CALIFORNIA STATE UNIVERSITY FULLERTON™

Department of Computer Science

This project has been satisfactorily demonstrated and is of suitable form.
This project report is acceptable in partial completion of the requirements for the Master of Science degree in Software Engineering.

Sorting Algorithms in Practice – A Survey
Project Title

Eti Navrotsky
Student Name

Ning Chen, Ph.D.
Advisor's Name

Advisor's signature Date

Reviewer's Name

Reviewer's signature Date

Sorting Algorithms in Practice – A Survey

THE DIFFERENCES BETWEEN THEORY & PRACTICE

Eti Navrotsky | Project 597 | Spring 2019

CSU Fullerton

Abstract

During the last decades, software engineering became an integral part of many industries. The usage of databases, in a variety of industries, increased the need for data retrieval and data dictionaries. The sorting algorithms are the basis of those operations. However, there are many platforms for executing those algorithms. As so, the combination between the algorithm and the platform is crucial for achieving the best performance.

The paper surveys the behavior of sorting algorithms on several platforms. A platform is a combination between the CPU or the GPU with a programming language. The survey is assembled from three sections: the theory section, the practice section, and the conclusion section. The theory section reviews the algorithms surveyed, with their time and space complexity; the practice section demonstrates the time performance of the algorithms in different platforms and with different array sizes, and the conclusion section comprises the results and the survey conclusions.

This paper proves the importance of preliminary knowledge before one starts coding. The differences between the performance do not leave any doubt. C++, had the best results, over other platforms and languages. The GPU, in this survey, did not achieve the expected results. In a matter of numbers, the C++, had between 50-80% better results than Java, while Java had much better performances than Python, around 95%.

In the comparison between GPU/CUDA and CPU/C++, in small arrays, C++ had better results than GPU (around 90% better). While in vast arrays CUDA had better or almost the same results. Since GPU/CUDA requires expensive hardware, and professional peoples, the advantages of using GPU, are reduced over CPU/C++.

Still, the purpose of this paper is to guide on how to choose an algorithm according to the platform one may use, and not determining which platform to use.

Table of Content

1 Introduction

With the growth in usage of software products, such as big data, embedded systems, and others, the importance of fast sorting algorithms is shown. Data retrieving and decision making are only two of many examples of those needs. Many papers, researched this subject, were published over the years. Several of them suggested new architectures, others tried to implement algorithms in new ways, and some even introduced the reader with new algorithms. However, for the average developer theoretical paper does not make any change to his daily work. This paper enters this gap by showing the performance of those algorithms in practice, and not only in theory.

To understand the complexity in theory and the performance in practice, we first need to review the Big O notation. The Big O notation or the $O(n)$ defines the time and space complexity of the algorithm. While $O(1)$ defines a constant space/time needed to execute the algorithm, the $O(n)$ defines a constant time that depends on the array size. Although in theory, the definitions for time complexity and space complexity are the same, in practice they are separate. In practice, the space $O(\log n)$ or $O(1)$ need almost the same space, while the time $O(\log n)$ or $O(1)$ is significantly different.

In theory, the complexity of the algorithm is defined by two definitions. The first one is the time complexity and the second is the space complexity. The time complexity is the time it takes to execute the algorithm. It is indicated by the Big O notation. For example, the bubble sort has a complexity of $O(n^2)$ while n is the number of elements to sort. Space complexity defines the additional space needed to execute the algorithm

In practice, the performance of the algorithm is depended on several factors. The platform one works on, the programming language used or even the processor (CPU/GPU) that runs the algorithms. Besides the technology's factors, every algorithm has its performance, that derives from its implementation. Those are the computation cost and the space needed. The computation cost is defined by the time it takes to sort the list. While space is defined by the memory needed to sort the list, some algorithms need only one element space (usually refer to algorithms with the swap function) and some algorithms need additional space as the list size to sort the list, such as Merge Sort.

The sorting algorithms are divided into two categories. The first one is comparison sorting, and the second is integer/counting sort. The comparison sorting algorithms are algorithms that compare each element with other elements in the array to decide their position. In this case, the best time complexity of the algorithm is O(*n* log *n*)[2]. The second is integer algorithms or counting algorithms. Those algorithms are useful only with the assumption that $n << 2^k$ where *n* is the number of elements to sort, *k* is the sorting keys exists and "<<" means "much less than." With this assumption, the time complexity of the sorting can be O(*n*), or less than O(*n* log *n*).

This paper comes to survey several known sorting algorithms in practice. The goal of this paper is to define the efficiency of each sorting algorithm regarding the factors described earlier. This paper investigates each algorithm in different platforms, implemented by different languages, and executes by different technologies, to be able to achieve the projects' goals.

The sorting algorithms are:

1. Selection sort
2. Insertion sort
3. Quicksort
4. Merge sort
5. Heapsort
6. Counting sort
7. Radix sort

The platforms are:

1. Windows – x64 architecture.
2. NVIDIA GPU

The program languages are:

1. Java - Object Oriented languages, compiled to virtual machines environment.
2. C++ - Object Oriented language compiled to machine code
3. Python – Object Oriented & Procedural language, interpreted

4. CUDA – For NVIDIA (GPU)

Those program languages represent the type of languages and the environments that are commonly used today. The program languages divide into two categories: 1. OO. 2. Procedural/Imperative. The running environments include 1. Complied to machine code. 2. Running on a virtual machine. 3. Interpreted 4. Graphic cards (GPU).

Report Body

The first section of this chapter reviews the sorting algorithms in theory. The algorithms are Bubble sort, Selection sort, Insertion sort, quick sort, merge sort, heap sort, Counting sort, and Radix sort. The theory of each algorithm includes a short explanation, pseudocode, an example, the time complexity and lastly the space complexity of the algorithm. The theory does not consider the platform or the technology one use to implement the algorithm.

The second section of this chapter includes the survey done for each algorithm. Here, as a contrast to the first section, every algorithm has its own execution time or cost. Sure enough, the technology, the list size, and the programming language have an impact too.

To achieve as much knowledge as needed and to encompass many programming language and technologies, the survey investigates each algorithm on several different platforms. The results will be introduced in the second section.

2 Sorting Algorithms in Theory

2.1 BUBBLE SORT

Bubble sort is a simple sorting algorithm. The algorithm sorts the elements by going from the end of the array to its begging and moving left each element that is less than the element before him. The bubble sort algorithm is straightforward to implement but its time complexity is very high, and so, it is hardly used in practice.

Disadvantage: high time complexity.

```
for i = 1 to A.length - 1
    for j = A.length downto i + 1
        if A[j] < A[j – 1]
            exchange A[j] with A[j – 1]
```

[Algorithm 1 – Bubble sort]

2.1.1 Example

Iteration 0:	1	5	7	3	4	9	3	2
Iteration 1:	1	2	5	7	3	4	9	3
Iteration 2:	1	2	3	5	7	3	4	9
Iteration 3:	1	2	3	3	5	7	4	9
Iteration 4:	1	2	3	3	4	5	7	9
Iteration 5:	1	2	3	3	4	5	7	9
Iteration 6:	1	2	3	3	4	5	7	9

2.1.2 Time complexity

The time complexity of bubble sort is $O(n^2)$ that means, for each element the algorithm checks all the elements in the array.

