Report
Benchmarking Sorting Algorithms

Computational Thinking with Algorithms

G00376368 – Karolina Szafran-Belzowska

Table of contents

1. Introduction

1.1. The concept of sorting

A sorting algorithm is a methods for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest or shortest-to-longest distance. Sorting algorithms take a list of items as input data, perform specific operations on those lists and deliver ordered arrays as output.[1]

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.[2][3]

Choosing the best sorting algorithm is as about knowing what you are sorting as it is about the relative performance of the algorithms.

The output of any sorting algorithm must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element),
2. And the output is a permutation (a reordering of the original elements) of the input.

The input data is often stored in an array, which allows random access, rather than a list, which only allows sequential access, though many algorithms can be applied to either type of data after suitable modification.

Sorting algorithms are often classified by:

- **Computational complexity** (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is $O(n \log n)$, with parallel sort in $O(\log^2 n)$, and bad behavior is $O(n^2)$. Ideal behavior for a serial sort is $O(n)$, but this is not possible in the average case. Optimal parallel sorting is $O(\log n)$. Comparison-based sorting algorithms need at least $\Omega(n \log n)$ comparisons for most inputs.
- **Computational complexity of swaps** (for "in-place" algorithms).
- **Memory usage** (and use of other computer resources). Some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes $O(\log(n))$ additional memory is considered "in-place".

- **Recursion.** Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- **Stability:** stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are **a comparison sort.** A comparison sort examines the data only by comparing two elements with a comparison operator.
- **General method:** insertion, exchange, selection, merging, *etc.* Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
- Whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
- **Adaptability:** Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

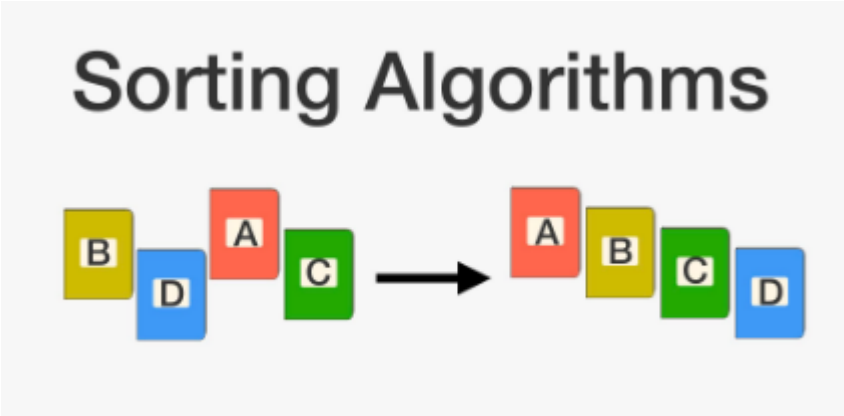


Image 1: Sorting algorithms: https://brilliant.org/wiki/sorting-algorithms/ on 08/05/2020

1.2. Time and Space Complexity

The complexity of an algorithm is the function f(n) which gives the running time and storage space requirement of the algorithm in terms of the size n of the input data. Complexity refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. **Best Case** : The minimum possible value of f(n) is called the best case.
2. **Average Case** : The expected value of f(n).
3. **Worst Case** : The maximum value of f(n) for any key possible input.

Best, worst, and average cases of a given algorithm express what the resource usage is *at least*, *at most* and *on average*. Average performance and worst-case performance are the most used in algorithm analysis.[4]

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input. To compare standard sorting algorithms on the basis of space, the Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.[5]

1.2.1. Bubble Time complexity

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Bubble sort takes minimum time (Order of n) when elements are already sorted.

1.2.2. Selection Sort Complexity

Time Complexity: O(n^2) as there are two nested loops.

Auxiliary Space: O(1)

Selection sort never makes more than O(n) swaps and can be useful when memory write is a costly operation.

1.2.3. Insertion Sort Complexity

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

1.2.4. Merge Sort Complexity

Time Complexity: Sorting arrays on different machines.

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + $\Theta(n)$

Time complexity of Merge Sort is $\Theta(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

1.2.5. Counting Sort Complexity

Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.

Auxiliary Space: O(n+k)

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

Image 2: Time complexity. Taken from: https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/ on 08/05/2020

1.2.6. Big O Complexity

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.[6] Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used by an algorithm.

Below are examples of Big O notation[7][8]:

1. O(1), constant time algorithm - describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
[a,b,c,d,e] => [a,b,c,d]
```

2. O(N), linear time algorithm - describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
[a,b,c,d,e] => [aa,bb,cc,dd,ee]
```

3. O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N^3), O(N^4) etc.
   If the list has two letters in it the program will take four operations to run, but if the list has four trillion letters, it may never finish running.

```
[a,b,c,d,e] => [abcde, bacde, cabde, dabce, eabcd]
```

4. O(2^N) denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an O(2^N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2^N) function is the recursive calculation of Fibonacci numbers.
5. Logarithms, logarithmic time algorithm - **Binary search** is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set. This type of algorithm is described as **O(log N)**. The bigger the input, the smaller proportion of the actual input your program has to go through.[9]

