Benchmarking Sorting Algorithms - Project 2020

May 2, 2020

GMIT Galway -Computational Thinking with Algorithms
Slawomir Sowa

1 Introduction:

Sorting is any process of ordering (arranging items in a sequence) or categorizing (grouping items) items. Sorting is one of the most important algorithms in computer science. We can use sorting to solve a wide range of problems:

- **Searching:** Searching for an item on a list works much faster if the list is sorted.
- **Selection:** Selecting items from a list based on their relationship to the rest of the items is easier with sorted data.
- **Duplicates:** Finding duplicate values on a list can be done very quickly when the list is sorted.
- **Distribution:** Analyzing the frequency distribution of items on a list is very fast if the list is sorted. [3]

Sorting is often an important step as part of other computer algorithms e.g. in computer graphic (CG) objects. Few real world examples Transactions in a bank account statement, results from a web search. [6]

We can define algorithm as a finite sequence of well-defined, computer-implementable instructions, to solve a problem or perform a computation [1]

In computer science sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order. Most commonly in numerical or a form of alphabetical order. Sorting algorithms can reduce the complexity of a problem. [2]

Comparison Sort Comparison sort is a type of sorting algorithm which uses comparison operators to determine which of two elements should appear first in sorted list. This type of algorithms are most widely applicable to diverse types of input data. The performance of comparison based algorithms in average or worst cases cannot be better than $O(n \log n)$. [6]

In Place sorting An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed[10]

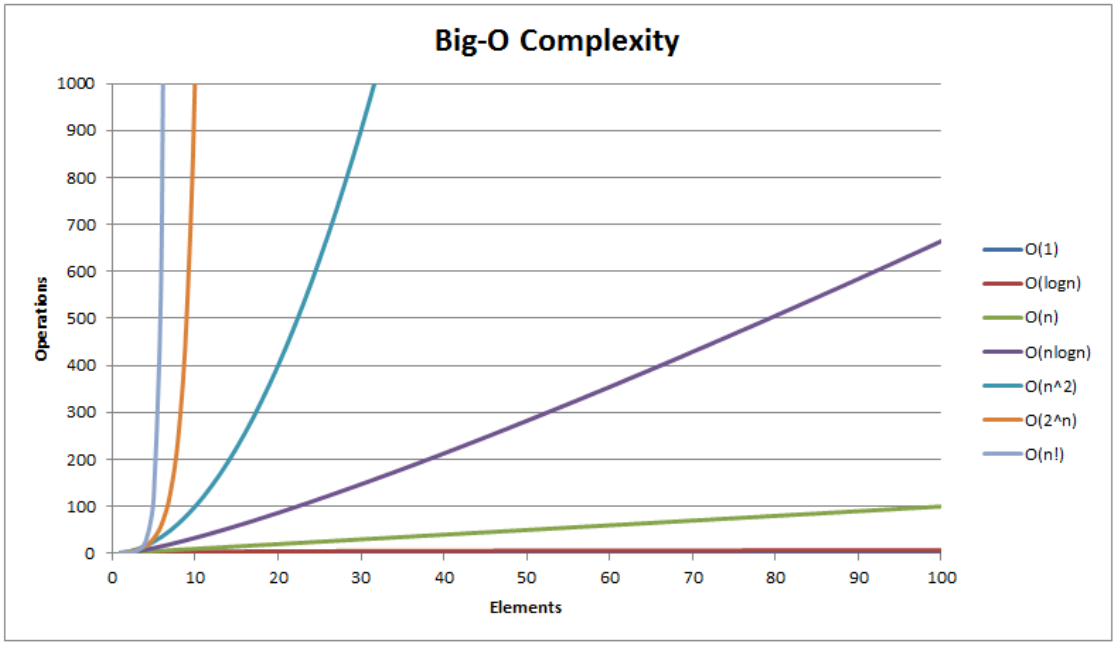
Not-in Place sorting Algorithms that require additional working memory and it depend on input size.

Time and Space Complexity Time complexity of an algorithm signifies the total time required by the program to run till its completion and is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity[11]

The space complexity of an algorithm is the amount of memory space required to solve a computational problem as a function of the size of the input. It is the memory required by an algorithm to execute a program and produce output.

Similar to time complexity, Space complexity is often expressed asymptotically in big O notation, such as $O(n)$, $O(n^2)$, $O(n \log n)$ etc., where n is the input size in units of bits needed to represent the input.[12]

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time COmplexity	Worst Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Counting Sort	$O(n)$	$O(n)$	$O(n)$	$O(n)$



Stability A sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.[13]

Unstable, meaning that it does not maintain the relative order of elements with equal values. This isn't an issue with primitive types (like integers and characters…) but it can be a problem when we sort complex types, like objects.