For n elements we search n times the array, to shift the smallest element.

n elements X n searching = $O(n^2)$

2.1.3 Space complexity

The algorithm swaps the element with the element before him, so it needs one place to execute the algorithm.

$O(1)$

2.2 SELECTION SORT

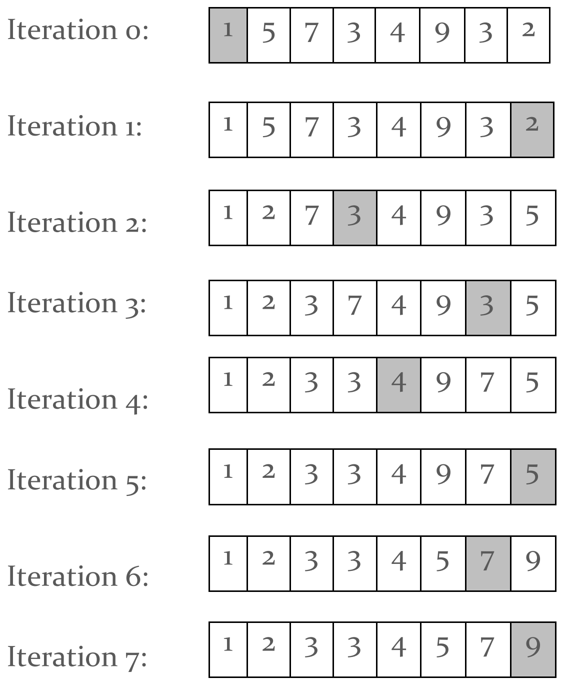
Selection sort is an algorithm to sort an array by finding the minimum element and move it to the beginning of the array. First, the array is separated into two parts, one is the sorted array, and the other is the unsorted array. Each iteration, the algorithm finds in the unsorted array the minimum element, and it swaps it to the sorted array. In this way, the number of swaps is less than in bubble sort, but still, the array needs to search n-1 times the array.

Disadvantage: high time complexity.

```
1.    for j = 1 to A.length - 1
2.        smallest = j
3.        for i = j + 1 to n
4.            if A[ i ] < A[ smallest ]
5.                smallest = i
6.        Swap A[ j ] with A[ smallest ]
```

[Algorithm 2 – Selection sort]

2.2.1 Example


Iteration 0: 1 5 7 3 4 9 3 2
Iteration 1: 1 5 7 3 4 9 3 2
Iteration 2: 1 2 7 3 4 9 3 5
Iteration 3: 1 2 3 7 4 9 3 5
Iteration 4: 1 2 3 3 4 9 7 5
Iteration 5: 1 2 3 3 4 9 7 5
Iteration 6: 1 2 3 3 4 5 7 9
Iteration 7: 1 2 3 3 4 5 7 9

2.2.2 Time complexity

The time complexity of selection sort is O(n^2) that means, for each element the algorithm checks n - 1 elements in the array, to find the minimum in each iteration.

For *n* elements we search *n* times the array, to find the min element.

n elements X *n* searching = O(n^2)

2.2.3 Space complexity

The algorithm swaps the Min element with the current element, so it needs one place to execute the algorithm.

O(*1*)

2.3 INSERTION SORT

This algorithm sorts the elements by insert each element in the correct place. Every element n_i is inserted into the correct place between $n_0..n_{i-1}$. For every element in the array, the algorithm runs *i-1* times, while i is between 1..*n*.

Disadvantage: Its worst case is O(n^2), so it is not efficient on large arrays.

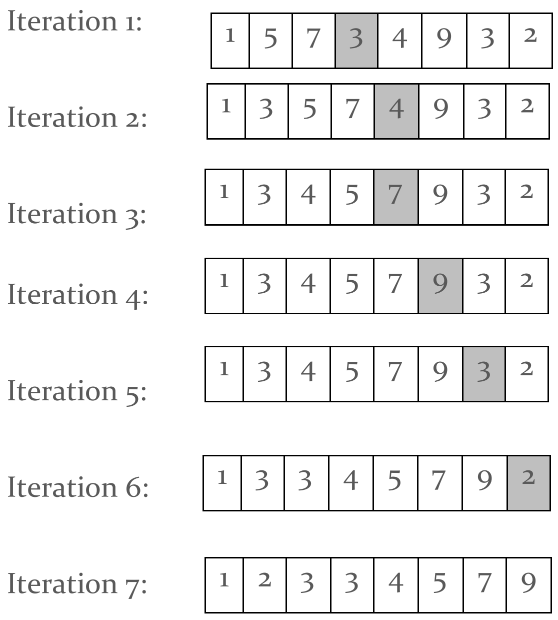
```
for $j = 2$ to A:length
    $key = A[j]$
    // Insert A[j] into the sorted sequence A[1..j−1].
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$
```

[Algorithm 3 – Selection sort]

2.3.1 Example

Iteration 0:

1	5	7	3	4	9	3	2


Iteration 1: 1 5 7 3 4 9 3 2
Iteration 2: 1 3 5 7 4 9 3 2
Iteration 3: 1 3 4 5 7 9 3 2
Iteration 4: 1 3 4 5 7 9 3 2
Iteration 5: 1 3 4 5 7 9 3 2
Iteration 6: 1 3 3 4 5 7 9 2
Iteration 7: 1 2 3 3 4 5 7 9

2.3.2 Time complexity

The time complexity of insertion sort is O(n^2) that means, for every element to sort the algorithm runs all the elements in front of it. Unlike the bubble sort, in this algorithm, we enter the second loop only if the conditions are met. So, in practice, the time complexity is less than Bubble sort.

n elements X n movements = O(n^2)

2.3.3 Space complexity

The algorithm saves the current element outside the array while it moves the elements to clear the correct place. For that, we need only one spot to sort the elements.

O(*1*)