Orders of growth

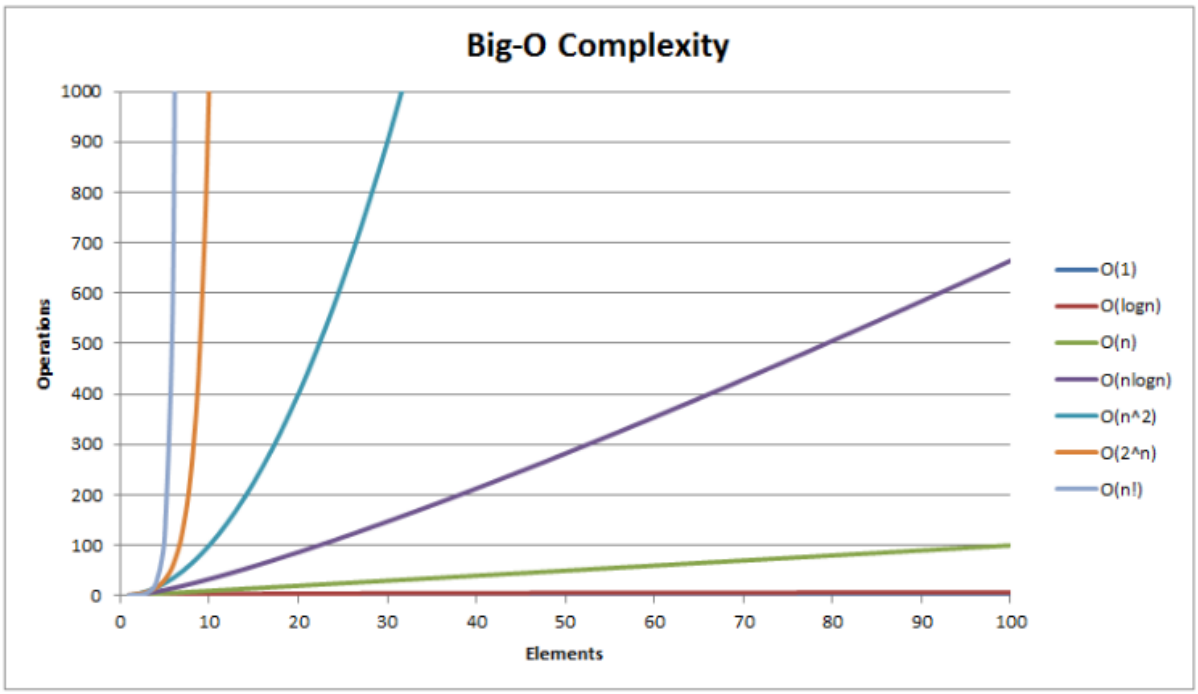


Image 3: Big O Complexity. Taken from: https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/ on 08/05/2020

1.3. In-place algorithm

In-place algorithm is an algorithm which transforms input using no auxiliary data structure. However a small amount of extra storage space is allowed for auxiliary variables. The input is usually overwritten by the output as the algorithm executes. In-place algorithm updates input sequence only through replacement or swapping of elements. An algorithm which is not in-place is sometimes called not-in-place or out-of-place.

An in-place algorithm is an algorithm that does not need extra space and produces an output in the same memory that contains the data by transforming the input “in-place”.

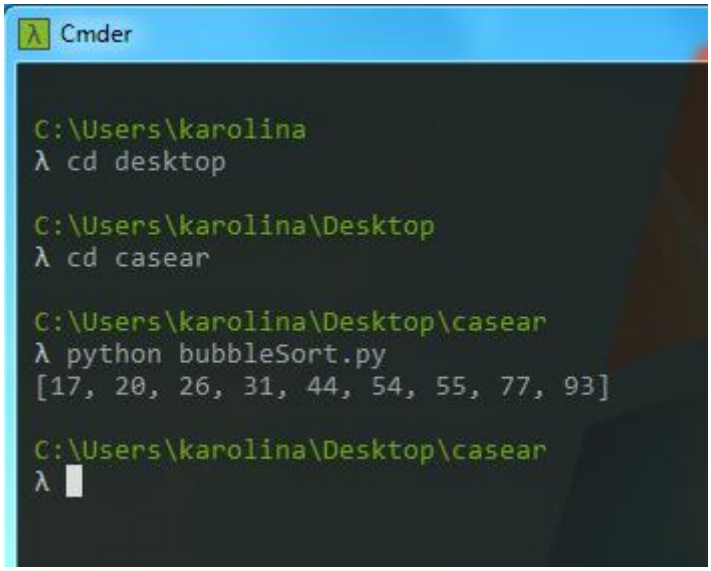
An example of in-place algorithm.

C: > Users > karolina > Desktop > casear > bubbleSort.py

```
1   def bubbleSort(alist):
2       for passnum in range(len(alist)-1,0,-1):
3           for i in range (passnum):
4               if alist[i]>alist[i+1]:
5                   temp = alist[i]
6                   alist[i] = alist[i+1]
7                   alist[i+1] = temp
8
9
10  alist = [54,26,93,17,77,31,44,55,20]
11  bubbleSort(alist)
12  print(alist)
13
```

Image 4: Taken from the module Computational Thinking with Algorithms (week 10), GMIT

And the result of the code above is:



An example of in-place algorithms: Bubble sort, Selection Sort, Insertion Sort, Heapsort, QuickSort.

Not In-Place : Merge Sort.

1.4. Stable sorting

A stable sorting algorithm is said to be sorted if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

The stability of a sorting algorithm is concerned with how the algorithm treats equal (or repeated) elements. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't.

All sorting algorithms use a key to determine the ordering of the elements in the collection, called the *sort key*. [10]

Stable Sorting Algorithms: Insertion Sort, Merge Sort, Bubble Sort, Tim Sort, Counting Sort.

Unstable Sorting Algorithms: Heap Sort, Selection Sort, Shell Sort, Quick Sort.