2 Sorting Algorithms

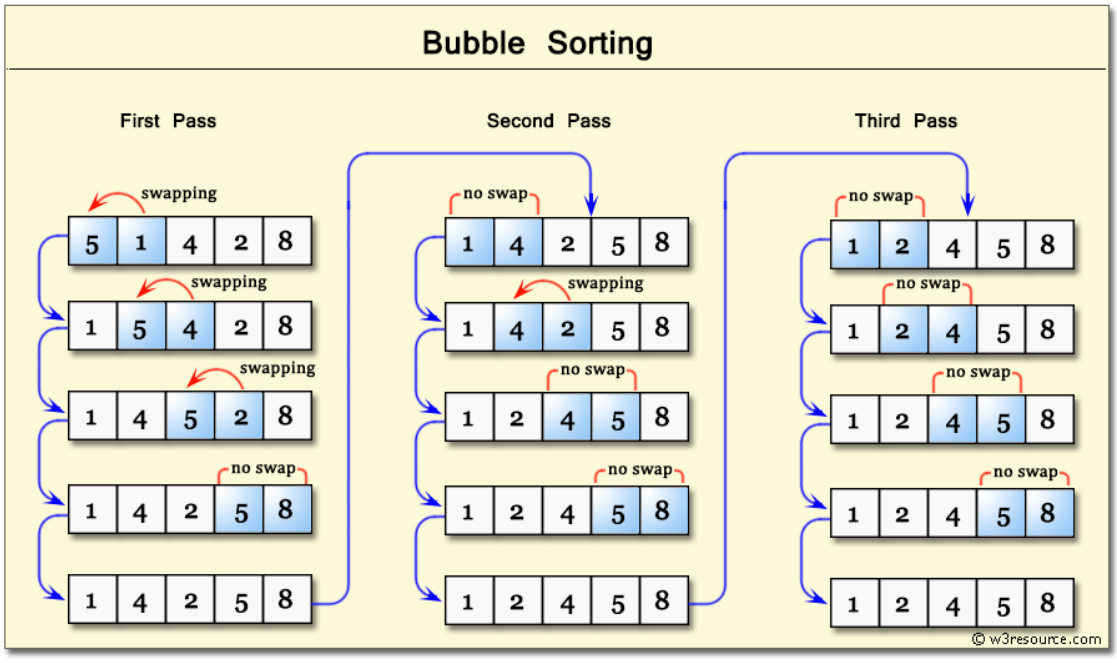
I selected five sorting algorithms for testing.

1. A simple comparison-based sort: **Bubble Sort** and **Selection Sort**
2. As efficient comparison-based sort: **Merge Sort** and **Heap Sort**
3. A non-comparison sort: **Counting Sort**

Now I will present the algorithms in turn to explain how each algorithm works.

2.1 Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. First analysed as early as 1956. [4], [5] Bubble sort algorithm is called comparison-based and the only way to gain information about the total order is by comparing a pair of elements. No algorithms that sorts by comparing elements can do better than $O(n \log n)$ performance in the average or worst cases. [6]



https://www.w3resource.com/javascript-exercises/javascript-function-exercise-24.php

Bubble sort is in-place sorting algorithm. In general is simple to implement but it is slow and impractical for most problems.

[1]:
```
# Code taken from https://www.geeksforgeeks.org/bubble-sort/
# An example of Bubble sort algorithm

def bubbleSort(arr):
    # length of the array assigned to variable
    n = len(arr)
    # Loop over all array elements
    for i in range(n):
    # After each iteration the largest element is placed at the end
        for j in range(0, n-i-1):
            # Swap if the element arr[j] is greater than arr[j+1]
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    # return sorted array
    return arr

alist = [65,125,48,469,5,333,2547,645,2,1,]
bubbleSort(alist)
print(alist)
```

```
[1, 2, 5, 48, 65, 125, 333, 469, 645, 2547]
```

Performance

- Best Case: $O(n)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$
- Space Complexity: 1
- Stable: Yes

