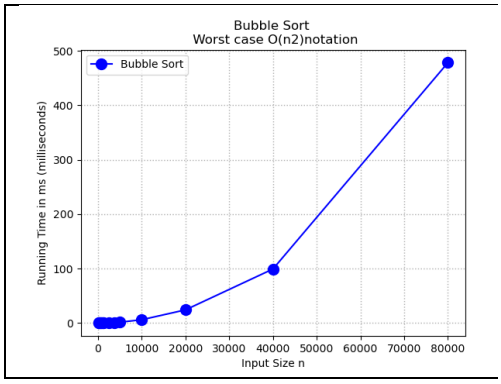
Bubble sort is easy to understand and to implement but is impractical for most application especial on bigger data samples because of the amount of time it takes to sort the data with his n^2(squared) time complexity. It compares each element except the last one with his neighbour to the right. If the elements out of order it swaps them. This puts the largest element at the end what is now in the correct place. In the next loop it does the same as before except the last two elements. This puts the second large element beside the largest one. This continues until all input data are sorted.



Bubble Sort Python Code

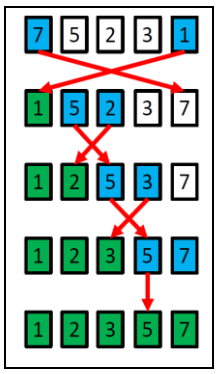
```
# Bubble Sort
# Use the function from GMIT lecture and adopted this to my purpose
def bubble_sort(bubblearr):
    n = len(bubblearr)
    for outer in range(n-1, 0, -1): # starts at the end of the array n-1 and goes to the start of the array 0 decrease every time by -1
        for inner in range(0,outer,1):#so called bubbling up here
            if bubblearr[inner] > bubblearr[inner+1]:# if not in order swap in the next line
                temp = bubblearr[inner]
                bubblearr[inner] = bubblearr[inner+1]# if in order write value in temp variable in the next line
                bubblearr[inner+1] = temp
```

Bubble Sort plot



Selection Sort

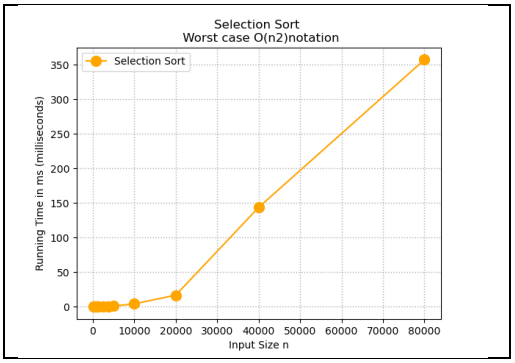
Selection sort is a simpler sorting algorithm. Selection sort algorithm belongs to the comparison algorithms with best, average and worst case time complexity O notation of n^2(squared) for all sample sizes. Selection sort algorithm is a in place sorting algorithm with a space complexity O(1) order of 1.
Selection sort is an unstable sorting algorithm. Selection sort is easy to understand and to implement. Selection sort is performing better than Bubble sort but is still not practical for a majority of applications especial on larger sample sizes.
The Selection sort iterate through the input data. In every iteration the minimum element when used in ascending order from unsorted sub array on the right is picked and moved over to the sorted sub array on the left. The Selection sort picks the first element (position 0) from the data input array and searches through the full length of the array/ data input for the element with the smallest value. When it found the element with the smallest value it swaps it with the first element (position 0). In the next iteration it picks the next array element (position1) and searches to the rest of the array for the element with the smallest value. When the smallest value is found it swaps it with the element on array position (1). This continues with the next element on position 2 and so on until nothing is left to search.