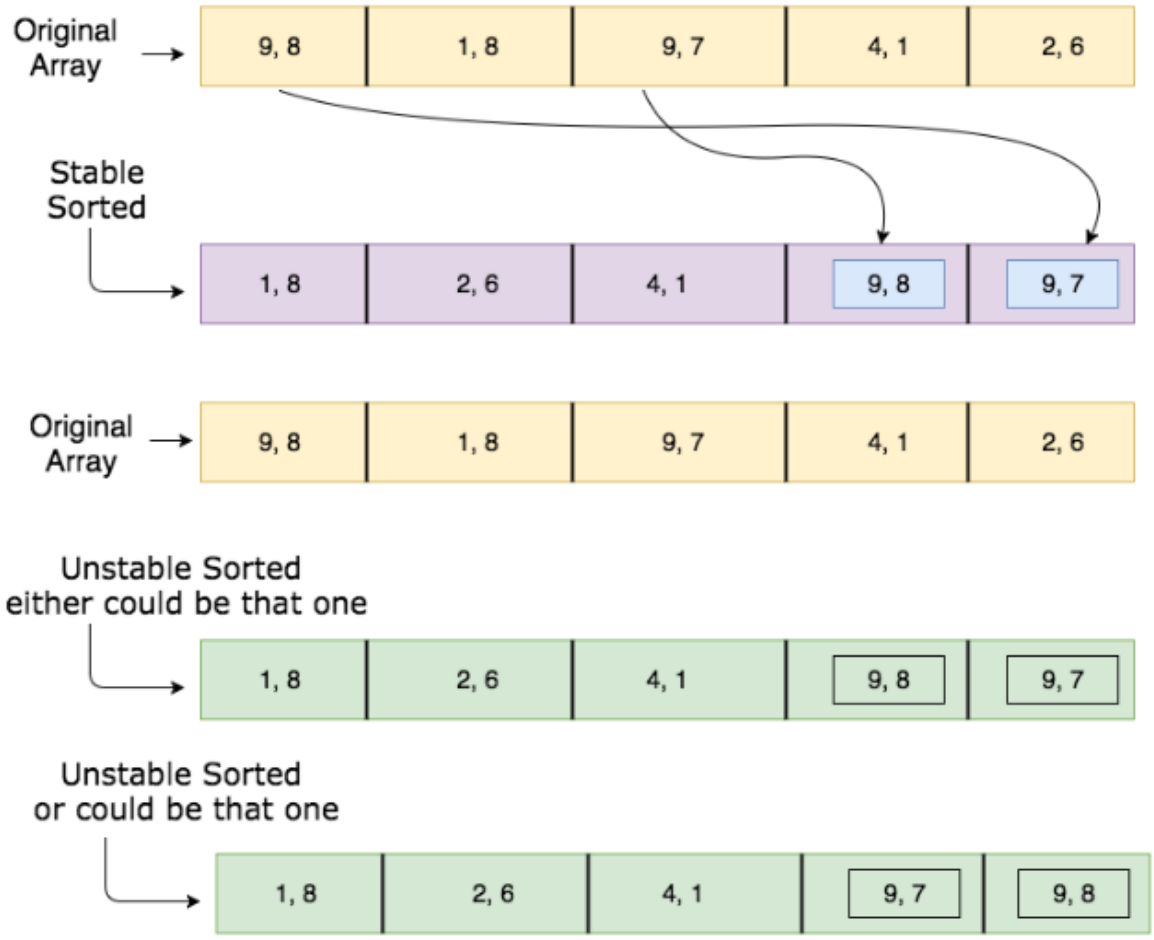


Image 5: Stable and unstable sorting. Taken from: https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important on 11/05/2020

1.5. Comparison-based and non-comparison-based sorting algorithms

A comparison sort is a type of sorting algorithm that 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.

A comparison sort algorithm sorts items by comparing values between each other. It can be applied to any sorting cases. The best complexity is O(n*log(n)).

A non-comparison sort algorithm uses the internal character of the values to be sorted. It can only be applied to some particular cases, and requires particular values. The best complexity is probably better depending on cases, such as O(n).[11]

All sorting problem that can be sorted with non-comparison sort algorithm can be sorted with comparison sort algorithm, but not the other way around.

One of the most critical differences between these two sorting algorithms is speed. Non-comparison sorting is usually faster than comparison sorting because of not doing the comparison. The limit of speed for comparison-based sorting algorithm is O(NlogN) while for non-comparison based algorithms its O(n) i.e. linear time.[12]

Examples of comparison-based sorting: Quick Sort, Merge Sort, Heap Sort, Selection Sort, Bubble Sort, Insertion Sort, Block Sort, Shell Sort, Heap Sort,

Examples of non-comparison-based sorting is: Radix Sort, Counting Sort, Bucket Sort, Postman Sort, Flash Sort, Burst Sort.

2. Sorting Algorithms

2.1. Bubble Sort

Bubble Sort, one of the simplest algorithms for sorting an array, consists of repeatedly exchanging pairs of adjacent array elements that are out of order until no such pair remains. The serial software implementation of bubble sort has a time complexity that is:

Worst and Average Case Time Complexity: O(n*2). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

This type of sort is a slow-and-predictable sorting algorithm. Is often used to introduce the concept of a sorting algorithm.[13]

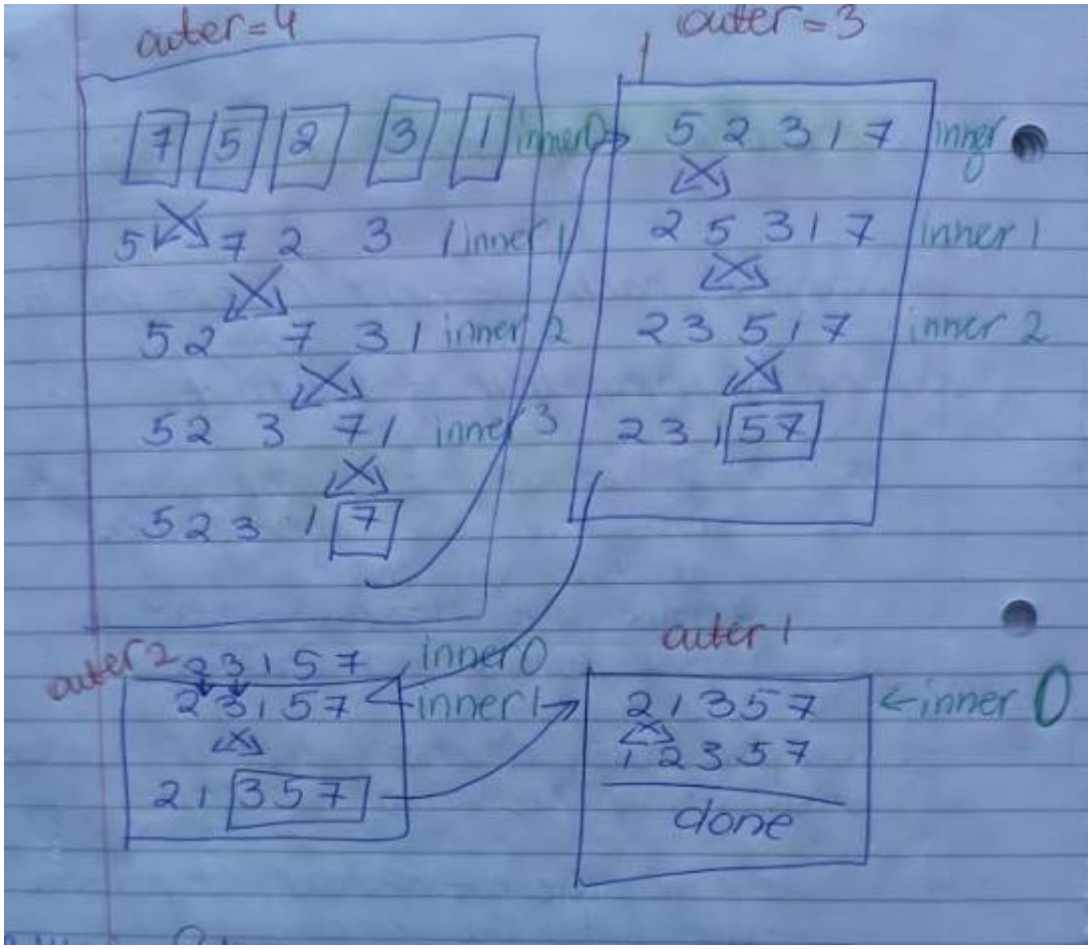


Image 6: Bubble Sort. Taken from my notes, module Computational Thinking with Algorithms (week 10), GMIT.

An implementation of Bubble sort is shown below:

```
# Defining bubble sort function
# Taken from: https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm on 04/05/2020
def bubbleSort(alist):

    for i in range(len(alist)-1,0,-1): # outer for loop to swap the elements in correct order
        for j in range(i):

            # compare the item on the left with the item on the right and if it's larger then swap places
            if alist[j] > alist[j+1]:
                temp = alist[j]
                alist[j] = alist[j+1]
                alist[j+1] = temp


# Create a list to use a Bubble sort
list = [98,65,29,12,6,102,587,33,46.59,72,84]

# Call the function
bubbleSort(list)

# print sorted list
print("Sorted list is:")
print(list)
```

```
Sorted list is:
[6, 12, 29, 33, 46.59, 65, 72, 84, 98, 102, 587]
```

Image 7: An Implementation of Bubble Sort. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

2.2. Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.[13]

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: O(1)

It never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Stability: The default implementation is **not stable**.