2.4 QUICK SORT

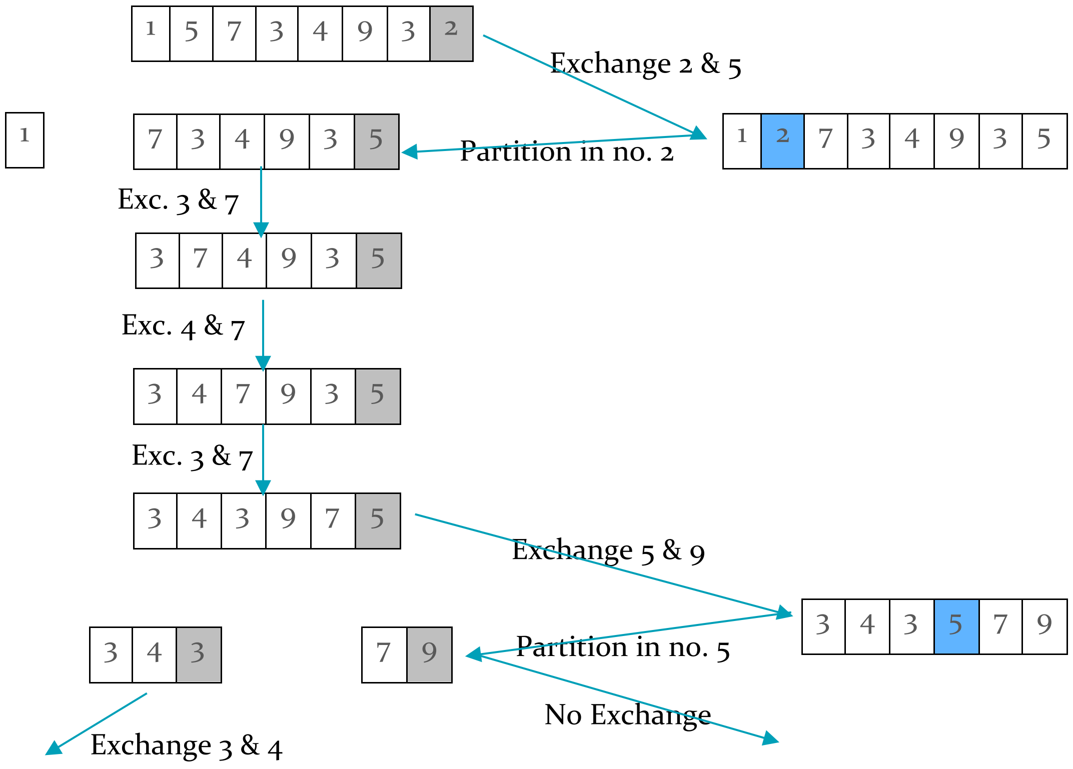
The QuickSort algorithm is a recursive algorithm like the Merge Sort, that means it sort by the divide and conquer method. However, unlike the Merge Sort here we use a pivot number. By this number, we divide the array into smallest arrays while one sub-array includes all the numbers that are smallest or equal to the pivot while the other sub-array includes all the numbers that are greatest than the pivot. The critical part of this algorithm is to find a right pivot that will split almost equally the array.

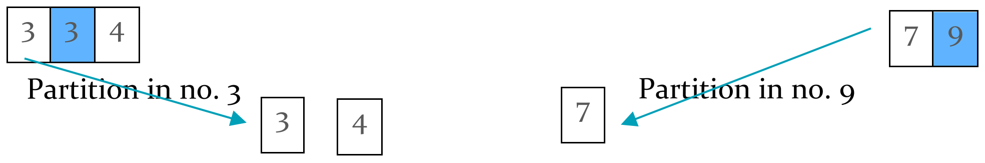
Disadvantage: In worst case its time complexity is $O(n^2)$

```
QUICKSORT (A, p, r)          // A – Array, p q & r – indexes where p <= q < r
1    if p < r
2        q = PARTITION (A, p, r)
3        QUICKSORT (A, p, q - 1)
4        QUICKSORT (A, q + 1, r)

PARTITION (A, p, r)
1    x = A[r]
2    i = p - 1
3    for j = p to r - 1
4        if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7    exchange A[i + 1] with A[r]
8    return i + 1
```

[Algorithm 4 – Quick Sort]

2.4.1 Example





Final Array:

1	2	3	3	4	5	7	9

Key:

▨ Pivot 🟦 Partition

2.4.2 Time complexity

The time complexity of the algorithm is O(n log n). Again, because this is a divide and conquer algorithm, it takes O(log n) to divide the algorithm. However, because we compare each element with the pivot number, we have more n elements to add to the time.

n comparison X n divide & conquered elements = O(n log n)

2.4.3 Space complexity

The algorithm needs to save log n sub-arrays of elements, so the space needed is O(log n).

O(log n)

2.5 MERGESORT

The Merge sort is a recursive algorithm. The Merge algorithm first divides the array into two small arrays, until the size of the sub-array is one, and then it merges them, in order. The algorithm runs recursively until the array's length is one, so the sub-array is sorted, and then it combines the sub-arrays into one sorted array.

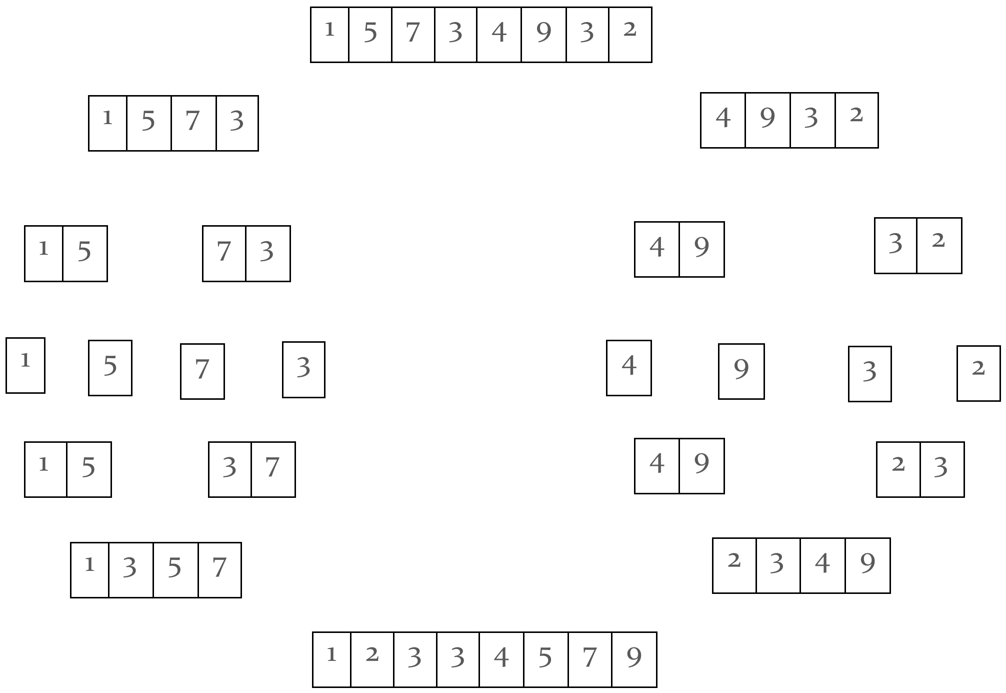
Disadvantage: Demand O(n) space to sort the algorithm.

```
MERGE-SORT(A, p, q)          // A – Array, p q & r – indexes where p <= q < r
```

```
1       if p < r
2           q = ⌊(p + r)/=2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q + 1, r)
5           MERGE(A, p, q, r)
MERGE(A, p, q, r)
1       n1 = q - p + 1
2       n2 = r - q
3       let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4       for i = 1 to n1
5           L[i] = A[p + i – 1]
6       for j = 1 to n2
7           R[j] = A[q + j]
8       L[n1 + 1] =  ∞
9       R[n2 + 1] =  ∞
10      i = 1
11      j = 1
12      for k = p to r
13          if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16          else    A[k] = R[j]
17              j = j + 1
```

[Algorithm 5 – Merge sort]

2.5.1 Example



2.5.2 Time complexity

The time complexity of merge sort is O(*n* log *n)*. The merge sort is a divide and conquers algorithm, in every call, it splits the list into two sub-lists, so, the complexity is $\log_2 n$ or log *n*. However, we have *n* sub-lists to sort, and this takes *n* times.

n elements X *n* divide & conquered elements = O(*n* log *n*)

2.5.3 Space complexity

The algorithm needs to save *n* sub-lists of one element. Thus, the complexity is *n* X 1.