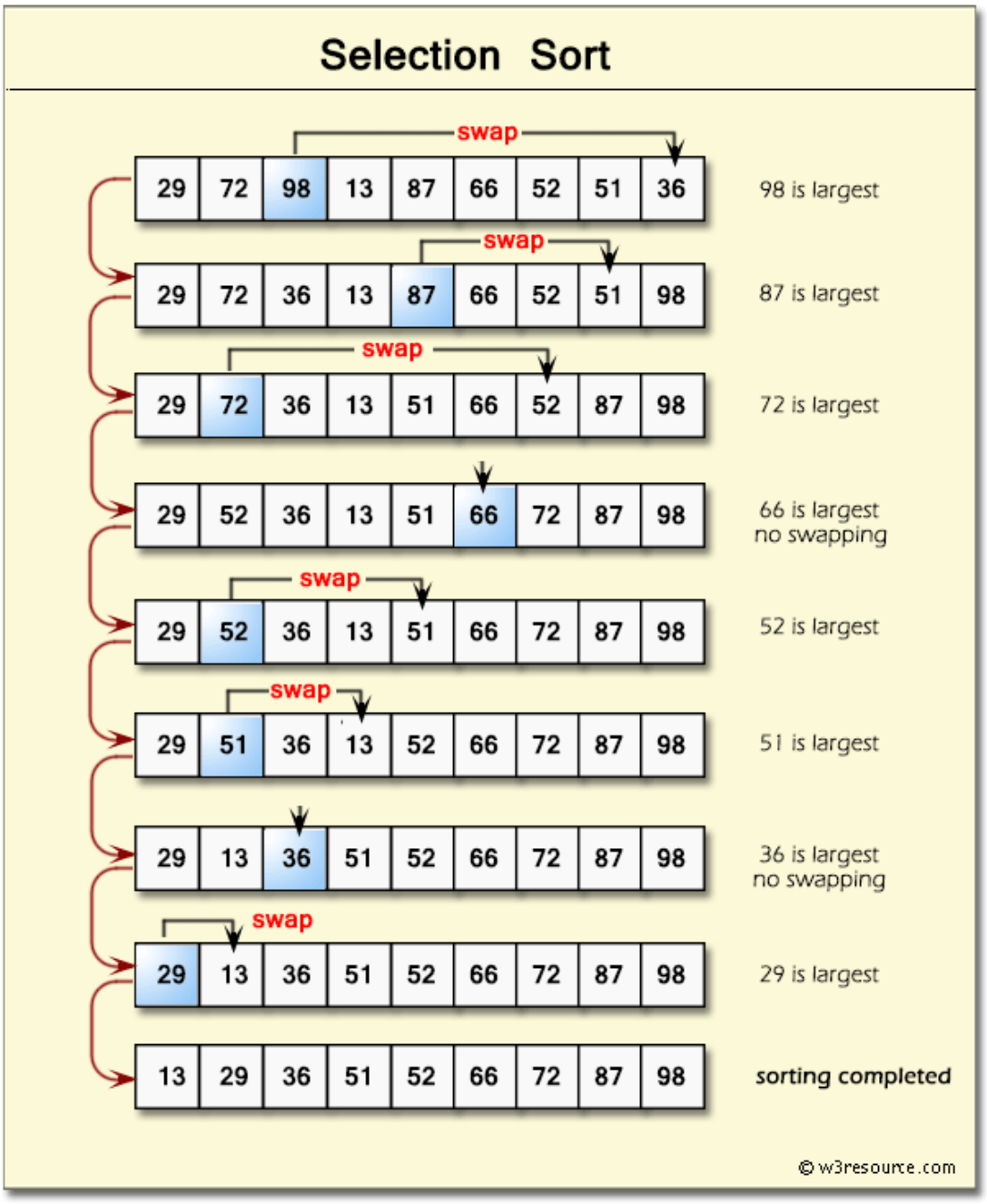
2.2 Selection Sort

Similar to Bubble sort, Selection sort is comparison based sorting algorithm, gives better performance than bubble sort, but still impractical for real world tasks with a significant input size. [6]

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. The algorithm maintains two sub arrays in a given array.

1) The sub array which is already sorted.
2) Remaining sub array which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted sub array is picked and moved to the sorted sub array. [7]



https://www.w3resource.com/c-programming-exercises/searching-and-sorting/c-search-and-sorting-exercise-2.php