In Place : Yes, it does not require extra space.

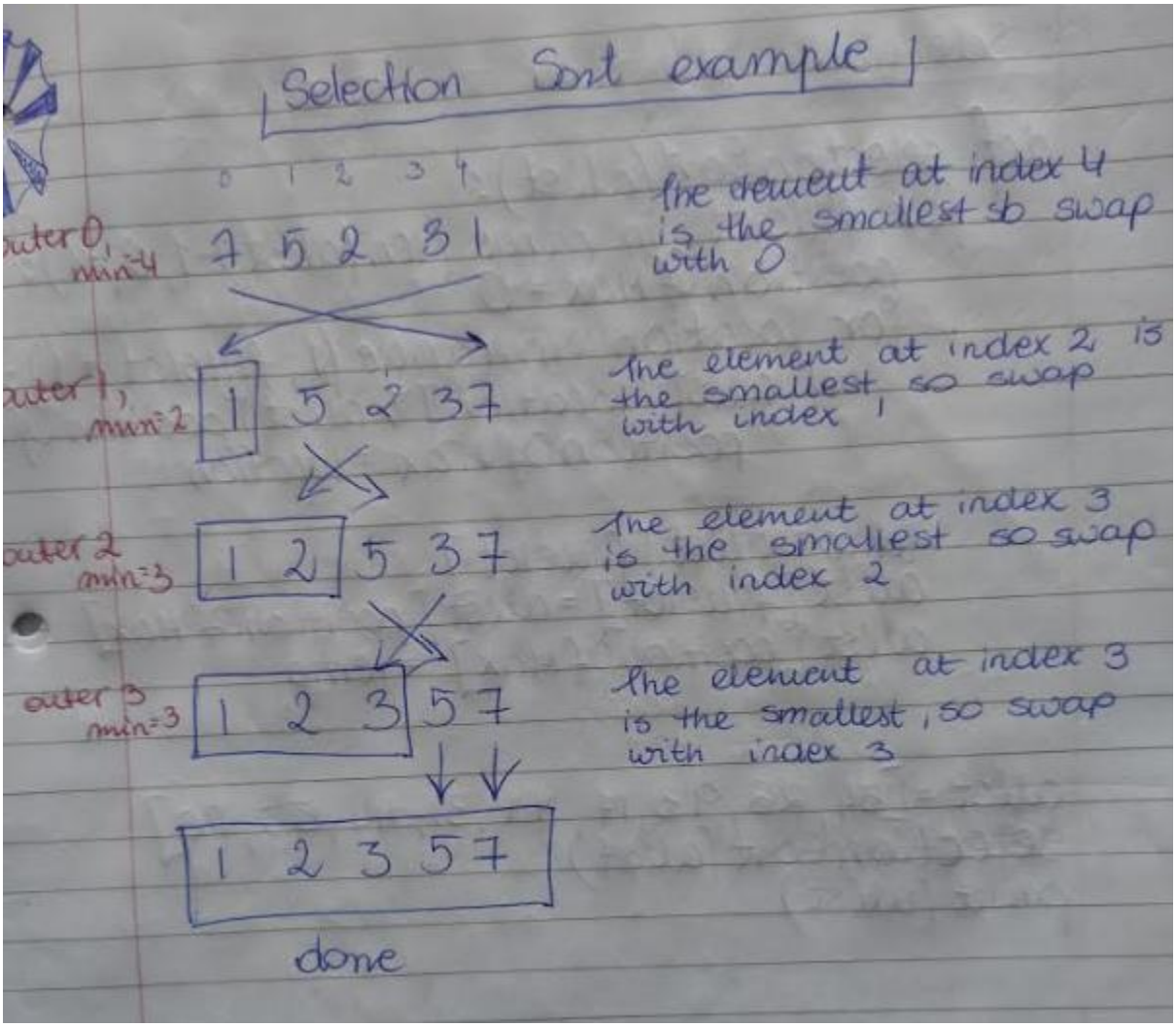


Image 8: Selection Sort. Taken from my notes, module Computational Thinking with Algorithms (week 10), GMIT.

An implementation of Selection sort is shown below:

```
# Defining a Selection Sort
# Taken from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheSelectionSort.html on 03/05/2020
def selectionSort(alist):
    # for loop will repeat until all elements are executed. It repeats from the last element to the first.
    for fillslot in range(len(alist)-1,0,-1):
        # maximum position is set as 0
        positionOfMax = 0
        # the inner for loop is used to find the maximum value in the unsorted subarray
        for location in range(1,fillslot+1):
            # Compare the current element with the next one
            if alist[location] > alist[positionOfMax]:
                positionOfMax= location

            # and swap the compared element with the maximum value
            temp = alist[fillslot]
            alist[fillslot] = alist[positionOfMax]
            alist[positionOfMax] = temp


# Create a list to use a Selection sort
alist = [54,26,93,17,77,31,44,55,20,159,458,41,789,364,9874]

# call the function
selectionSort(alist)

#print sorted list
print("Sorted list is:")
print(alist)
```

```
Sorted list is:
[17, 20, 26, 31, 41, 44, 54, 55, 77, 93, 159, 364, 458, 789, 9874]
```

Image 9: An Implementation of Selection Sort. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

2.3. Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time when elements are already sorted. Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.[13]

It involves finding the right place for a given element in a list. At the beginning the function compares the first two elements and sorts them by comparing them. Then the third element needs to find its proper position among the previous two sorted elements. This way more and more elements are added to the already sorted list by putting them in their proper position.[16]