O(*n*)

2.6 HEAP SORT

The Heapsort uses a binary tree to sort the elements. First for every node in the tree the left and right nodes are defined. Then, we build the tree with the Heapify function. The tree defines for every node of its parent. The parent must be bigger or equal to the node, such as A[parent(i)] >= A[i]. After the tree is built, the first node will be the highest number of the array. The Heapsort function exchanges the first node with the last node in the array. Now, when the highest number is already sorted, we call again to the Heapify, but this time without the last sorted number (A.length – 1). Because the function is recursive, each time we call the function with one less number.

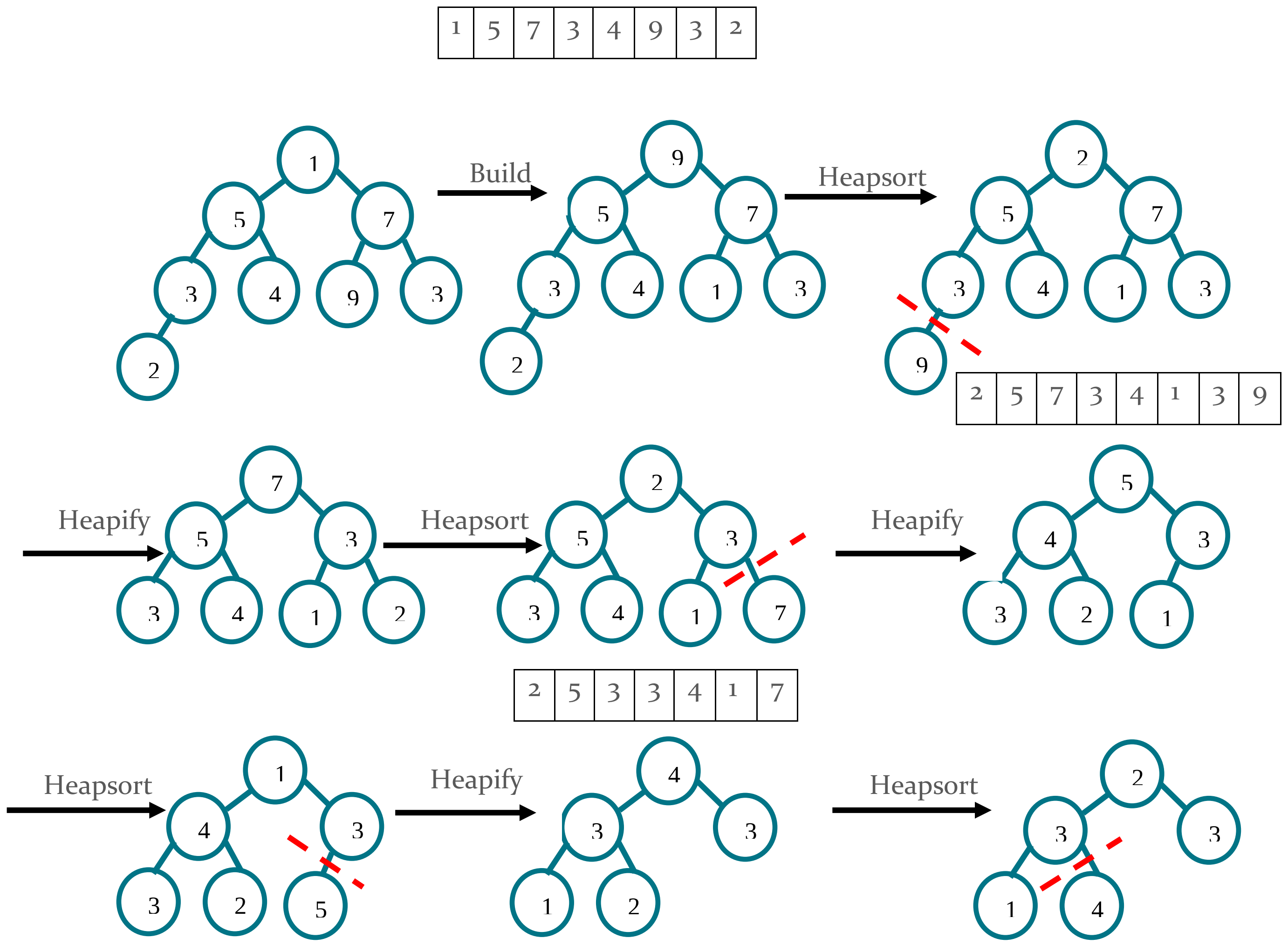
Disadvantage: None, besides the number of exchanges to create the heap.

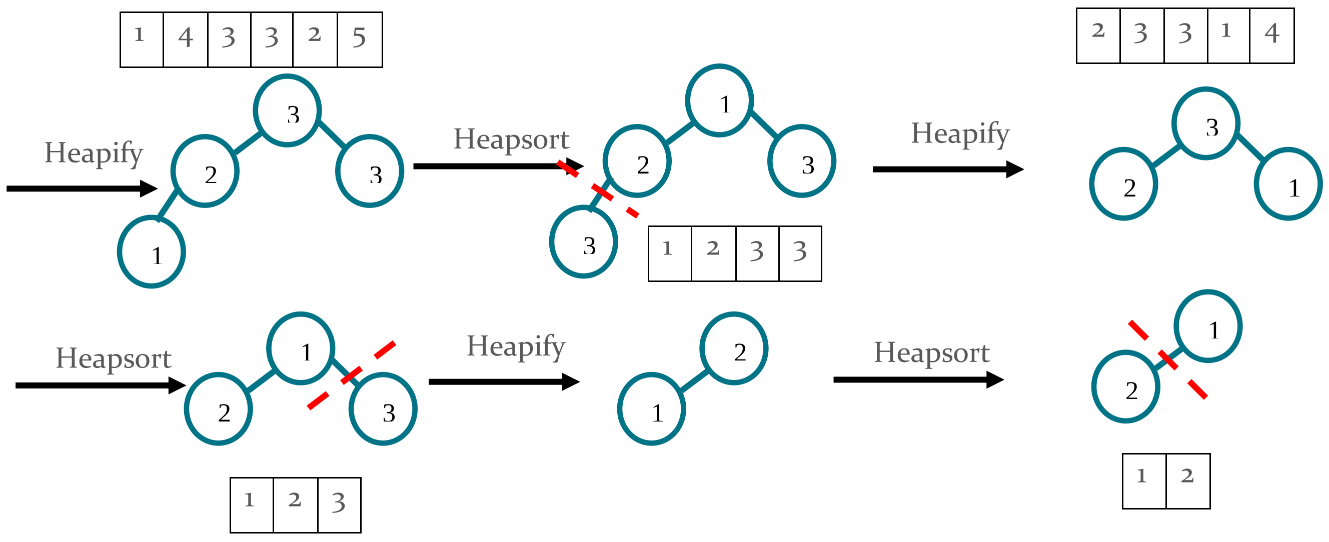
```
LEFT(i)        return 2i
RIGHT(i)       return 2i + 1

MAX-HEAPIFY(A, i)
1       l = LEFT(i)
2       r = RIGHT(i)
3       if l ≤ A.heap-size and A[l] > A[i]
4               largest = l
5       else largest = i
6       if r ≤ A.heap-size and A[r] > A[largest]
7               largest = r
8       if largest ≠ i
```

```
9           exchange A[i] with A[largest]
10          MAX-HEAPIFY(A, largest)

BUILD-MAX-HEAP(A)
1     A.heap-size = A.length
2     for i = [A.length/2] downto 1
3           MAX-HEAPIFY(A, i)

HEAPSORT(A)
1     BUILD-MAX-HEAP(A)
2     for i = A.length downto 2
3           exchange A[1] with A[i]
4           A.heap-size = A.heap-size  1
5           MAX-HEAPIFY(A, 1)
```