Selection Sort Python Code

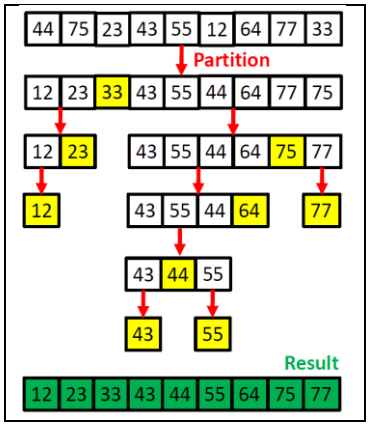
```
# Selection Sort
# Use the function from GMIT lecture and adopted this to my purpose
def selection_sort(selectionarr):
    n = len(selectionarr)
    for outer in range(0,n,1):# the loop runs through each element of the data input array
        min = outer
        for inner in range(outer+1,n,1): # it increments by 1 (outer+1)and searches in every iteration for the smallest element
            if selectionarr[inner] < selectionarr[min]:# if the next element is smaller we write the value in min variable next line if not jump to temp line
                min = inner # and running through the rest of the array

        temp = selectionarr[outer] # we write the value of first array element in the temp variable
        selectionarr[outer] = selectionarr[min] # we swap write the minimum value of the array to the first element of the array
        selectionarr[min] = temp # we swap write the temp variable value as a minimum value in the array
```

Selection Sort plot



Quick Sort

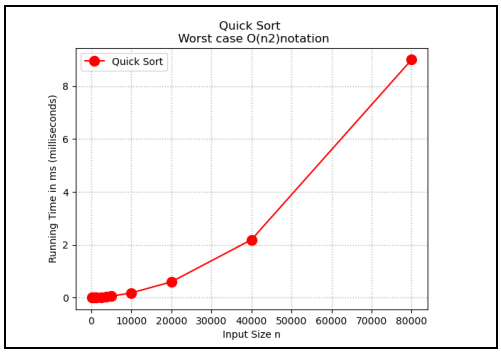
The Quick sort algorithm is another comparison algorithm. It was developed in 1959 by the British computer scientist Charles Anthony Richard Hoare. The Quick sort algorithm has in the best and average case a time complexity O notation of O(n log n) logarithm function. In the worst case time complexity O notation of n^2 (squared) function. The memory space the Quick sort algorithm is using is depending of the Quick sort algorithm type. It can vary from O notation of O(n log n) logarithm function until O notation of O(n) linear function. Quick sort is unstable in the standard version but some stable Quick sort versions exist. The Quick sort algorithm is one of the fastest sorting algorithms.
The Quick sort algorithm uses the recursive so called “divide and conquer” principle. Through kind of portioning process the data’s are divided roughly into small and large elements. The small elements move to the left side and the larger elements move to the right side. Pick an element so called pivot anywhere in the data array but preferable in the middle of the array. All the elements with the values smaller than the pivot moving into the smaller partitioning section and all values greater than the pivot moving into the bigger partitioning section. Repeat this process recursively again to pick a pivot from each sub array and divide and sub array in smaller and bigger partitioning section. Repeat this until only one element in the sub array is left. When the deepest level of the recursion for all partitions is reached the array or data collection deemed to be sorted.



Quick Sort Python Code

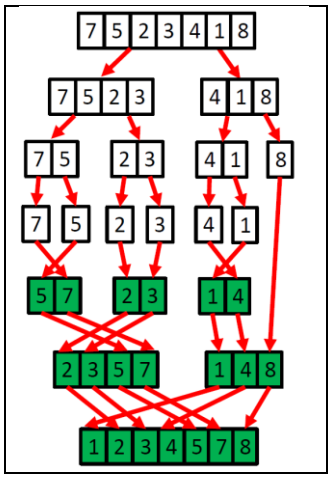
```
# Quick Sort
#found this code on https://www.educative.io/edpresso/how-to-implement-quicksort-in-python and adopted this to my purpose
def quick_sort(quickarr):
    elements = len(quickarr)

    #Base case
    if elements < 2:
        return quickarr

    current_position = 0 #Position of the partitioning element

    for i in range(1, elements): #Partitioning loop
        if quickarr[i] <= quickarr[0]:
            current_position += 1
            temp = quickarr[i]
            quickarr[i] = quickarr[current_position]
            quickarr[current_position] = temp

    temp = quickarr[0]
    quickarr[0] = quickarr[current_position]
    quickarr[current_position] = temp #Brings pivot to it's appropriate position

    left = quick_sort(quickarr[0:current_position]) #Sorts the elements to the left of pivot
    right = quick_sort(quickarr[current_position+1:elements]) #sorts the elements to the right of pivot

    quickarr = left + [quickarr[current_position]] + right #Merging everything together

    return quickarr
```