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

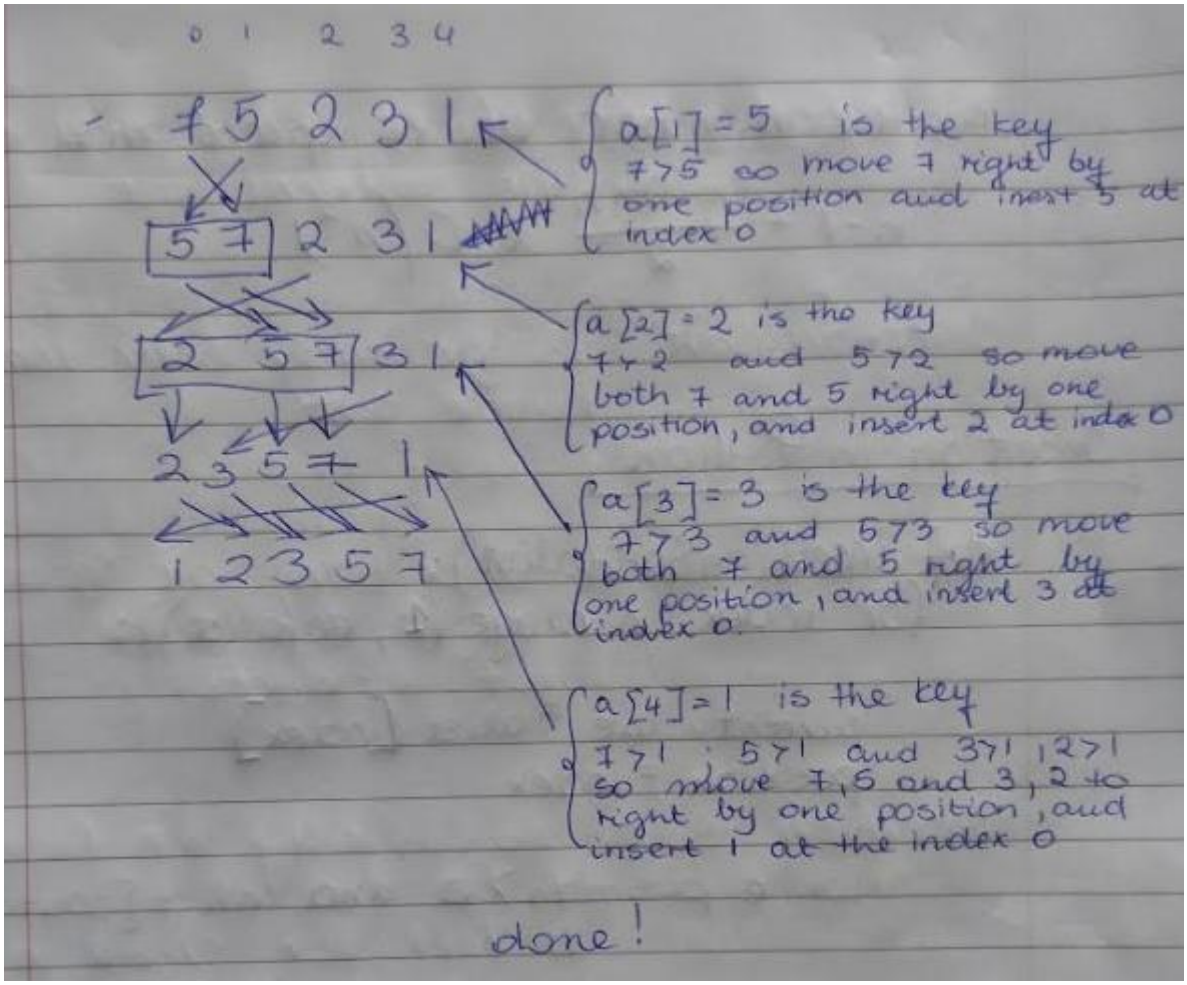


Image 10: Insertion Sort. Taken from my notes, module Computational Thinking with Algorithms (week 10), GMIT.

An implementation of Insertion sort is shown below:

```
# Defining insertion sort function
# Taken from: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html on 02/05/2020
def insertionSort(alist):
    for i in range (1, len(alist)):
        currentValue = alist[i] # create kind of temporary variables
        position = i  # position which I am checking is equal to i index of the character

        while (position > 0) and (alist[position - 1] > currentValue):
                            # while loop will check all numbers until they get
                            # the right place and every time I'll do this
                            # I want to reduce position by 1.
            alist[position] = alist [position - 1]
            position -= 1

            alist[position] = currentValue

# Create a list to use an Insertion sort
alist = [12,15,25,31,21,5,82,76,30,29,33,17,105]

# Call the function
insertionSort(alist)

# print sorted list
print("Sorted list is:")
print(alist)
```

```
Sorted list is:
[5, 12, 15, 17, 21, 25, 29, 30, 31, 33, 76, 82, 105]
```

Image 11: An Implementation of Insertion Sort. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

2.4. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.[13]

It repeatedly breaks the array down into smaller chunks for an input array. After complete deconstruction arrays are recombined but not in their original order. The key to the merge sort algorithm is that when we begin the reconstruction we do not just put the elements back in their original order, instead we insert the elements in their appropriate sorted order. Merge is faster when it has to merge two sorted halves of an array than to sort the full array in place.

In the original input array in implementation will use recursion.

Time Complexity: Sorting arrays on different machines. Time complexity can be expressed as following recurrence relation. T(n) = 2T(n/2) + $\Theta(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(nLogn)$.

Time complexity of Merge Sort is $\Theta(nLogn)$. in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Every element in the input approximately log in times giving n times log n.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer (recursive)