[Algorithm 6 – Heap sort]

2.6.1 Example





2.6.2 Time complexity

The time complexity of the algorithm is O (n log n). The Build function takes O(n) to arrange the array by A[parent(i)] > A[i]. Also, the heapsort function takes O (n log n) to sort the heap.

n elements to build the heap X log n for sorting = O (n log n)

2.6.3 Space complexity

The space complexity of Heapsort is O(1) since it exchanges the elements in the place.

O (1)

2.7 COUNTING SORT

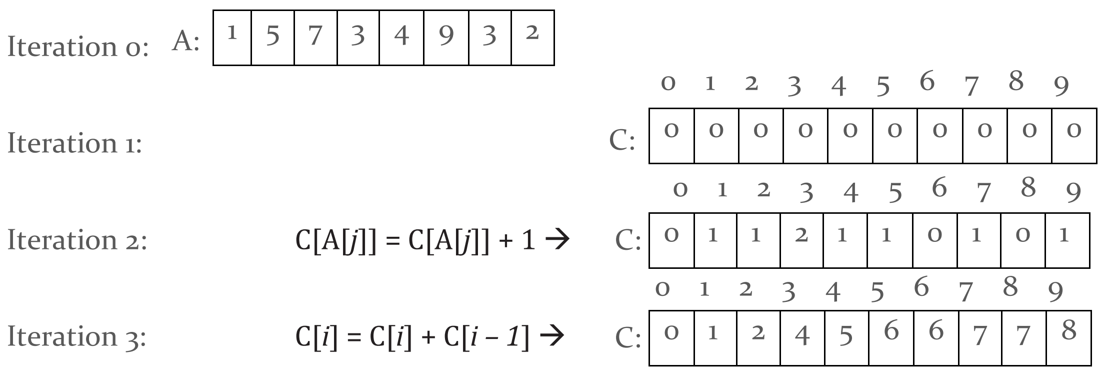
Unlike the above algorithms, counting sort is a linear algorithm. The algorithm sorts the elements by counting the appearance of each element in the 0..max(A) range. The algorithm has thee arrays, A[n] – the data, B[n] – the sorted data and C[k] – an array in size of the maximum number in A. The algorithm C includes for each number in array A, the times it appears in A, with this information it knows for each element where is its place in B. The algorithm is efficient if $k << n$.

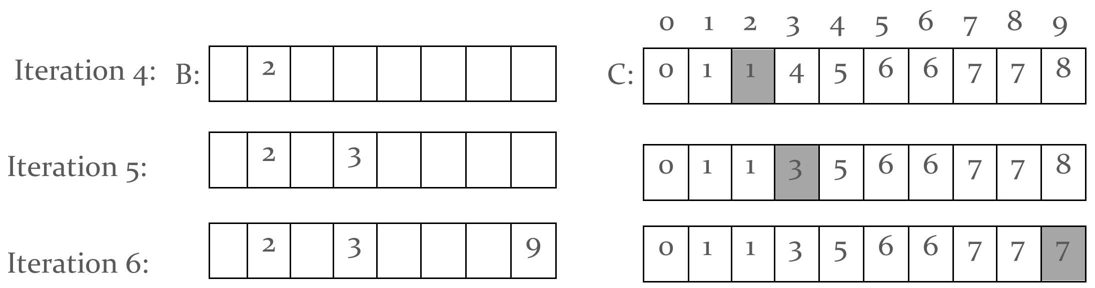
Disadvantage: Efficient only on integers.

```
// A – source array; B – sorted array; C – counting array from 0..max(A)
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
for i = 1 to k
        C[i] = C[i] + C[i – 1]
for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] – 1
```

[Algorithm 7 – Counting sort]

2.7.1 Example





Iteration 7:			2		3	4			9		0	1	1	3	**4**	6	6	7	7	7
Iteration 8:			2	3	3	4			9		0	1	1	**2**	4	6	6	7	7	7
Iteration 9:			2	3	3	4		7	9		0	1	1	2	4	6	6	**6**	7	7
Iteration 10:			2	3	3	4	5	7	9		0	1	1	2	4	**5**	6	6	7	7
Iteration 11:	B:	1	2	3	3	4	5	7	9	C:	0	**0**	1	2	4	5	6	6	7	7

2.7.2 Time complexity

The time complexity of counting sort is O(n + k). Since the algorithm includes only the *for* function, it needs to run O(n) for the sorting and O(k) for the counting.

O(n) + O(k) = O(n + k)

2.7.3 Space complexity

The algorithm needs two addition arrays. B[n] – for the sorted data, and C[k] for the counting.

2.8 RADIX SORT

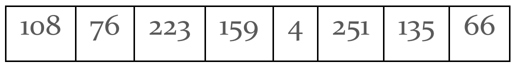
Radix sort, like counting sort, it is a linear algorithm. The algorithm sorts the numbers by sorting each iteration the least significant digit first, and then the second least significant digit and so on until all digits are sorted. When the algorithm finished sorting all digit, the array is sorted. This algorithm is a stable algorithm, and it uses a stable sorting algorithm to sort each digit (most likely by counting sort). The advantage of radix sort is when we have several keys that we need to sort by. For example, when we are sorting by date (mm/dd/yyyy), in this case, we can sort parallelly.

Disadvantage: The algorithm is used with integer numbers.

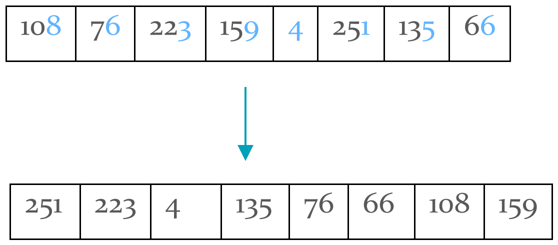
```
// A- an array of integers, d – number of digits of the max number
1 for i = 1 to d
2 use a stable sort to sort array A on digit i
```

[Algorithm 8 – Radix sort]

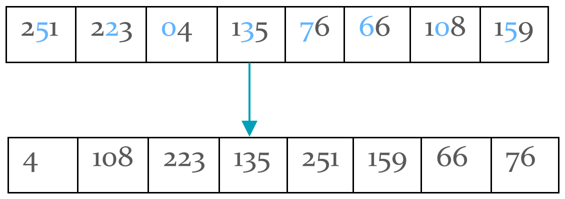
2.8.1 Example

A:


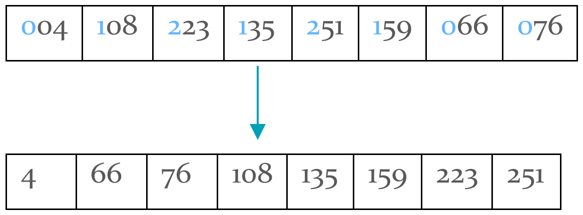
Iteration 1:



Iteration 2:



Iteration 3:



2.8.2 Time complexity

The time complexity of radix sort is O($d(n + k)$). It needs to run d times on array, and each time to sort by counting sort, which has a complexity of O($n + k$).

d times O(n) + O(k) = O($d(n + k)$)

2.8.3 Space complexity

Since the algorithm sorts in each iteration by the digit, it needs A[n] for the array, B[k] for the counting of each digit.