Quick Sort plot



Merge Sort

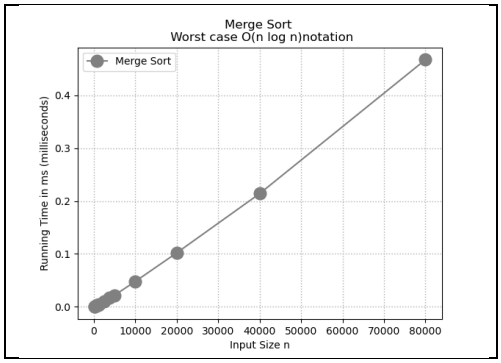
Merge sort algorithm is a comparison algorithm. It was proposed by John von Neumann in 1945. The Merge sort algorithm has for the best, average and worst case time complexity O notation of O(n log n) logarithm function. The same time complexity for best, average and worst case makes the Merge Sort algorithm very predictable for the running time. The Merge sort algorithm has a memory space of O notation of O(n) linear function and is a stable sorting algorithm. Merge sort algorithm is very useful for applications with data that have slow access times and data what can not held in a internal RAM memory or linked lists. Merge sort algorithm is one of the fastest sorting algorithms.
The Merge sort algorithm uses also the recursive “divide and conquer” principle. It divides a data array into two separate sub arrays of equal or nearly equal sizes in every iteration. The sub arrays will be divided so long only sub arrays with one element exist. In the next step the neighbouring sub arrays will be merged and sorted. This principle will happen now in reverse order. It will continues until only one data array is left. Now we have all the values from the original data input in a sorted array.



Merge Sort Python Code

```
# Merge sort
# found this code on https://www.geeksforgeeks.org/merge-sort/ and adopted this to my purpose
def merge_sort(mergearr):
    if len(mergearr)>1:
        mid = len(mergearr)//2 # Finding the mid of the array
        left = mergearr[:mid] # Dividing the array elements into left from middle
        right = mergearr[mid:] # Dividing the array elements into right from middle
        merge_sort(left) # Sorting the left half
        merge_sort(right) # Sorting the right half
        i = j = k = 0 # set the array index i,j,k to 0

        while i < len(left) and j < len(right):# Copy data to temp arrays left[] and right[]
            if left[i] < right[j]:
                mergearr[k] = left[i]
                i = i + 1 # increment the index i for the left side
            else:
                mergearr[k] = right[j]
                j = j + 1 # increment the index j for the right side
            k = k + 1 # increment the index k for the mergearr

        while i < len(left): # Checking if any element was on the left side
            mergearr[k] = left[i] # write the left index i value in the mergearr on the appropiate position
            i = i + 1 # increment the index i for the left side
            k = k + 1 # increment the index k for the mergearr

        while j < len(right): # Checking if any element was on the right side
            mergearr[k] = right[j] # write the right index j value in the mergearr on the appropiate position
            j = j + 1 # increment the index j for the right side
            k = k + 1 # increment the index k for the mergearr
```