Sorting In Place: No in a typical implementation

Stable: Yes

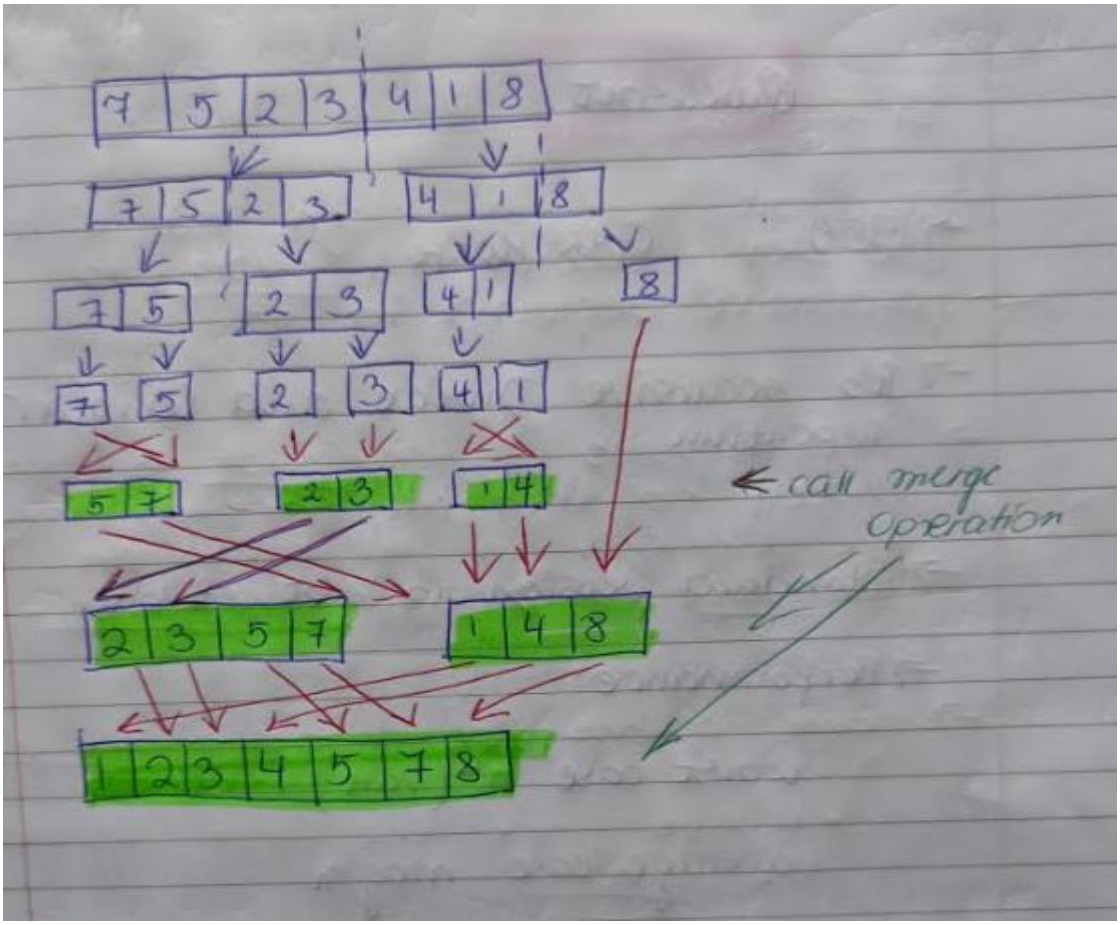


Image 12: Merge Sort. Taken from my notes, module Computational Thinking with Algorithms (week 11), GMIT.

An implementation of Merge sort is shown below:

```
# Defining a Merge sort function
# Taken from: https://www.youtube.com/watch?v=3aTfQvs-_hA on 30/04/2020
def mergeSort(a,b):
    c=[]  # That will be final and sorted array
    a_idx,b_idx = 0,0
    while a_idx < len(a) and b_idx < len(b):
                # while loop will repeat until all elements are used. On each repeat
                # all elements are compared and appended whichever is smaller onto a new merged arrays.
        if a[a_idx] < b[b_idx]:
            c.append(a[a_idx])
            a_idx += 1
        else:
            c.append(b[b_idx])
            b_idx += 1
    if a_idx == len(a): c.extend(b[b_idx:])
    else:               c.extend(a[a_idx:])
    return c
              # At the end of the while loop I extended the merged list with two input arrays

# If I want to use a Merge sort function I need to create two lists: a and b
a= [12,16,20,25,69]
b= [11,26,33,45,70]

# Call and print the function
print("Sorted list is:")
print(mergeSort(a,b))
```

```
Sorted list is:
[11, 12, 16, 20, 25, 26, 33, 45, 69, 70]
```

Image 13: An Implementation of Merge Sort. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

2.5. Counting Sort

Counting Sort runs in O(n) time, amking it asymptotically faster than comparison-based sorting algorithms. It only works when the range of potential items in the input is known ahead of time.[14] Counting sort works by iterating through the input, counting the number of times each item occurs, and using those counts to compute an item's index in the final, sorted array.

Counting sort allows to do something which seems impossible – sort a collection of items in linear time.

Worst case time: n + k

Best case time: n + k)

Average case time: n + k

Space: n + k

Stable: Yes

Image 14: Counting Sort. Taken from my notes, module Computational Thinking with Algorithms (week 11), GMIT.

An implementation of Counting sort is shown below:

```
# Defining a Counting sort function
# Taken from: https://github.com/Thalmann/counting_sort/blob/master/counting_sort.py on 04/05/2020
def countingSort(alist):

    k = max(alist) + 1
    n = len(alist)

    # create a count array to count the number of instances
    count = [0] * k

    # the for loop will execute all elements in alist and will count occurences of each number in the array
    for x in alist:
        count[x] += 1

    total = 0
    for x in range(k):
        old = count[x]
        count[x] = total
        total += old

    # create nlist
    nlist = [0] * n
    for x in alist:
        nlist[count[x]] = x
        count[x] += 1

    return nlist


# Create alist to use counting sort
alist = [15,6,159,36,25,93,33,18,49,75,258,763,75,156,8743]

# call and print the function
print("Sorted Array is:")
print(countingSort(alist))
```

```
Sorted Array is:
[6, 15, 18, 25, 33, 36, 49, 75, 75, 93, 156, 159, 258, 763, 8743]
```

Image 15: An Implementation of Counting Sort. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

3. Implementation and Benchmarking

This project contains a Python codes which will be used to benchmark five different sorting algorithms. I have implemented and benchmarked five sorting algorithms with random array. I have chosen the fallowing algorithms:

1. A simple comparison-based sort:
    - Bubble Sort
    - Selection Sort (my choice)
    - Insertion Sort(my choice)
2. An efficient comparison-based sort:
    - Merge Sort
3. A non-comparison sort:
    - Counting Sort

3.1. Python code

The implementation of each sorting algorithms are shown above (in Sorting Algorithms section). The main idea of benchmarking is to figure out how fast the code executes and where the bottlenecks are. These actions lead to optimization. There are situations where you need your code to run faster because your business needs have changed, and you need to figure out what parts of your code are slowing it down.[15] Benchmarking involves running sorting algorithms for random numbers with different input sizes and measuring the time needed to run.

The running time was measured ten times and I have used numpy.mean to return the average of an array of ten run times for each algorithm and input size. I have used a variety of different input sizes, e.g. size = 100,250,500,750,1000,1250,2500,3750,5000,6250,7500,8750 and 10000. I used different sizes to test the effect of the input size on the running time of each algorithm.

At the beginning of Benchmarking sorting algorithms (coding) I imported all libraries important for the project.

```
import time
import random
from random import randint
import numpy as np
import pandas as pd
```

- time is used for counting each algorithm
- random is used to generate random numbers(arrays)
- numpy to generate lists of random arrays
- pandas to create DataFrame for final results.

3.1.1. Random arrays

I used *random_array* to return an array of size n. The random module is imported with the randint function to generate random integers in the range, here between 0 and 99.

