Jakub Korytko (album no. [redacted])

Part-time computer science bachelor's degree

(Year 1, Semester 2, Ex. Gr. 1)

Sorting algorithms

Algorithms and Data Structures

How the program works

In this paper we will compare the well-known complexity of sorting algorithms with our own implementation. We will make the comparison on the basis of a console program in C language, which has a menu for algorithm selection and an option to run all algorithms in sequence. The program runs in the following steps:

1. Specifying the size of the array
2. Approval with enter
3. Getting acquainted with the array loaded from the file
4. Confirm with enter that the correct array has been loaded
5. Selecting the sorting algorithm (or another menu option)

Before starting the program, you should:

1. Place the `data` file (without extension) in the root directory. It must contain numbers (random by design) separated by spaces. You can use the `tools/generate` tool to generate the file.
    a. To use this tool, first run the `make generate` command in the root directory. This will create a `generate`[1] file in the `tools/out` folder. The syntax of the command (assuming you are in the root directory) is:

```
tools/out/generate <number_of_elements> <filename>
```

[1] On a Windows operating system it will be a file with the extension ".exe"

Both arguments are optional, defaulting to "100000" and "data". On Windows, you may need to enclose the command (without arguments) in quotation marks.

2. Once we have the `data` file in the root directory, we can compile the program. (To be clear, this was possible before, but it didn't make much sense; without the `data` file, the program wouldn't run properly) We run `make` in the root directory.
3. This creates a `sort`[2] file in the root directory.

The file used to run the program is `sort`[3] . We run it by calling its name, i.e. the `sort` command in the root directory.

[2] On a Windows operating system it will be a file with the extension ".exe"
[3] See above

The application also includes a program to generate and save the results of the running time of the algorithms to "*.txt" files (for the 3 tests listed later in this document) for the given array sizes. It's also have to be compiled, with the command `make results` in the root directory. This will create the `results`[4] file in the `tools/out` folder. The command syntax (assuming you are in the root directory) is:

```
tools/out/results <number_of_calls> <numbers_file_name>
<algorithms_indices_from_the_array> (separated by
spaces, between 0 and 5)>
```

`<number_of_calls>`: each successive call starts sorting for the number of elements specified by the sequence: 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000...

i.e. 1 call - for 1000 elements,

2 calls - for 1000 el., for 2000 el.,

3 calls - for 1000 el., for 2000 el., for 5000 el.,

...

When using the command for N calls, the time of each algorithm call will be saved to the files, for each of the 3 tests separately.

[4] On a Windows operating system it will be a file with the extension ".exe"

`<algorithms_indices_from_the_array>`:

0 – Insertion sort,

1 – Selection sort,

2 – Bubble sort,

3 - Quicksort,

4 - Shellsort,

5 - Heapsort

After running the command, the result files available in the `results` directory will be created.

Since the operation of the program is already clear, we will try to approximate the time complexity of the algorithms based on Microsoft Excel graphs. The data for the graphs were generated with the command `tools/out/results` .

Algorithms studied

Group I

- Insertion sort
- Selection sort
- Bubble sort

Group II

- Quicksort
- Shellsort
- Heapsort

These groups differ in time complexity but also in implementation difficulty. Group 1 is extremely easy to implement but quite slow (very bad for large array sizes), while group 2 is fast but more difficult to implement.

To better illustrate the differences in implementation complexity, we will show these algorithms in pseudo-code and in block diagrams.

Group I algorithms - description

- Insertion sort

The insertion sort algorithm involves taking elements one by one and inserting them into the correct places in the already sorted sequence of elements. It can be compared to arranging cards taken from a deck, where we first take the first card and then take more cards until we have exhausted the entire deck. We compare each card we take with the cards we already have in our hand and look for a suitable place for it in the sequence, so as to keep it in order. If the new element is larger than the elements in the sequence, we place it at the end, otherwise we insert it in the appropriate place before the elements already sorted.

- Selection sort

The selection sort algorithm involves finding the element with the smallest value and swapping it with the element in the first position. In this way, we place the element with the smallest value in its proper position. We repeat the same process for the remaining elements of the set until all elements are sorted.

- Bubble sort

The bubble sort algorithm is based on comparing adjacent elements and swapping their order if they do not meet the order criterion. This process is performed recursively until the entire collection is sorted.