[2]:
```python
# Code taken from https://stackabuse.com/sorting-algorithms-in-python/

# An example of Selection sort algorithm

def selectionSort(arr):
    for i in range(len(arr)):
        # We assume that the first item of the unsorted segment is the smallest
        lowest_value_index = i
        # This loop iterates over the unsorted items
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[lowest_value_index]:
                lowest_value_index = j
        # Swap values of the lowest unsorted element with the first unsorted
        # element
        arr[i], arr[lowest_value_index] = arr[lowest_value_index], arr[i]
    # return sorted array
    return arr

alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Algorithm has an $O(n^2)$ time complexity, which makes it inefficient on large lists,

Performance

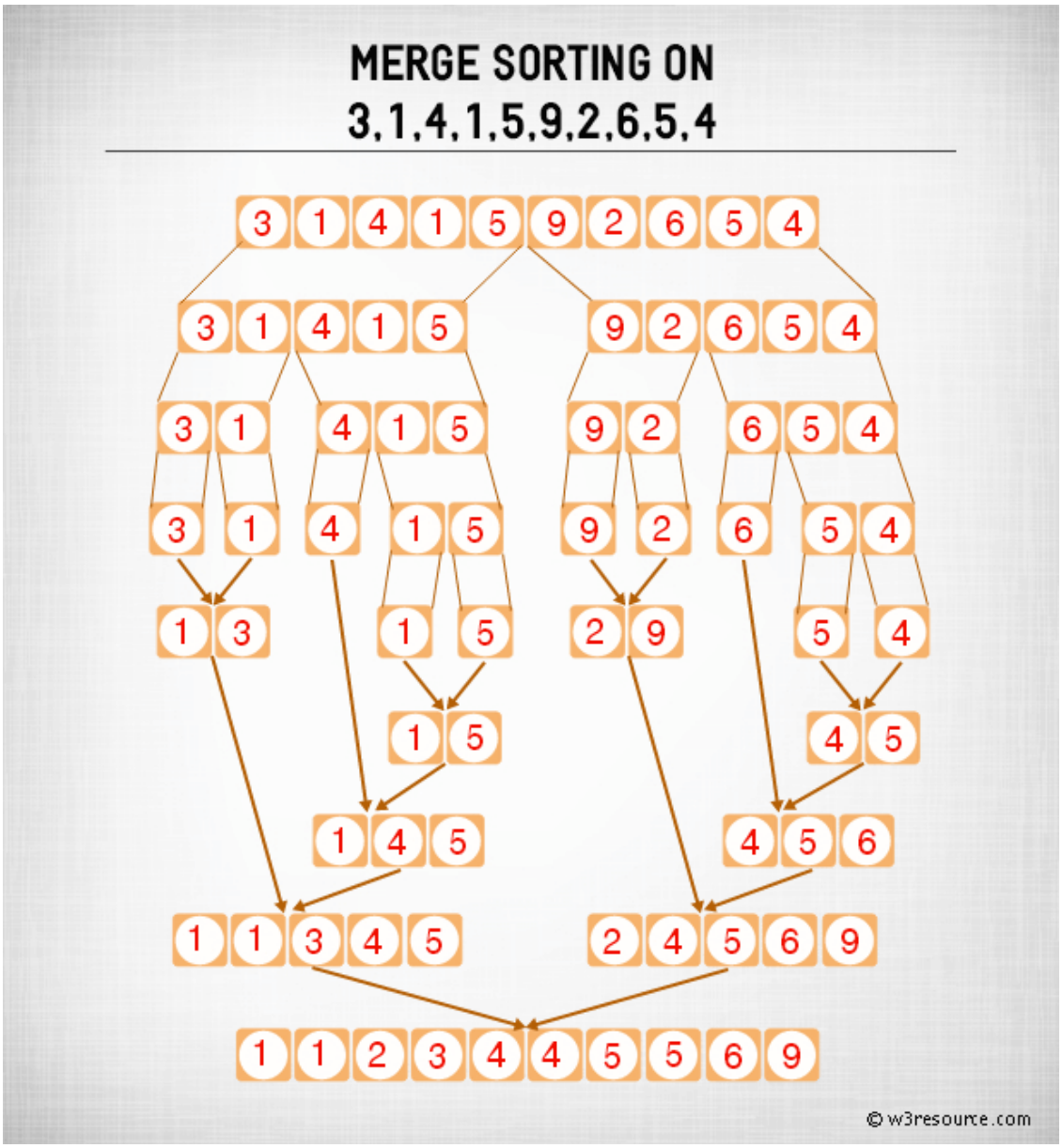
- Best Case: $O(n^2)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$
- Space Complexity: 1
- Stable: No

2.3 Merge Sort

Merge Sort was proposed by John von Neumann in 1945. This algorithm exploits a recursive divide-and-conquer approach, resulting in a worst-case running time of $O(n \log n)$. Merge sort gives good all-around performance. [6]

Conceptually, a merge sort works as follows:

- Divide the unsorted list into n sub lists, each containing one element (a list of one element is considered sorted).
- Repeatedly merge sub lists to produce new sorted sub lists until there is only one sub list remaining. This will be the sorted list. [8]



https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-2.php

[3]:
```
# Code taken from https://stackabuse.com/sorting-algorithms-in-python/

def merge(left_list, right_list):
    sorted_list = []
    left_list_index = right_list_index = 0
    # Create Variables
    left_list_length = len(left_list)
    right_list_length = len(right_list)
    for _ in range(left_list_length + right_list_length):
```

```
        if left_list_index < left_list_length and right_list_index <␣
 ↪right_list_length:
            # We check which value from the start of each list is smaller
            # If the item at the beginning of the left list is smaller, add it
            # to the sorted list
            if left_list[left_list_index] <= right_list[right_list_index]:
                sorted_list.append(left_list[left_list_index])
                left_list_index += 1
            # If the item at the beginning of the right list is smaller, add it
            # to the sorted list
            else:
                sorted_list.append(right_list[right_list_index])
                right_list_index += 1
        # If we've reached the end of the of the left list, add the elements
        # from the right list
        elif left_list_index == left_list_length:
            sorted_list.append(right_list[right_list_index])
            right_list_index += 1
        # If we've reached the end of the of the right list, add the elements
        # from the left list
        elif right_list_index == right_list_length:
            sorted_list.append(left_list[left_list_index])
            left_list_index += 1
    # final result of the sorting
    return sorted_list

##### Function definition #####
# Function performing the merge sort; it takes an array to be sorted as an␣
 ↪argument
def merge_sort(array):
    # If the list is a single element, return it
    if len(array) <= 1:
        return array
    # Use floor division to get midpoint, indices must be integers
    mid = len(array) // 2
    # Sort and merge each half
    left_list = merge_sort(array[:mid])
    right_list = merge_sort(array[mid:])
    # Merge the sorted lists into a new one
    return merge(left_list, right_list)

alist = [3,1,4,1,5,9,2,6,5,4]