Those integers were appended to an array. Those arrays were used to generate integers with different input sizes “size” (in my project). I also used them to run time for each algorithm in milliseconds.

```
def random_array(n):  # Taken from: https://docs.python.org/3/library/random.html
    array = []        # Create an array called array
    for i in range(0, n, 1): # low, high, size
        array.append(random.randint(0, 100))  # randomly generated integers
                                              # between 0 and 100 and append that numbers to array
    return array
```

Image 16: Random array.. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

I also created *create_array* to generate randomized arrays to implement and benchmark each algorithm in seconds. It is more to save time when it is executed and see how it all work.

```python
def create_array(length=10, maxint=100): # Length parameter gives me the length of the new array
                                         # and maxint parameter to mark the upper bound
    new_arr = [randint(0,maxint) for _ in range(length)]
    return new_arr
```

Image 17: An Create array. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

This helped me more to understand benchmarking. Results are below.

```python
print("  Benchmark the runtime performance of Bubble Sort, Insertion Sort, Selection Sort,")
print("  Merge Sort and Counting Sort - output in seconds")
print("              ")
print("''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''")
# Input sizes to test the effect of the input sizes on the running time of each algorithm.
# Taken from: https://www.youtube.com/watch?v=vuOpxXeFjg8&t=327s on 03/05/2020
def benchmark(size = [100,250,500,750,1000,1250,2500,3750,5000,6250,7500,8750,10000]):
    # time function will be used to time the sorting algorithms
    from time import time

    b1 = []  # Bubble Sort times
    b2 = []  # Insertion Sort times
    b3 = []  # Selection Sort times
    b4 = []  # Merge Sort times
    b5 = []  # Counting Sort times
    for length in size:
        a = create_array(length,length)              # t0 - start time
                                                     # t1 - end time
```

```python
        # Bubble time sort
        t0 = time()
        s = bubble_sort(a)
        t1 = time()
        b1.append(t1-t0)  # record bubble time

        # Insertion time sort
        t0 = time()
        s = insertion_sort(a)
        t1 = time()
        b2.append(t1-t0)  # record insertion time

        # Selction time sort
        t0 = time()
        s = selection_sort(a)
        t1 = time()
        b3.append(t1-t0)  # record selection time

        # Merge time sort
        t0 = time()
        s = merge_sort(a)
        t1 = time()
        b4.append(t1-t0)  # record merge time

        # Counting time sort
        t0 = time()
        s = counting_sort(a)  # sort with counting sort
        t1 = time()
        b5.append(t1-t0)  # record counting time


    print("size \tBubble Sort\tInsertion Sort\tSelection Sort\tMerge Sort\tCounting Sort")
    print("_____________________________________________________________________________________________________")
    for i, cur_n in enumerate(size):
        print("%d\t%0.5f \t%0.5f \t%0.5f \t%0.5f \t%0.5f"%(cur_n,b1[i],b2[i],b3[i],b4[i],b5[i]))
benchmark()
print("               ")
print("_______________________________________________________________________________________________________")
print("               ")
```

Image 18: The benchmark performance in seconds. Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

Benchmark the runtime performance of Bubble Sort, Insertion Sort, Selection Sort, Merge Sort and Counting Sort - output in seconds

size	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Counting Sort
100	0.00400	0.00000	0.00400	0.00200	0.00000
250	0.02300	0.00000	0.01300	0.00200	0.00000
500	0.06000	0.00000	0.03000	0.00300	0.00000
750	0.12701	0.00000	0.06700	0.00300	0.00000
1000	0.19601	0.00000	0.11301	0.00500	0.00000
1250	0.30702	0.00000	0.18301	0.00500	0.00100
2500	1.24507	0.00000	0.76404	0.01300	0.00100
3750	3.03217	0.00100	1.74510	0.02000	0.00200
5000	5.24130	0.00100	3.10818	0.02800	0.00200
6250	8.13747	0.00100	4.68627	0.03600	0.00300
7500	11.72467	0.00200	6.72638	0.04100	0.00400
8750	15.91091	0.00300	9.34353	0.04800	0.00400
10000	20.91420	0.00300	12.13269	0.05700	0.00400

Image 19 : The results of benchmarking in seconds, Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

3.1.2. Timing the algorithm and benchmark

To measure the running time of all sorting algorithms I used time module to record the start time (t0) and the end time (t1). This gives the results in seconds.

Benchmark_a - used all sorting algorithms and for each input size run them 10 times and measured the time taken for it to sort randomly generated arrays.

It was used in for loop to execute an array 10 times for each algorithm.

I called time.time ()twice, before sort algorithms were started to measure the start time and when they ended to measure the end time.. Arrays generated from *random_array* were appended into lists which were created to store results.

3.1.3. Average time

avg_time_bubble_sort – (here just for bubble sort but it is done for all algorithms) – returns average running time of the lists which store results from timing. It tastes input sizes and sorting algorithms as inputs. I used numpy.mean to return the average figures.

To get results in milliseconds I multiplied by 1000.

Those results were appended again into different lists which I created before for each sorting algorithm.

3.1.4. DataFrame

The results are recorded and stored in a data table. The data set is organized into columns and rows. Rows correspond to sorting algorithms and columns correspond to the size of the array. The data set type is Pandas DataFrame and is allocated to a variable named df_average.

The output is formatted into 3 decimal places and transposed. I used .round(3).transpose to get it.

Names given to the table:

1. Size = size of the array
2. Bubble Sort
3. Insertion Sort
4. Selection Sort
5. Merge Sort
6. And Counting Sort

```
import pandas as pd

# To display data (results) I need dataFrame. It is two-dimensional, size-mutable, potentially
# heterogeneous tabular data. Pandas is a fast, powerful, flexible and easy to use Python open source
# data analysis.
df_average = pd.DataFrame()

# All outputs => times of each algorithm will be added to dataFrame
df_average['Size'] = size

# Average time added and names given to table
df_average['Bubble Sort'] = avg_bubble
df_average['Insertion Sort'] = avg_insertion
df_average['Selection Sort'] = avg_selection
df_average['Merge Sort'] = avg_merge
df_average['Counting Sort'] = avg_counting

# Set the DataFrame index
# Taken from: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html on 07/05/2020
df_average.set_index('Size', inplace=True)

# Output is formated to 3 decimal places and is transposed
# Taken from: https://www.geeksforgeeks.org/python-pandas-dataframe-round/
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.round.html
# https://datatofish.com/list-to-dataframe/ and
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transpose.html
# https://www.geeksforgeeks.org/python-pandas-dataframe-transpose/ all on 07/05/2020
df_average.round(decimals=3).transpose()
```

Image 19: DataFrame Implemantation,
Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

3.1.5. Plot Graph

At the very end I generated a plot to show all results in graph. To do this I imported matplotlib.pyplot to create a line plot of the results. I added title, xlabel with fontsize = 13, xticks, ylabel with fontsize =13 and yticks. I called grid which exposes the graph as very readable and also called gca().legend() to show which line belongs to which sort.

3.1.6. Final results

As I mentioned before I decided to choose 5 sorting algorithms, which are:

- Bubble Sort,
- Insertion Sort,
- Selection Sort,
- Merge Sort,
- And Counting Sort.

Below are shown results of my research. This contains average times of ten runs of each sorting algorithm. It is measured for each input size (size= 100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750 and 1000) and results are in milliseconds.