Group I algorithms - features

PROS

- These algorithms are very simple to implement

- They can be easily derived on their own based on knowledge of the principle of operation

- For a small amount of data they fulfill their role

- They are a good example for learning sorting concepts; complex algorithms such as sorting by mounding could make it difficult for people who have not been exposed to sorting before to understand the concept

CONS

- These algorithms are characterized by time complexity $O(n^2)$

- Much better sorting algorithms are known

- They are impractical for large board sizes

Group I algorithms - pseudocode

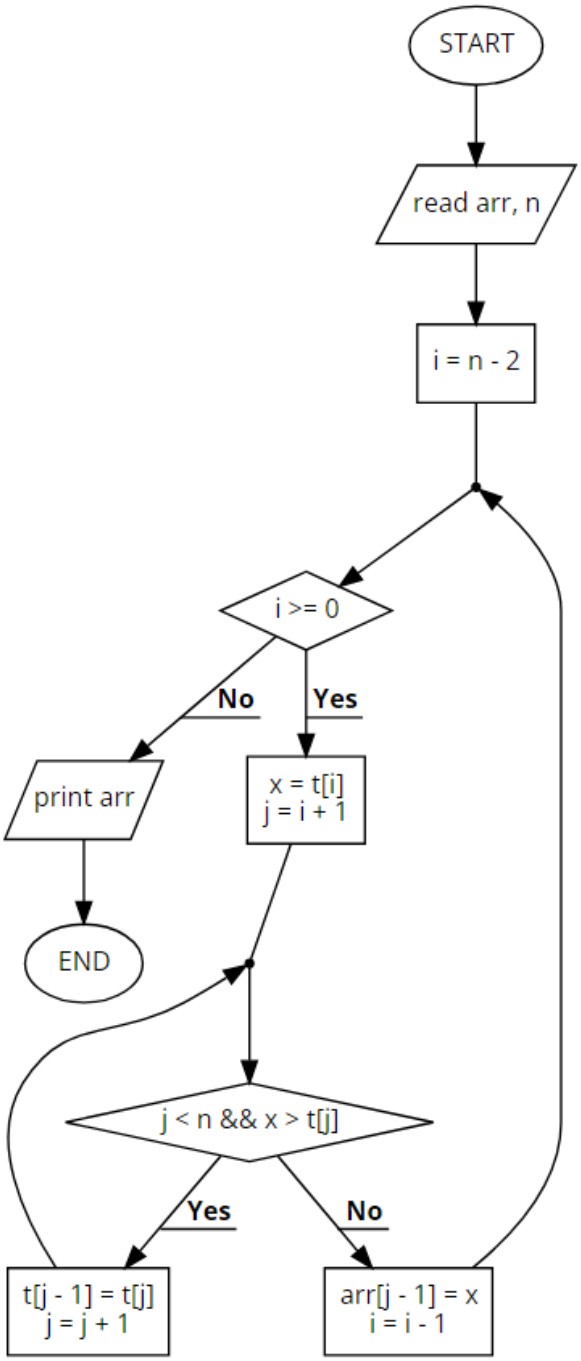
```
// Insertion sort

From i = array length - 2 to i >= 0 {

    Set x = t[i] and j = i+1

    As long as j < array length and x > t[j] {
        t[j - 1] = t[j]
        Set j = j + 1
    }

    Set d[j - 1] = x
}

// Selection sort

From i = 0 to i < array length - 1 {
    Set min = i

    From j = i + 1 to j < array length {
        If t[j] < t[min] then set min = j
    }

    Swap t[min] with t[i]
}

// Bubble sort

From i = 0 to i < array length - 1 {
    From i = 0 to i < array length - 1 {
        If t[j]>t[j+1], replace them
    }
}
```