print(merge_sort(alist))
```

```
[1, 1, 2, 3, 4, 4, 5, 5, 6, 9]
```

Performance

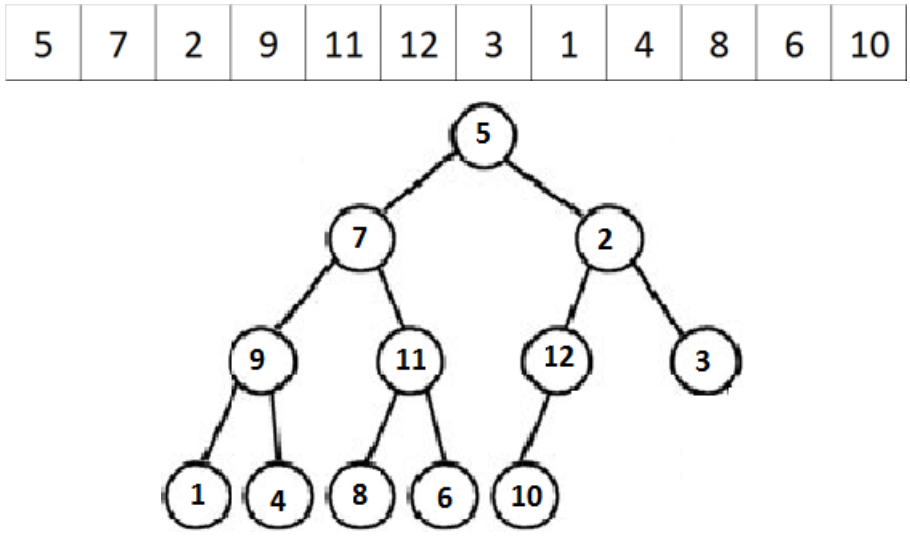
- Best Case: $O(n \log n)$
- Worst Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Space Complexity: $O(n)$
- Stable: Yes

2.4 Heap Sort

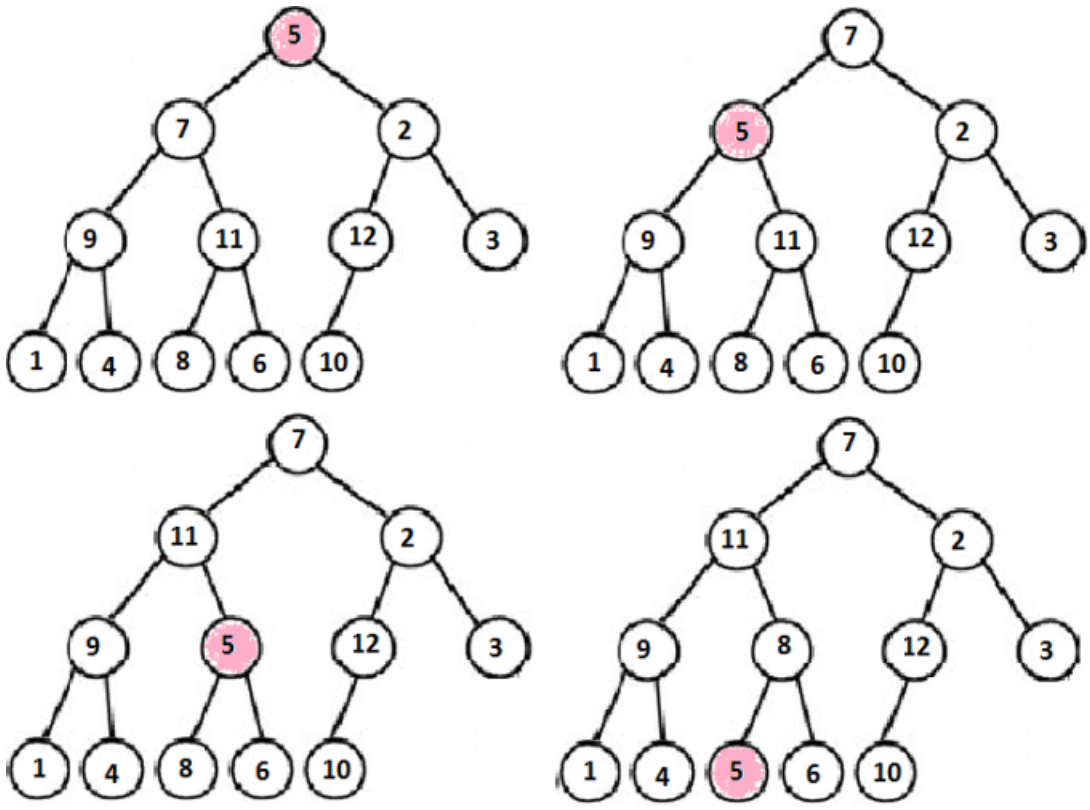
Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure.

The heap sort algorithm can be divided into two parts.

In the first step, a heap is built out of the data. The heap is often placed in an array with the layout of a complete binary tree. The complete binary tree maps the binary tree structure into the array indices, each array index represents a node, the index of the node's parent, left child branch, or right child branch are simple expressions. Let's say that we wanted to store an array of 12 elements in a binary tree then we would place them in as follows:



If we begin with 5, the first element, we see that it has two children, 7 on the left and 2 on the right. The process follows as such, we will compare all three elements so that the largest of the three is placed as the parent. We then take the parent we just placed as a child and compare it to its new children, recursively.[16]



Heap is a binary tree:

- it must always have a heap structure, where all the levels of the binary tree are filled up, from left to right
- it must be ordered as a max heap where every parent node (including the root) is greater than or equal to the value of its children nodes.

In the second step, a sorted array is created by repeatedly removing the largest element from the heap (the root of the heap), and inserting it into the array. The heap is updated after each removal to maintain the heap property. Once all objects have been removed from the heap, the result is a sorted array. [14]

Heapsort can be performed in place.

[4]:
```python
# Code taken from https://www.educative.io/edpresso/how-to-implement-heap-sort

def heapify(arr, n, i):

    largest = i # set largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2
    # See if left child of root exists and is
    # greater than root
    if (l < n and arr[i] < arr[l]):
        largest = l
    # See if right child of root exists and is
    # greater than root
    if (r < n and arr[largest] < arr[r]):
        largest = r
    # Change root, if needed
    if (largest != i):
        arr[i],arr[largest] = arr[largest],arr[i] # swap
        # Heapify the root.
        heapify(arr, n, largest)