Out[5]:

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	1.5	11.101	43.402	97.506	189.211	297.917	1241.971	2733.256	4945.983	8054.661	11471.156	16174.225	20662.382
Insertion Sort	0.6	3.900	15.601	35.602	68.704	110.206	461.126	1037.459	1836.305	2918.167	4348.449	5826.833	7745.043
Selection Sort	1.1	7.000	29.602	62.603	120.807	185.711	723.741	1603.992	3045.774	4679.668	6879.993	9429.139	12110.493
Merge Sort	0.4	1.200	2.300	3.700	5.000	6.900	14.601	23.401	40.902	41.802	54.903	65.904	79.404
Counting Sort	0.0	0.300	0.300	0.300	0.600	0.400	0.700	1.200	1.400	1.900	2.300	2.800	3.200

Image 20 : The results of benchmarking in milliseconds, Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

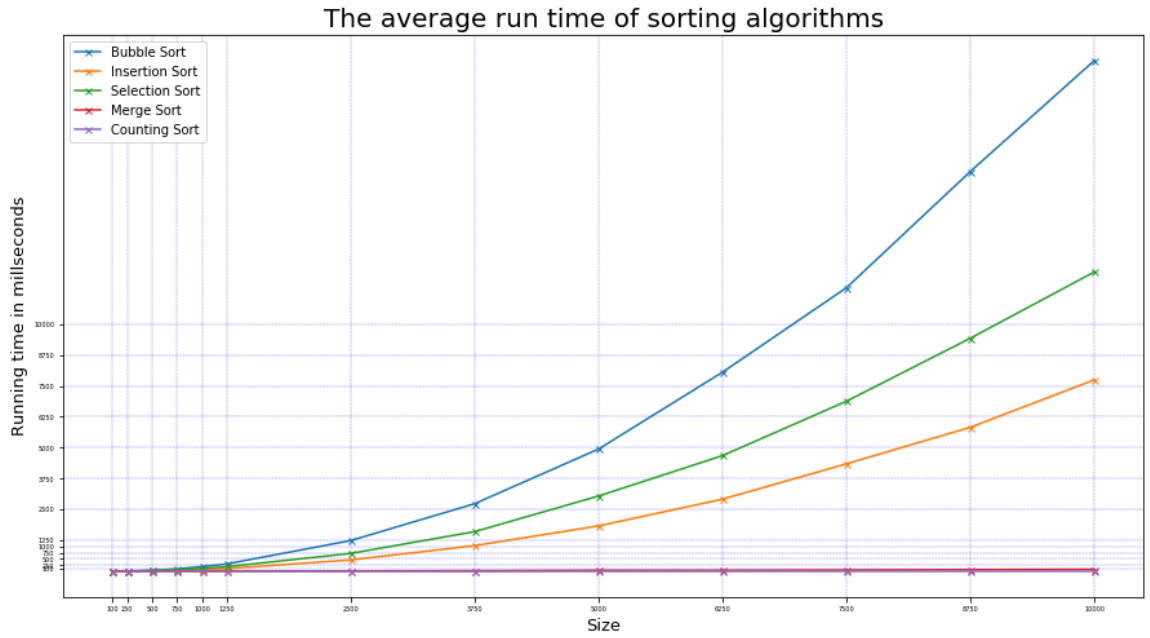


Image 21 : Plot graph – the average running time for each of five sorting algorithms for input sizes, Taken from my Jupyter notebook – CTA – Project, https://github.com/karolinaszafranbelzowska/CTA-project

Looking at the results I can say that Bubble Sort had the longest running time of all algorithms. For the very large input sizes it needs almost twice more time than Selection Sort. Bubble Sort is suitable for small input sizes and in this case it needed much more time than others, and I was

expecting this. It is extremely slow and probably because of this algorithm it took around 40 minutes to execute outputs.

Selection Sort had the second longest time in my research. It usually gives better performance than Bubble Sort and so it did in here. Like Bubble it is suitable for small data, input sizes.

Insertion Sort as the algorithm is a simple comparison-based algorithm and it is suitable for small lists which are close to be sorted. Very inefficient for large random arrays and like Bubble and Selection Sorts needs more time to run input instances.

Those three sort algorithms I would not recommend to use in benchmarking as it needs a lot of time to execute large numbers. Bubble, Selection and Insertion Sorts suffer from exponential time complexity in the worst cases, these algorithms begin to slow down substantially as the input array size increases.

Merge had the second best time performance. The best, worst and average cases are very similar (n logn). Merge sort implementation shows that is significantly faster for larger input array sizes than Bubble, Selection and Insertion Sort. For the smaller sizes I see a much closer competition. Average time in Merge Sort is very close to average time in Insertion Sort at an input array size of 100. The larger the size is the greater the difference between each algorithm is. It is more efficient for large input instances and I could see that in my study.

Counting Sort had the best average time of all algorithms and I was expecting this. Counting is suitable for large input instances and it allows to execute something which seems impossible.

4. References

[1] https://whatis.techtarget.com/definition/sorting-algorithm

[2] https://www.studytonight.com/data-structures/introduction-to-sorting

[3] https://en.wikipedia.org/wiki/Sorting_algorithm

[4] https://en.wikipedia.org/wiki/Best,_worst_and_average_case

[5] https://www.geeksforgeeks.org/g-fact-86/

[6] https://en.wikipedia.org/wiki/Big_O_notation

[7] https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

[8] https://medium.com/better-programming/a-gentle-explanation-of-logarithmic-time-complexity-79842728a702

[9] http://mca.kiet.edu/webguru/unit-1ds.html

[10] https://www.baeldung.com/cs/stable-sorting-algorithms

[11] https://stackoverflow.com/questions/25788781/definition-of-non-comparison-sort

[12] https://en.wikipedia.org/wiki/Comparison_sort

[13] https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

[14] https:/www.interviewcake.com/concept/java/counting-sort

[15] https://www.blog.pythonlibrary.org/2016/05/24/python-101-an-intro-to-benchmarking-your-code/

[16] https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm

Other Web sides:

[17] https://numpy.org

[18] https://www.w3schools.com

[19] https://pandas.pydata.org

[20] https://www.geeksforgeeks.org

[21] https://datatofish.com

[22] https://matplotlib.org

[23] https://docs.python.org

[24] https://stackoverflow.com

[25]
https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html

https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheSelectionSort.html

https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html

https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html

Videos

[1] https://www.youtube.com/watch?v=AthG28-_RuM&t=445s

[2] https://www.youtube.com/watch?v=AgtzMtrzhzs

[3] https://www.youtube.com/watch?v=JxTghISBmI8

[4] https://www.youtube.com/watch?v=3aTfQvs-_hA

Github Repositories

https://github.com/jennifer-ryan/benchmarking-sorting-algorithms

https://github.com/andkoc001/CTA_sorting_project

https://github.com/johndunne2019/CTA-Project-Benchmarking-Sorting-Algorithms

https://gist.github.com/haandol/a5df913cfd278820e43e

https://github.com/Thalmann/counting_sort/blob/master/counting_sort.py