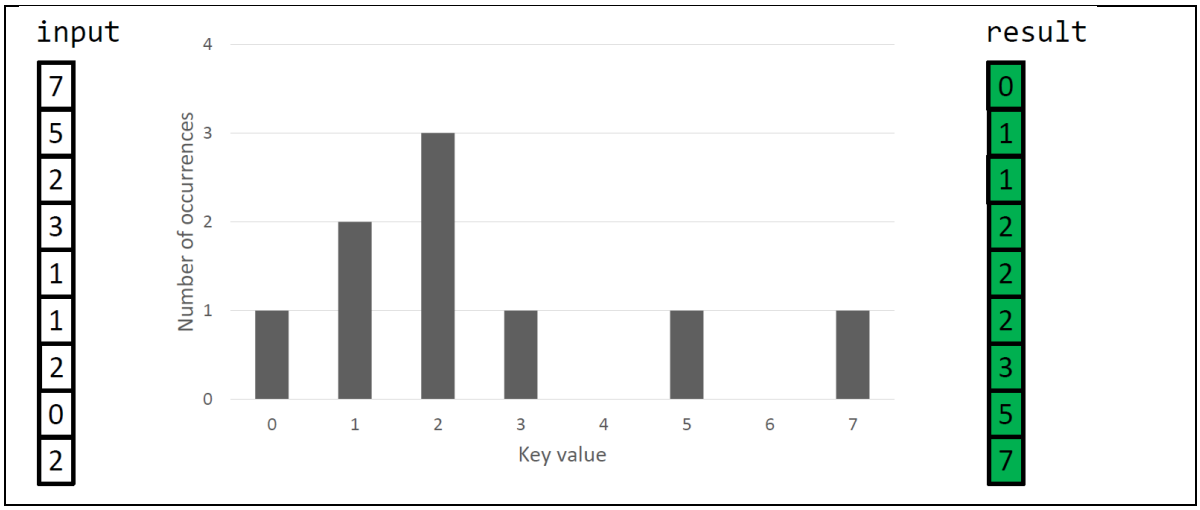
Merge Sort plot



Count Sort

Count sort algorithm is a non-comparison algorithm. The algorithm was proposed and developed by Harold H. Seward in 1954. With Count sort it is possible or nearly possible to sort a data collection close to the best, average and worst time complexity O notation of O(n+k) linear function. The variable k stands for keys. Key means in this case the number of possible values. How does it work? Instead of comparing elements, Counting Sort counts how often each elements occur in the data collection. Counting Sort can be used in a simplified version when sorting integer numbers. They will be sorted to their keys. Count sort requires a memory space of O notation of O(n+k) linear function. The potential advantage of a fast running time comes at a cost what it can not be used as widely as a comparison sorting algorithm. Count sort algorithm belongs to the stable algorithms if used in a correct way. Count sort algorithm is one of the fastest sorting algorithms.
First we have to create new array for the keys with a length of all the different values in the data collection. In the next step we have to create another output/result array where the values (e.g. the amount of each number) of each key been stored in. This array will be used to store the sorted data collection at the end of the sorting algorithm. The algorithm start to iterate through the data collection and counts how often the same value as a key appears in the data collection. It takes the value of the keys and writes it in the created output/result array. In the last step we iterate over the

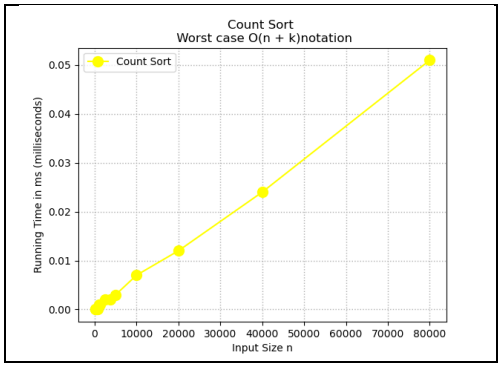
output/result array what is a histogram array what show the frequency the values appears in the data collection and write in the values back in a sorted manner in the output/result array.



Count Sort Python Code

```python
# Counting sort
# https://www.geeksforgeeks.org/counting-sort/ and adopted this to my purpose

def count_sort(countarr):
    max_element = int(max(countarr))
    min_element = int(min(countarr))
    range_of_elements = max_element - min_element + 1

    count_arr = [0 for _ in range(range_of_elements)]# Create a count array to store count of individual elements and initialize count array as 0
    output_arr = [0 for _ in range(len(countarr))]# Create a output array to store the count array of individual elements and initialize output array as 0

    for i in range(0, len(countarr)): # Store count of each count array element
        count_arr[countarr[i]-min_element] += 1

    for i in range(1, len(count_arr)): # Change count_arr[i] so that count_arr[i] now contains actual position of this element in output array
        count_arr[i] += count_arr[i-1]

    for i in range(len(countarr)-1, -1, -1): # Build the output array
        output_arr[count_arr[countarr[i] - min_element] - 1] = countarr[i]
        count_arr[countarr[i] - min_element] -= 1

    for i in range(0, len(countarr)):# Copy the output array to arr, so that arr now contains sorted characters
        countarr[i] = output_arr[i]
```