# Function to sort an array
def heap_sort(arr):
    n = len(arr)
    # Build a heap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)
    # return sorted array
    return arr

alist = [3,1,4,1,5,9,2,6,5,4]
print (heap_sort(alist))
```

```
[1, 1, 2, 3, 4, 4, 5, 5, 6, 9]
```

Performance

- Best Case: $O(n \log n)$
- Worst Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Space Complexity: $O(n)$
- Stable: No

2.5 Counting Sort

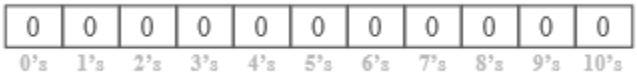
Counting sort is an algorithm for sorting a collection of objects according to keys. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. [15] First proposed by Harold H. Seward in 1954.

Counting sort runs in $O(n)$ time, making it faster than comparison-based sorting algorithms.

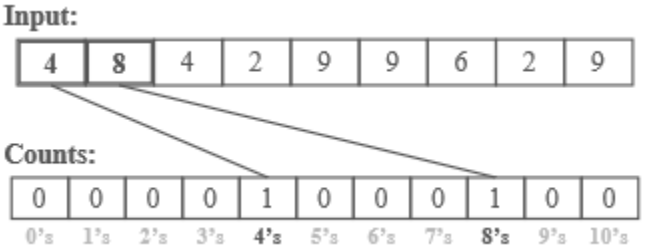
Counting sort works by iterating through the input, counting the number of times each item occurs, and using those counts to compute an item's index in the sorted array. [17]

We have an array to sort:

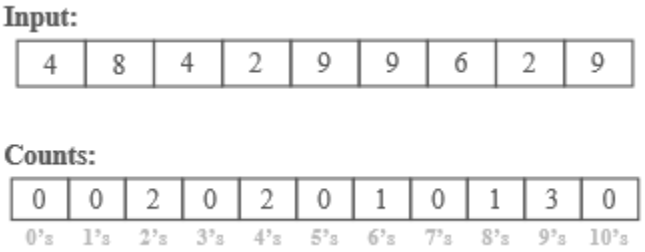
Set all elements in the count array to 0



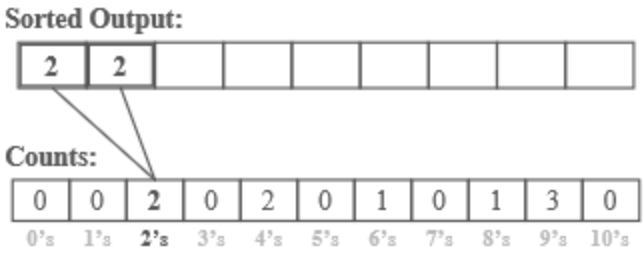
We'll iterate through the input once. The first item is a 4, so we'll add one to count[4]. The next item is an 8, so we'll add one to count[8].



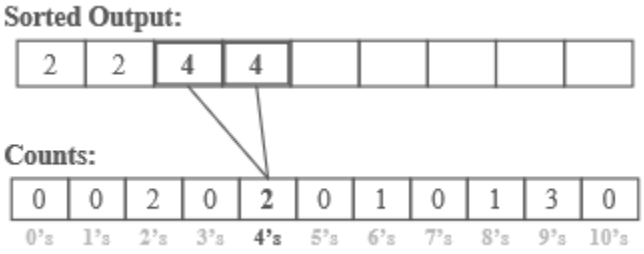
And so on. When we reach the end, we'll have the total counts for each number:



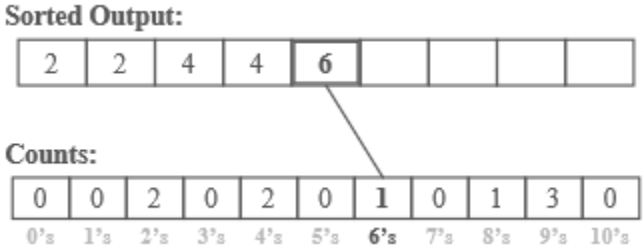
Now that we know how many times each item appears, we can fill in our sorted array. Looking at counts, we don't have any 0's or 1's, but we've got two 2's. So, those go at the start of our sorted array.



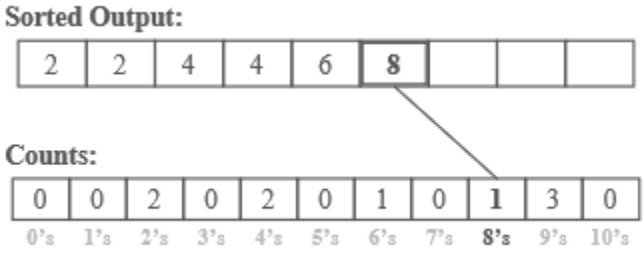
No 3's, but there are two 4's that come next.



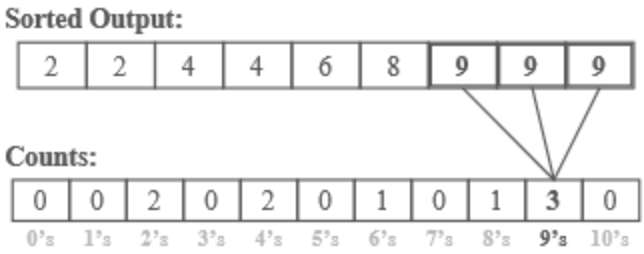
After that, we have one 6,



one 8,



and three 9's



As a result we received sorted array

[5]:
```
# Code taken from https://www.w3resource.com/python-exercises/
 ↪data-structures-and-algorithms/python-search-and-sorting-exercise-10.php
# An example of Selection sort algorithm