3 Sorting Algorithms in Practice

The objective of this section is to capture the results of using the above algorithms in practice. In the previous section, seven algorithms were introduced including each algorithm's complexity, in a manner of time and space. As explained in the introduction section, the variance between the theory and practice may change the selections one takes.

3.1 THE PLATFORM

NVIDIA – GeForce 1050 TI

CPU – i5 Intel

3.2 THE ENVIRONMENT

The survey executed the algorithms in four languages/compilers:

1. C++ - Microsoft C++ Compiler version 19.16.27027.1 x64
2. JAVA – version 1.8.0_201
3. Python – Version 3.7.2
4. CUDA – Version 10.0

3.3 IMPLEMENTATION INFORMATION

For executing the algorithms, four different arrays were created. The arrays were built randomly, with a uniform distribution. Several array sizes were checked, and the selected arrays are:

1. A_1: 100K elements, zero to 1K range.
2. A_2: 100K elements, zero to 1M range.
3. A_3: 2M elements, zero to 1K range.

4. A_4: 2M elements, zero to 1M range.

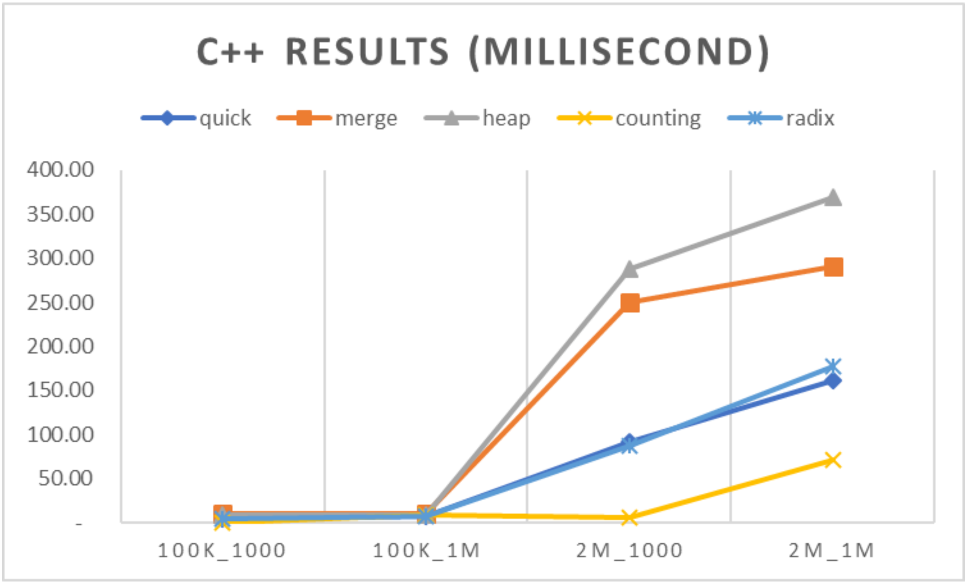
Since the values of the elements are integers [4 Bytes], the difference between the ranges is mainly to survey the linear algorithms, the counting, and radix sort.

During the survey, the execution time was recorded in milliseconds. The execution time in CPU included the time it took to execute the algorithm. In GPU it included the time to copy the memory from/to the CPU too. The time it took to create the array from the file, was excluded in both platforms.

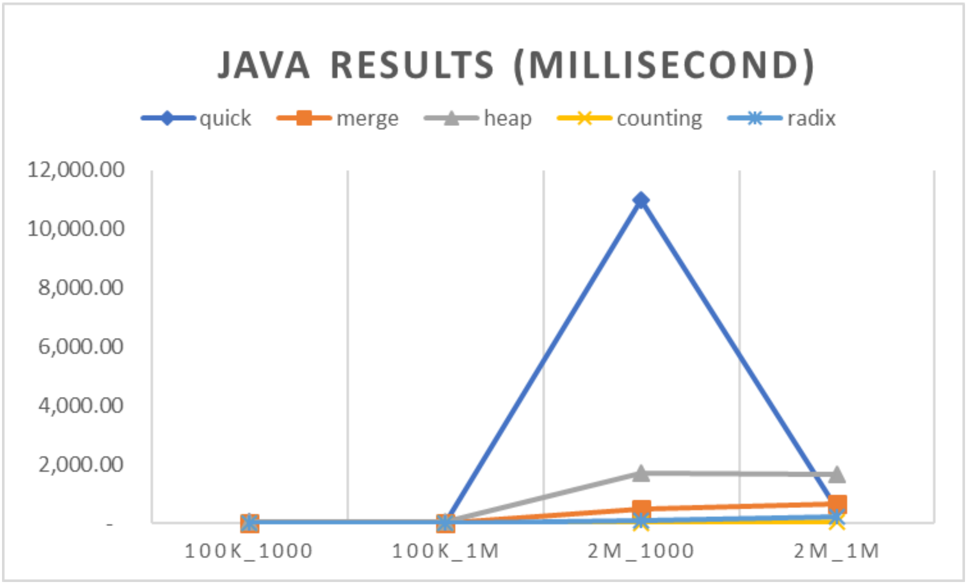
3.4 THE SURVEY RESULTS

Throughout the years, several papers were published on sorting algorithms. Thus, parts of the results were expected. However, in practice, one shall never take anything for granted. Questions should always be asked. The platform, the hardware, and the code have an impact on the results.