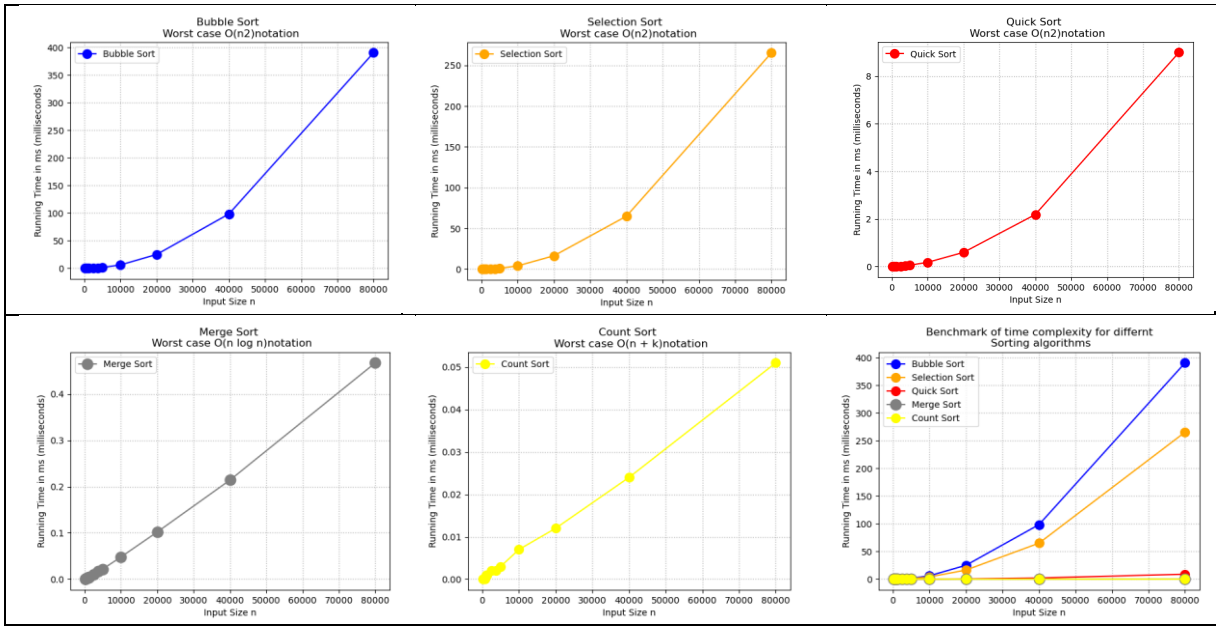
Count Sort plot

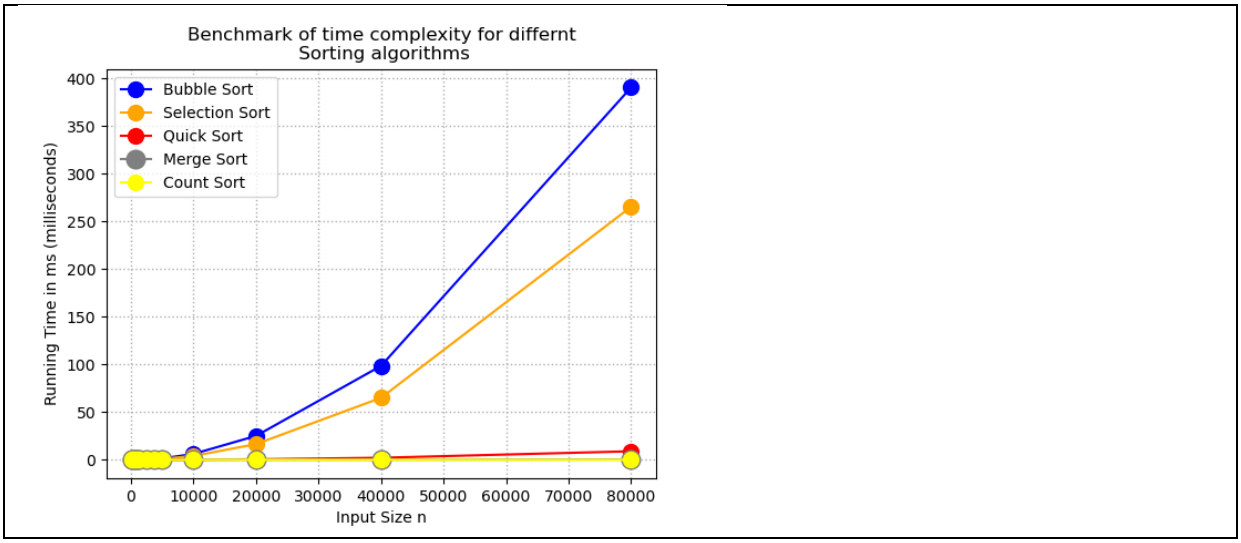


Benchmarking

Benchmarking analysis in the computer industry is a methodology of comparing the performance of computer algorithms. Sometimes is called a posteriori analysis as well what means to gain knowledge from experiments. The experimental data will be used to validate and compare the data with the theoretical assumption. The computer hardware and software, computer system architecture, CPU design, background performances as well energy saving technologies influence the running time of algorithms. Therefore it is very important to run the same algorithm under the same experimental conditions several times to conduct an average. In our benchmarking process we measure the time (run time) it takes to sort randomly created data with different sorting algorithms. The data collections have different sample size ranging from 100 until 80000 samples. To measure the run time of the algorithm the python time module is used. The python time module measure the time when the algorithm start and measure the time when the algorithm ends. The start time will be subtracted from the end time what gives us the running time of the algorithm as a result. This process will be repeated 10 times for each sample and for each algorithm.