def counting_sort(array):

    #get the max value from the array
    maxval = max(array)
    # adding 1 to max value m
    m = maxval + 1
    # set all elements in the count array to 0
    count = [0] * m
    # checking each element in array and counts how many times element occurs
 ↪
    for a in array:
        count[a] += 1
    # sort in place, copy back into original list
    i = 0
    for a in range(m):
        for c in range(count[a]):
            array[i] = a
            i += 1
    # return sorted array
    return array

alist = [4,8,4,2,9,9,6,2,9]
print (counting_sort(alist))
```

```
[2, 2, 4, 4, 6, 8, 9, 9, 9]
```

Performance

- Best Case: $O(n)$
- Worst Case: $O(n)$
- Average Case: $O(n)$
- Space Complexity: $O(n)$
- Stable: Yes

3 Implementation & Benchmarking

In this section I will describe the results from the benchmark that was returned from running the python script benchmark.py Benchmark was run 10 times for each inout size n, then average time value was calculated and converted to milliseconds. This steps was taken for each sorting algorithm and its input size.

For benchmarking was taken 5 algorithms: - Bubble Sort - Merge Sort - Counting Sort - Heap Sort - Selection Sort

Results of benchmark are displayed in table below.

[6]:
```
import benchmark  # imported python file benchmark.py where are algorithm codes␣
 ↪and benchmark test is implemented

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
%matplotlib inline

# Run benchmark.py and store result as a pandas dataframe
df = benchmark.main()

# transpose dataframe for better readability
df.T
```

[6]:
```
Input Size       500     1100     1600     2500     3400      4500      6000   \
Bubble Sort     15.714  73.186  179.435  458.896  783.574  1393.823  2554.887
Merge Sort       0.099   0.311    0.510    0.507    0.799     1.001     1.726
Counting Sort    2.387   5.300    8.889   14.619   19.096    27.311    37.500
Heap Sort        3.813   9.489   14.902   24.964   32.913    44.808    64.692
Selection Sort  12.487  56.514  138.280  331.921  540.639  1024.570  1698.709