Group I algorithms - block diagrams

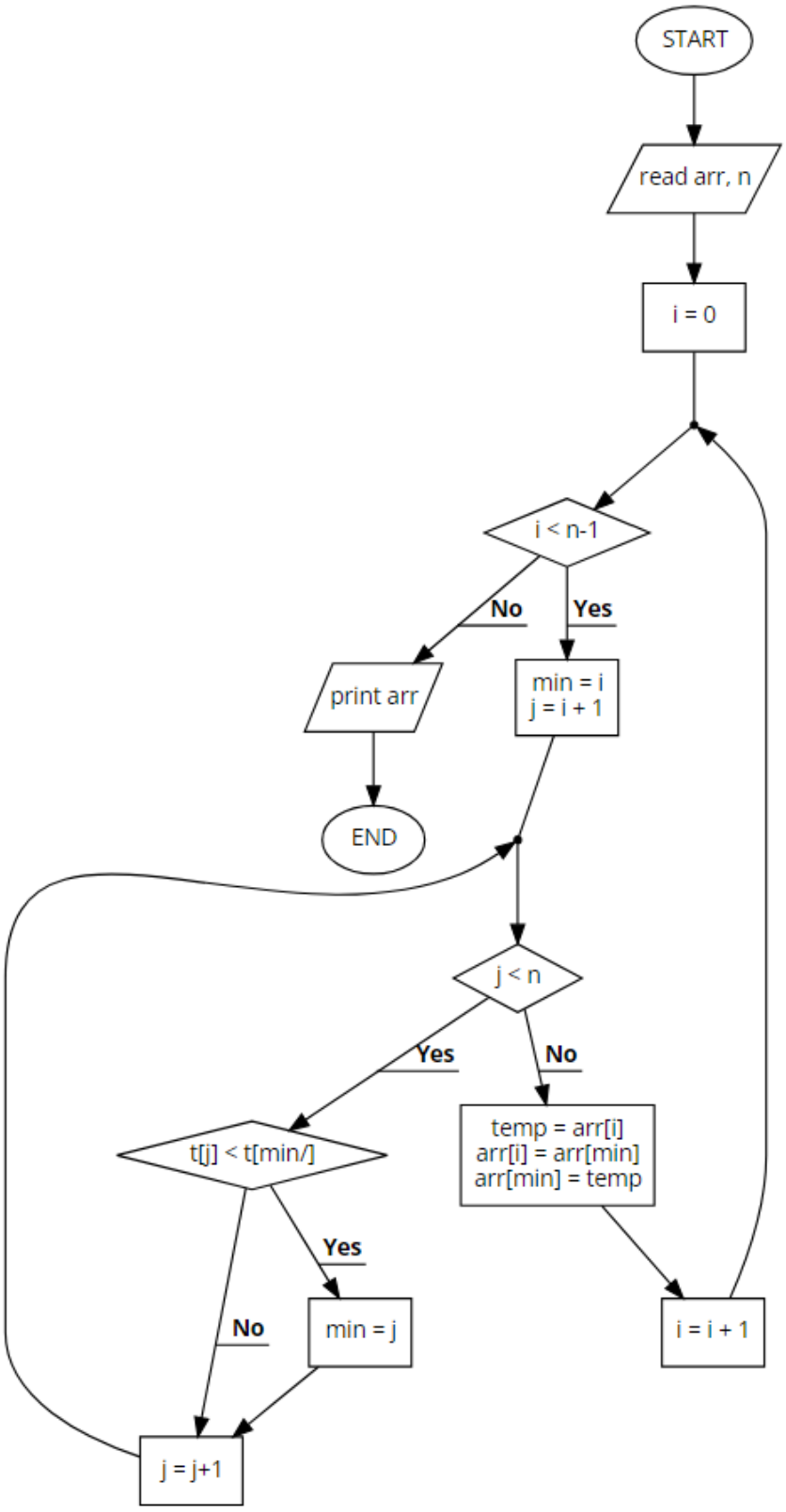
n: *array size*

arr: *array with the elements to sort*

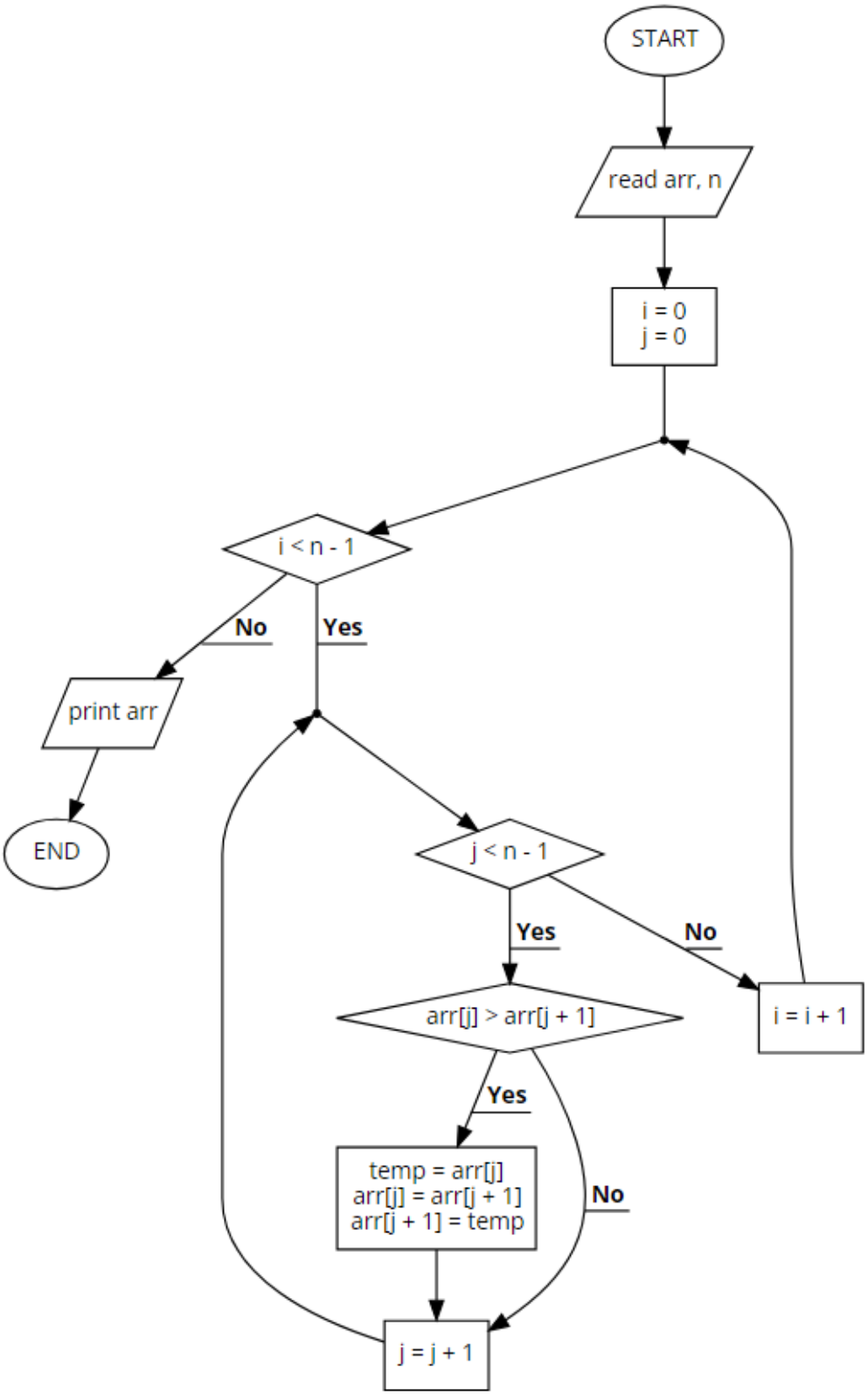
Insertion sort



Selection sort



Bubble sort


START
read arr, n
i = 0
j = 0
i < n - 1
No
Yes
print arr
END
j < n - 1
Yes
No
arr[j] > arr[j + 1]
i = i + 1
Yes
temp = arr[j]
arr[j] = arr[j + 1]
arr[j + 1] = temp
No
j = j + 1

Group II algorithms - description

- Quicksort

The quicksort algorithm involves dividing a sorted set into two parts such that all elements on the left side (called the left partition) are less than or equal to all elements on the right side (called the right partition). Each partition is then sorted recursively using the same algorithm. After both partitions are sorted, they are combined into a single sorted set.

- Shellsort

The Shellsort algorithm involves dividing a set into subsets and sorting each of these subsets using the insertion sort algorithm with different spacing. Then the spacing is reduced, and the sorting and reducing operations are repeated until the spacing is equal to 1, after which we sort using the insertion sort algorithm.

- Heapsort

Heapsort is a method for sorting elements in a tree structure, called a heap. A heap is a tree in which each node has a value no less than that of its parent. The algorithm for heapsort involves building a heap of unordered elements, then pulling out the element with the highest value and placing it at the end of the array until the entire array is sorted.

Group II algorithms - features

PROS

- These algorithms are characterized by their time complexity $O(n * \log n)$ (with the exception of Shell's sorting algorithm, but it is nevertheless the fastest of the algorithms with a class of $O(n^2)$

- Thanks to their speed, large arrays of data can be sorted

- These are very fast algorithms

CONS

- These algorithms are relatively difficult to understand (as with Group I algorithms, simple logic is enough to visualize how they work, it is not so obvious, for example, in the case of the heapsort algorithm which requires computer knowledge - heaps, binary trees, etc.).