Sample Size	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**3750**	**5000**	**1000**	**20000**	**40000**	**80000**
Bubble Sort	0.001	0.004	0.017	0.033	0.058	0.092	0.379	0.888	1.519	6.214	25.118	98.515	390.446
Selection Sort	0.001	0.003	0.01	0.024	0.042	0.064	0.263	0.577	1.028	4.120	16.478	65.035	265.352
Quick Sort	0.000	0.002	0.003	0.005	0.007	0.009	0.022	0.039	0.058	0.181	0.601	2.193	8.999
Merge Sort	0.000	0.001	0.002	0.003	0.004	0.005	0.010	0.017	0.022	0.048	0.102	0.214	0.467
Count Sort	0.000	0.000	0.000	0.000	0.001	0.001	0.002	0.002	0.003	0.007	0.012	0.024	0.051





Sample Size	100	250	500	750	1000	1250	2500	3750	5000	1000	20000	40000	80000
Bubble Sort	0.001	0.004	0.017	0.033	0.058	0.092	0.379	0.888	1.519	6.214	25.118	98.515	390.446
Selection Sort	0.001	0.003	0.01	0.024	0.042	0.064	0.263	0.577	1.028	4.120	16.478	65.035	265.352
Quick Sort	0.000	0.002	0.003	0.005	0.007	0.009	0.022	0.039	0.058	0.181	0.601	2.193	8.999
Merge Sort	0.000	0.001	0.002	0.003	0.004	0.005	0.010	0.017	0.022	0.048	0.102	0.214	0.467
Count Sort	0.000	0.000	0.000	0.000	0.001	0.001	0.002	0.002	0.003	0.007	0.012	0.024	0.051

References

[1] Lecturer notes
[2] https://en.wikipedia.org/wiki/Sorting_algorithm
[3] https://en.wikipedia.org/wiki/Time_complexity
[4] https://www.geeksforgeeks.org/stability-in-sorting-algorithms/
[5] https://www.happycoders.eu/algorithms/sorting-algorithms/