Input Size         7000      8500      10000
Bubble Sort     3349.083  4912.016  6767.911
Merge Sort         1.504     1.794     2.001
Counting Sort     42.009    51.100    59.609
Heap Sort         72.098    88.485   107.306
Selection Sort  2227.601  3238.407  4476.256
```

We can clearly see that the Merge Sort is the fastest sorting algorithm tested.

If we compare the chart obtained from tests with the Big O chart, we will see that the results match the theoretical values.

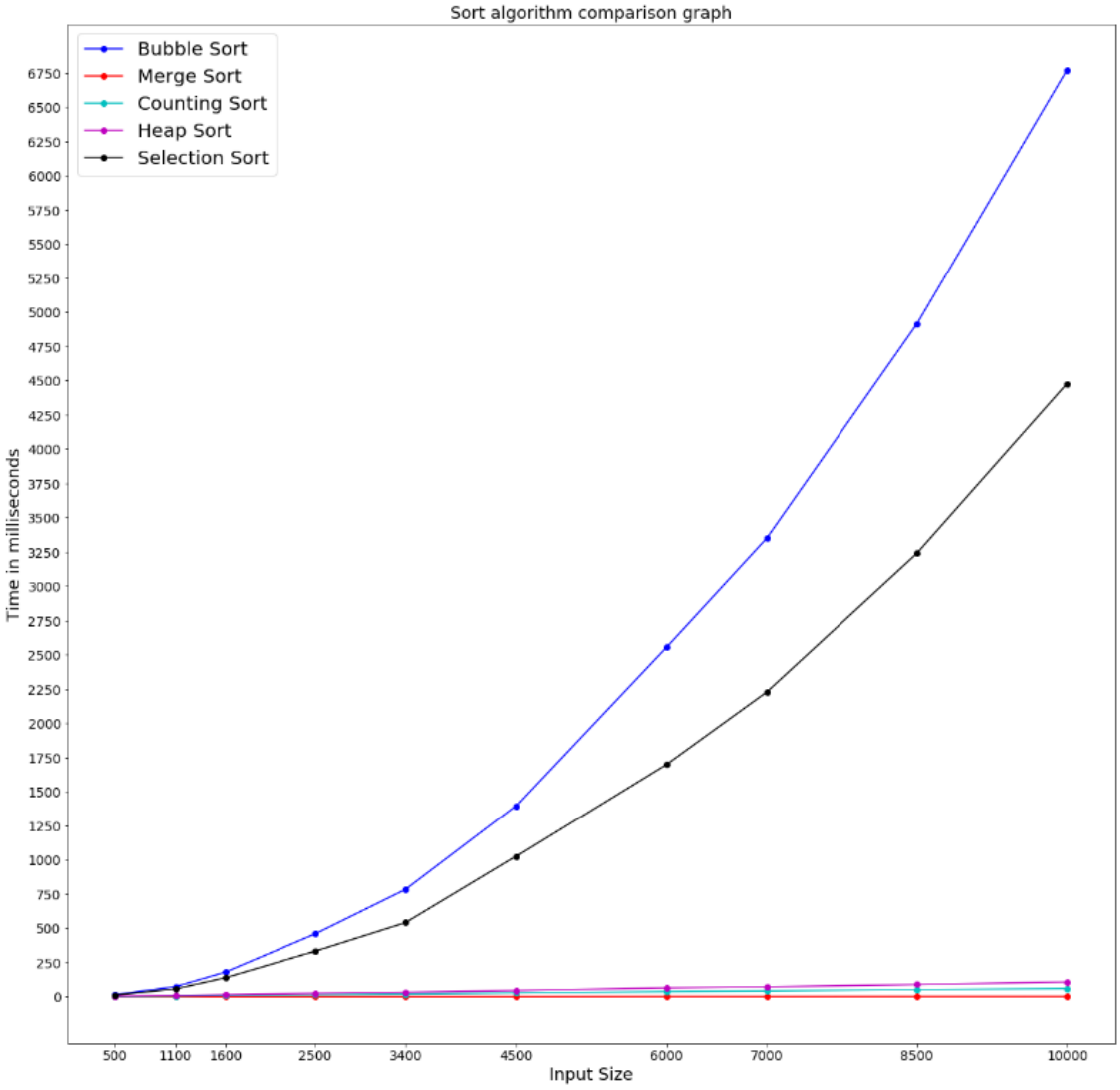
As expected (because of $O(n^2)$ average Time Complexity) Bubble Sort and Selection Sort are the slowest algorithms tested. The fastest algorithms are Merge Sort and Counting Sort.

[7]:
```
#input_size = [200,300,500,750,1000,1250,1500,2500, 3750]
input_size = df.T.columns
val_max =  int(df.values.max())

plt.figure(figsize=(20,20))
plt.plot(df.index, df['Bubble Sort'], '-bo', label='Bubble Sort')
plt.plot(df.index, df['Merge Sort'], '-ro',label='Merge Sort')
plt.plot(df.index, df['Counting Sort'],'-co', label='Counting Sort')
plt.plot(df.index, df['Heap Sort'], '-mo',label='Heap Sort')
plt.plot(df.index, df['Selection Sort'],'-ko', label='Selection Sort')

# Adding labels, legend and ticks
plt.title('Sort algorithm comparison graph',fontsize=18)
plt.xlabel("Input Size",fontsize=18)
plt.xticks(input_size,fontsize=14)
plt.yticks(range(0,val_max+25,250),fontsize=14)
plt.ylabel("Time in milliseconds",fontsize=18)

plt.legend()
plt.legend(fontsize=20)
plt.show()
```



The results we obtain largely depend on hardware and software factors, such as:

- CPU design
- System Architecture
- Background processes
- Operating system

References

[1] https://en.wikipedia.org/wiki/Algorithm

[2] https://guide.freecodecamp.org/algorithms/sorting-algorithms

[3] https://realpython.com/sorting-algorithms-python

[4] https://en.wikipedia.org/wiki/Bubble_sort

[5] https://www.geeksforgeeks.org/bubble-sort

[6] Lecture notes - GMIT Computational Thinking With Algorithm, 07 Sorting Algorithms Part 1

[7] https://www.geeksforgeeks.org/selection-sort

[8] https://en.wikipedia.org/wiki/Merge_sort

[9] https://stackabuse.com/heap-sort-in-python

[10] https://www.geeksforgeeks.org/in-place-algorithm

[11] https://www.studytonight.com/data-structures/time-complexity-of-algorithms

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

[13] https://www.geeksforgeeks.org/stability-in-sorting-algorithms

[14] https://en.wikipedia.org/wiki/Heapsort

[15] https://en.wikipedia.org/wiki/Counting_sort

[16] https://formulafunction.wordpress.com/2017/07/18/heapsort

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

Technical Helps:

- https://stackoverflow.com/a/24571145/11107506 - yticks
- https://stackoverflow.com/a/39473158/11107506 - Legend fontsize
- https://www.geeksforgeeks.org/how-to-get-column-names-in-pandas-dataframe - Data Frame column names
- https://texblog.org/2014/06/24/big-o-and-related-notations-in-latex - Latex notation
- https://www.overleaf.com/learn/latex/sections_and_chapters - Latex Paragraphs