- The quicksort and Shellsort algorithm can degrade to a class of $O(n^2)$

- These algorithms are difficult to implement

Group II algorithms - pseudocode

```
// Quicksort

Function(arr, left, right):

    Set i = left - 1, j = right + 1
    Set pivot = arr[Integer of (left + right) / 2]

    Until (true):
    {
        As long as (arr[++i] < pivot)
        As long as (arr[--j] > pivot)

        If i <= j: replace arr[i] with arr[j]
        Otherwise: break
    }

    If left < j then Function(arr, left, j)
    If right > i then Function(arr, i, right)

Function(arr, 0, n-1)
```

```
// Shellsort

From h = 1 to h < array length, 3 * h + 1
h = Integer of h / 9
If h == 0 then set h = h + 1

As long as h != 0: {
    From i = array length - h - 1 to i >= 0: {
        Set x = t[i], j = i + h

        As long as j < array length and x > t[j]: {
            Set t[j - h] = t[j] and j = j + h
        }
        Set t[j - h] = x
    }
    Set h = Integer of h / 3
}
```

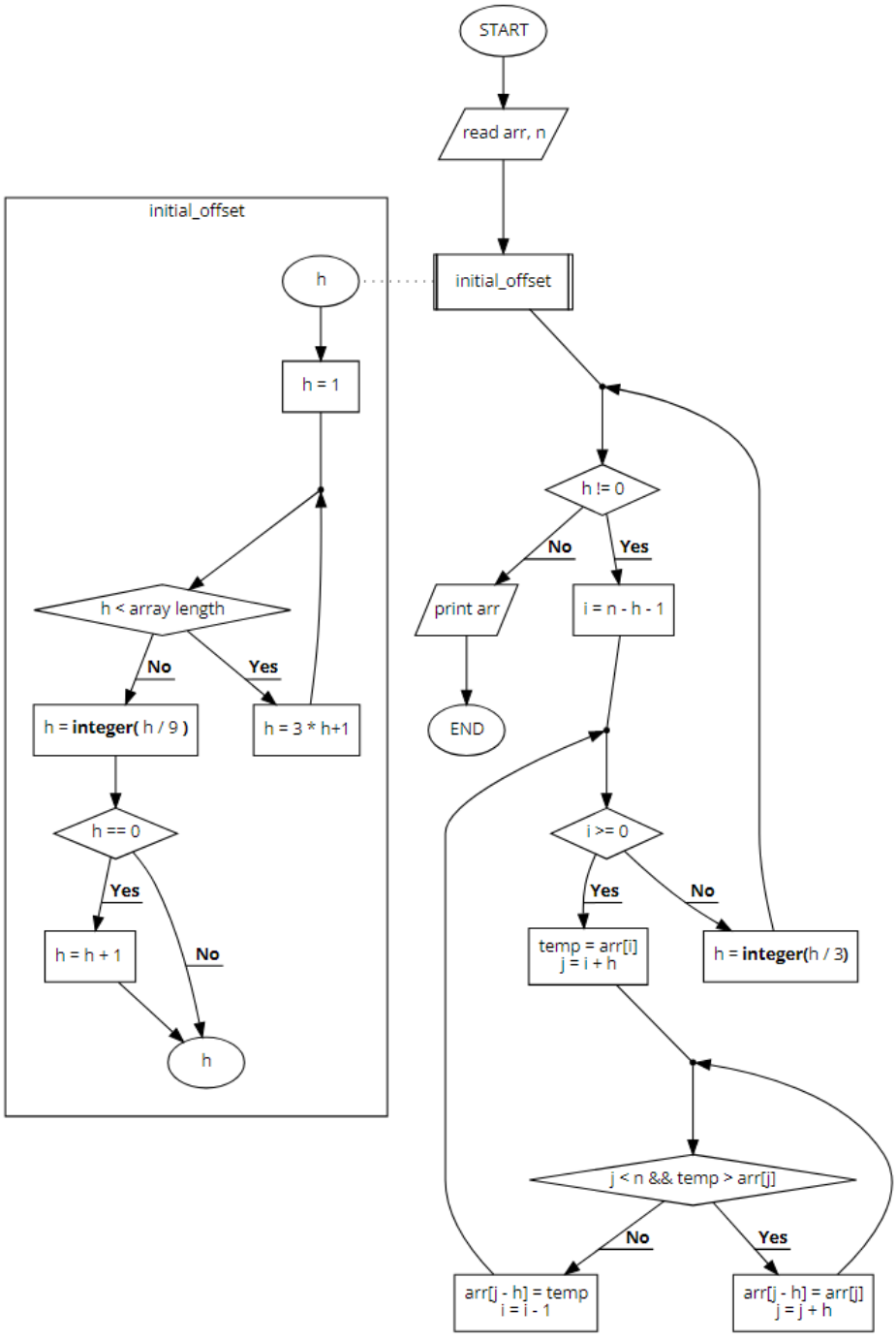
```
// Heapsort

// Building a heap

From i=2 to i<= the length of the array:
    Set j=i, Integer k = j / 2, x = arr[i]
    As long as k>0 and arr[k] < x: {
        Set arr[j] = arr[k],
        j = k, k = Integer j / 2
    }
    arr[j] = x
}

// Dismantling of the heap

From i = array length to i > 1: {

    Swap arr[1] with arr[i]
    Set j = 1, k = 2

    As long as k < i: {
        If k + 1 < i and arr[k + 1] > arr[k]:
            m = k + 1
        Otherwise:
            m = k

    If arr[m] <= arr[j], end of loop (break)
    Replace arr[j] with arr[m]
    Set j = m, k = j + j
    }
}
```