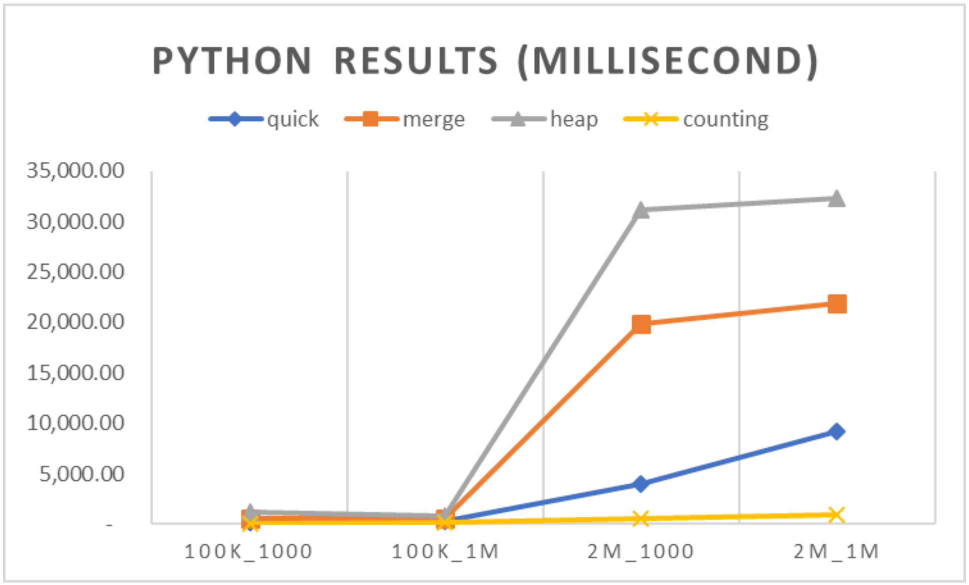
The survey investigated the performance of seven algorithms on the CPU platform and two algorithms on the GPU platform. On CPU, five of the algorithms were comparison algorithms, and the other two were counting algorithms. Two of the comparison algorithms have a time complexity of $O(n^2)$ in their worst case. Those algorithms were far inferior to the others. Their results will be shown only in the tables, and not in the graphs. The other five algorithms (Quick, Merge, Heap, Counting, and Radix) are presented in graphs and tables.



[Graph 1 – C++ performance]



[Graph 2 – Java Performance]



[Graph 3 – Python Performance]

The results are separated into three categories:

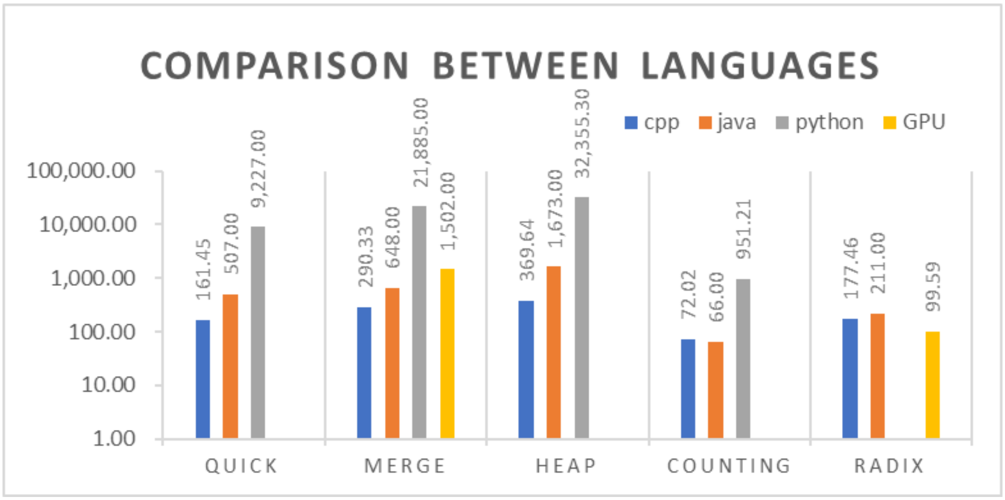
1. Comparison between languages.
2. Comparison between algorithms.
3. Comparison between platforms.

3.4.1 Comparison between languages

When comparing between languages, the C++ language is the best language between the four (C++, JAVA, Python, CUDA), considering the aspects of small arrays, cost of hardware and expert developers.

Java and C++ have close results in the non-comparison algorithms. However, in the comparison algorithms results the difference between the languages is high. Thus, when one can use non-comparison algorithms, he can choose either C++ or Java. However, in the case of developing with comparison algorithms, the preferred language is C++.

As shown in graph 4, the difference between the languages that are compiled into machine code and Python, that is an interpreted language brings to only one conclusion. A language that is not compiled to a machine code should not be used.



[Graph 4 – performance on array 2M_1M (Logarithmic scale)]

Graph 4 shows the performance of the algorithms on an array size of 2M and an element range of 1M. The graph uses a logarithmic scale, to be able to include all the algorithms on one scale.

3.4.2 Comparison between algorithms

As shown previously, C++ is the best language, and the counting sort is the preferred algorithm for sorting.

First, the counting sort algorithm had the best results from all seven algorithms. Since, as mentioned before, all four arrays included integer numbers, the advantage of counting sort over the other algorithms were shown [see tables 1-3].

	100k_1000	100K_1M	2M_1000	2M_1M
Selection Sort	321.00	10,291.10	4,114,140.00	4,071,280.00
Insertion Sort	1,471.01	1,436.74	608,054.00	604,593.00
Quick Sort	4.37	6.93	91.83	161.45
Merge Sort	10.61	10.96	249.51	290.33
Heap Sort	9.03	9.02	287.15	369.64
Counting Sort	0.26	8.53	5.93	72.02
Radix Sort	4.10	8.34	87.58	177.46

[Table 1 – C++ Results in Millisecond]

	100k_1000	100K_1M	2M_1000	2M_1M
Selection Sort	10,051.00	13,093.00	8,011,829.00	>>24hrs
Insertion Sort	11,629.00	12,715.00	16,291,072.00	>>24hrs
Quick Sort	38.00	22.00	10,968.00	507.00
Merge Sort	26.00	25.00	471.00	648.00
Heap Sort	40.00	43.00	1,712.00	1,673.00
Counting Sort	5.00	13.00	31.00	66.00
Radix Sort	17.00	21.00	120.00	211.00

[Table 2 – Java Results in Millisecond]

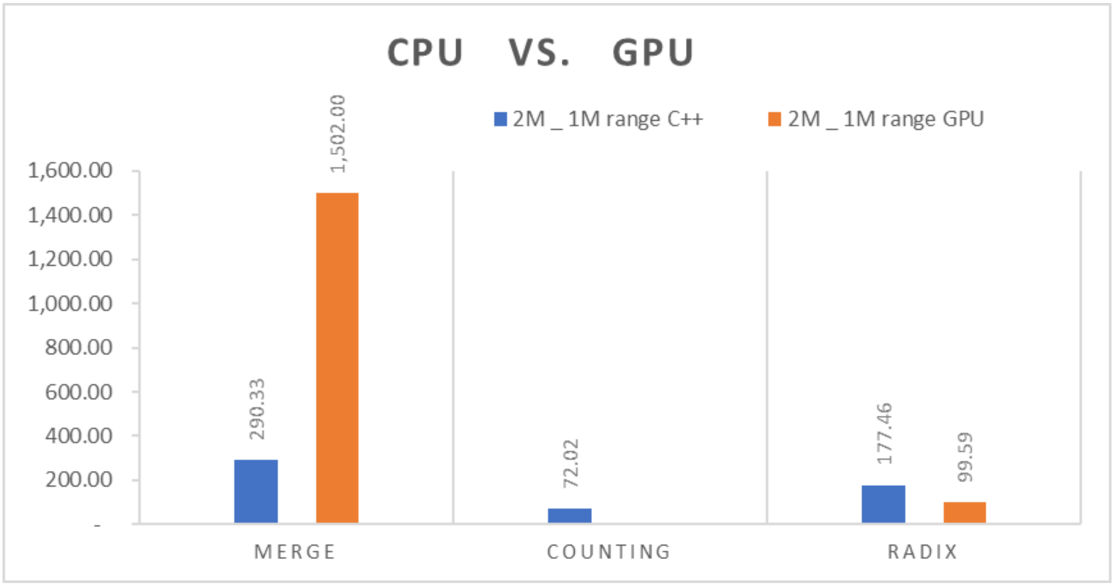
	100k_1000	100K_1M	2M_1000	2M_1M
Selection Sort	349,548.00	363,284.00	>>24hrs	>>24hrs
Insertion Sort	593,640.00	631,870.00	>>24hrs	>>24hrs
Quick Sort	113.60	233.38	3,949.00	9,227.00
Merge Sort	556.50	550.52	19,800.00	21,885.00
Heap Sort	1,145.90	749.90	31,110.00	32,355.30
Counting Sort	31.90	231.00	535.54	951.21
Radix Sort	19,688.30	19,478.00	517,014.00	681,589.00