Group II algorithms - block diagrams

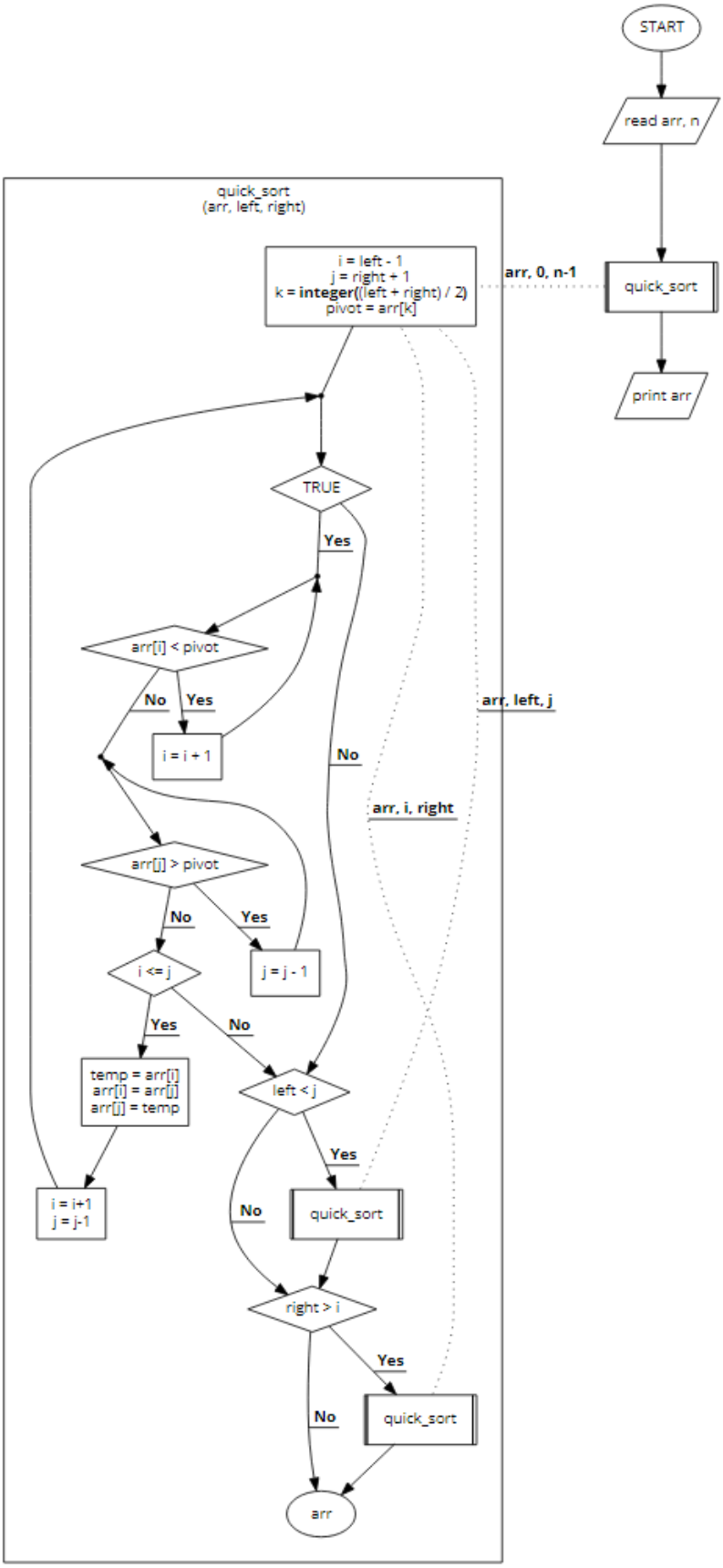
n: *array size*

arr: *array with the elements to sort*

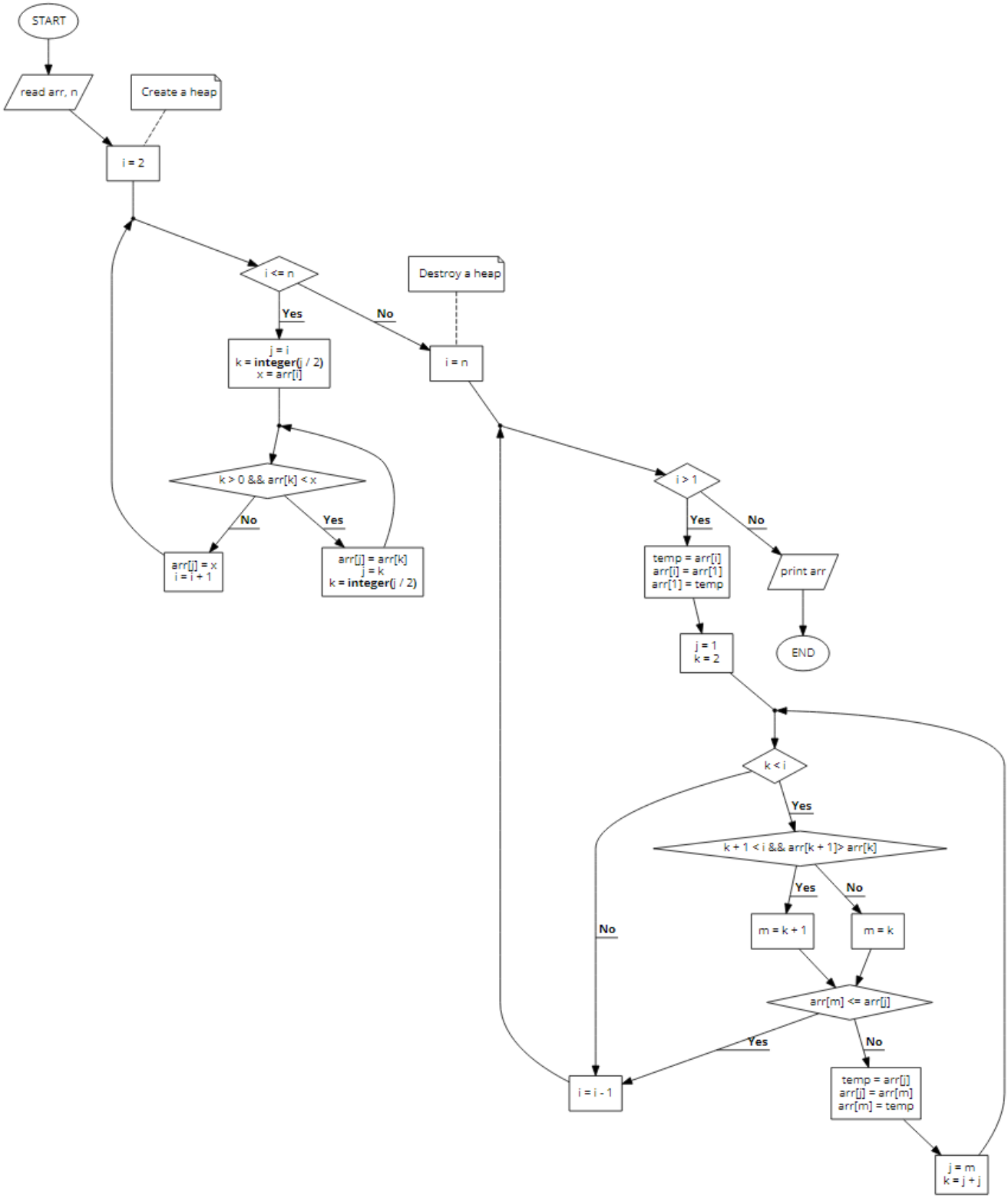
Shellsort



Quicksort


START
read arr, n
quick_sort
print arr
arr, 0, n-1
quick_sort
(arr, left, right)
i = left - 1
j = right + 1
k = integer((left + right) / 2)
pivot = arr[k]
TRUE
Yes
arr[i] < pivot
No
Yes
i = i + 1
No
arr, left, j
arr, i, right
arr[j] > pivot
No
Yes
j = j - 1
i <= j
Yes
No
temp = arr[i]
arr[i] = arr[j]
arr[j] = temp
left < j
Yes
i = i+1
j = j-1
No
quick_sort
right > i
Yes
No
quick_sort
arr