[Table 3 – Python Results in Millisecond]

As shown in table 1, in one array (100K_1M), the quicksort had better performance than the counting sort. The most reasonable explanation is that the numbers range was significantly higher than the array size. In this case, the advantage of counting over quicksort is not relevant.

Second, from the comparison algorithms, the quicksort performance is better than any other comparison algorithm. This result matches other papers that demonstrate the quicksort abilities over other comparison algorithms. In one case, where the array was 2M elements and 1M range [see table 2], the quicksort, had the worst results. After more investigation, it showed that the partition index, was always the last number, of the array. Since the range was small (1,000), as a result, it took a long time to execute the program. In case that the array had only zeros, the program crashed with a stack overflow, from the same reason of a lousy division.

3.4.3 Comparison Between Platforms



[Graph 5 – performance on array 2M_1M (Milliseconds)]

Graph 5 shows the comparison between the platforms. As mentioned before, the radix sort algorithm was implemented by Nvidia with the Cub addition. The cub is a library with components for parallel coding. The merge sort was implemented independently, by a CUDA developer. While Radix sort, had the same execution time, for any size array [see table 4], the merge sort execution time was not stable. The execution time of both algorithms show, from different aspects, the importance of expert developers on CUDA.

Another important time consuming is the memory allocation and the memory copy to/from the GPU. To sort the array, CUDA first allocates memory for the array on the graphics card, and then, it copies the data.

The disadvantages of GPU are the hardware's cost and the necessity of expert developers. Those two disadvantages reduce the usage of this platform across developers.

Overall, between those platforms, a developer may prefer the C++ platform, then the GPU platform. However, the performance of the radix sort shows the advantage GPU has over any programming language.

	100K_1000	100K_1M	2M_1000	2M_1M
Merge sort	1,137.00	1,125.00	2,055.00	1,502.00
Radix Sort	99.83	92.21	97.72	99.59

[Table 4 – GPU results in Millisecond]

4 Conclusion

This survey reiterates the importance of preliminary knowledge in sorting algorithms before one starts developing. The sorting algorithms were and still are the basis of many software programs. The differences above, explain simply, how to choose a sorting algorithm and, if applicable, which programming language accommodates one needs.

The survey investigated seven sorting algorithms in practice. The results showed that comparison algorithms with time complexity of $O(n^2)$ in theory, should be excluded in practice. Since, in any size array, and any platform, their results were far inferior to the other algorithms.

Additionally, the survey introduced CPU and GPU performances. It showed without any doubt that C++ is the best programming language between the three (C++, Java & Python). The C++ was able to execute vast arrays in a reasonable time. For example, in one case, the C++ ran the selection sort, known with $O(n^2)$, in less than ten hours. For comparison, Python needed more than one month to execute the same algorithm.

Since this survey investigated only integer numbers, the advantage of counting sort and radix sort demonstrated in all platforms. On the other side, between the quick sort, merge sort and heap/binary sort, the quicksort had the best performance.

After the CPU investigation, the survey introduced GPU performance. Although better results were expected in the GPU running time, in practice different results were seen. The merge sort implementation did not have better performance than the CPU algorithm. Moreover, it executed the algorithm inefficiently in comparison with the CPU. On the other side, on large arrays, the radix sort had better results than the CPU. This demonstrates that when the arrays are large, the efficiency of GPU exceeded CPU, even when the time of copying the array is counted.

Lastly, the survey defines the best algorithm per platform. Sometimes, one cannot select the platform for programming. For those cases, the survey helps to choose between the algorithms, with emphasis on the size and range of the arrays.

For conclusion, understanding this survey shall help one select the best algorithm per platform. However, as shown before, this survey has its limitations. Thus, more surveys with different constraints and assumptions may clarify the subject more.

5 References

[1] Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

[2] Ashok Kumar Karunanithi, June 2014, "*A Survey, Discussion and Comparison of Sorting Algorithms*", Department of Computing Science Ume˚a University

[3] Dmitri I. Arkhipov, Di Wu, Keqin Li, and Amelia C. Regan, Sep. 2017, *"Sorting with GPUs: A Survey",* University of California, Irvine

[4] Gabriele Capannini, Fabrizio Silvestri, Ranieri Baraglia, Jan. 2011, "*Sorting on GPUs for large scale datasets: A thorough comparison",* Information Processing and Management

[5] Keliang Zhang and Baifeng Wu, 2012, "*A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess*", 2012 IEEE 14th International Conference

Sorting Algorithms, Brilliant.org https://brilliant.org/wiki/sorting-algorithms/

Pseudocodes

Algorithm 1 – P.40, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Algorithm 2 – selection sort, http://www.algolist.net/Algorithms/Sorting/Selection_sort

Algorithm 3 – P.18, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Algorithm 4 – P.171, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Algorithm 5 – P.31-34, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Algorithm 6 - P.151-161, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Algorithm 7 – P.195, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Algorithm 8 – P.198, Cormen, Leiserson, Rivest, Stein (2009), Introduction to Algorithms 3rd edition, Massachusetts Institute of Technology.

Code

https://github.com/kevin-albert/cuda-mergesort

https://elixir.bootlin.com/linux/latest/source/lib/sort.c

https://www.programiz.com/dsa/insertion-sort

https://techiedelight.com/compiler/

http://www.algolist.net/Algorithms/Sorting/Quicksort

https://www.geeksforgeeks.org/radix-sort/

https://www.geeksforgeeks.org/selection-sort/

https://nvlabs.github.io/cub/

https://www.inertia7.com/projects/53

https://www.geeksforgeeks.org/heap-sort/

https://www.geeksforgeeks.org/merge-sort/

https://www.w3resource.com/python-exercises/data-structures-and-algorithms/

https://www.pythoncentral.io/insertion-sort-implementation-guide/

https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Python

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html

6 Appendixes

All the source codes are attached to this project in a Zip file. To run each module, you should copy the text files with the arrays into the library of each programming language. Those files are sitting in the “data” library.

CPP files – library “cpp”, main file “filetoarray.cpp”

Java files – library “java”, main file “test2.java”

Python files – library “python”, main file “main.py”

CUDA files – library “cuda”, to run merge sort choose “mergesort_kernal”, to run radix sort choose “radixsort_kernal”