Heapsort


START
read arr, n
Create a heap
i = 2
i <= n
Yes
No
j = i
k = integer(j / 2)
x = arr[i]
k > 0 && arr[k] < x
No
Yes
arr[j] = x
i = i + 1
arr[j] = arr[k]
j = k
k = integer(j / 2)
Destroy a heap
i = n
i > 1
Yes
No
temp = arr[i]
arr[i] = arr[1]
arr[1] = temp
print arr
END
j = 1
k = 2
k < i
Yes
No
k + 1 < i && arr[k + 1]> arr[k]
Yes
No
m = k + 1
m = k
arr[m] <= arr[j]
Yes
No
i = i - 1
temp = arr[j]
arr[j] = arr[m]
arr[m] = temp
j = m
k = j + j

Algorithm tests

We will run 3 tests for each group of algorithms, then try to approximate the time complexity of the algorithms and compare their summary graphs.

Test1 - For randomly generated data

Test2 - For data sorted in reverse order (descending)

Test3 - For data sorted properly (ascending)

Time is given in microseconds (to get seconds, multiply times by 10^{-6})

Test results (table)

A table of results for all 3 tests based on which we will create charts.

Test 1			
N	Bubble sort	Insertion sort	Selection sort
1000	5000	1000	4000
2000	17000	2000	5000
5000	156000	19000	39000
10000	578000	89000	161000
20000	2257000	304000	640000
50000	14318000	1938000	4288000
100000	69017000	13257000	16594000
200000	255626000	37141000	75299000
N	Heapsort	Quicksort	Shellsort
1000	<1	<1	<1
2000	<1	<1	<1
5000	<1	<1	<1
10000	2000	2000	2000
20000	5000	3000	3000
50000	13000	4000	10000
100000	22000	8000	16000
200000	55000	18000	48000
500000	148000	79000	111000
1000000	342000	100000	200000
Test 2			
N	Bubble sort	Insertion sort	Selection sort
1000	9000	2000	1000
2000	59000	5000	6000
5000	185000	55000	47000
10000	515000	192000	183000
20000	1770000	778000	777000
50000	15920000	4874000	4697000
100000	64649000	24089000	17894000
200000	189851000	86573000	69549000

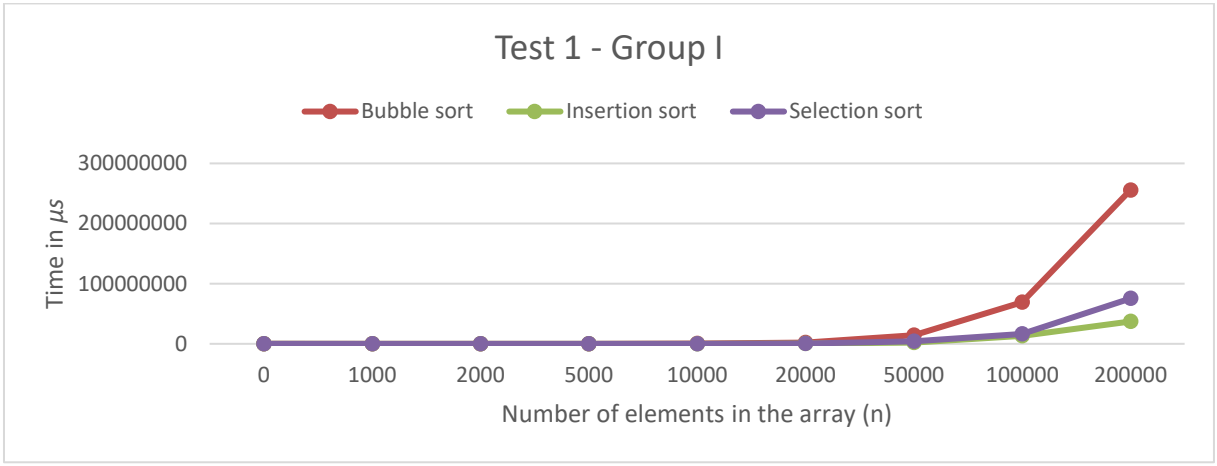
N	Heapsort	Quicksort	Shellsort
1000	<1	<1	<1
2000	<1	<1	<1
5000	2000	<1	<1
10000	2000	1000	<1
20000	5000	1000	1000
50000	9000	3000	2000
100000	21000	6000	8000
200000	49000	13000	16000
500000	126000	37000	44000
1000000	210000	77000	74000

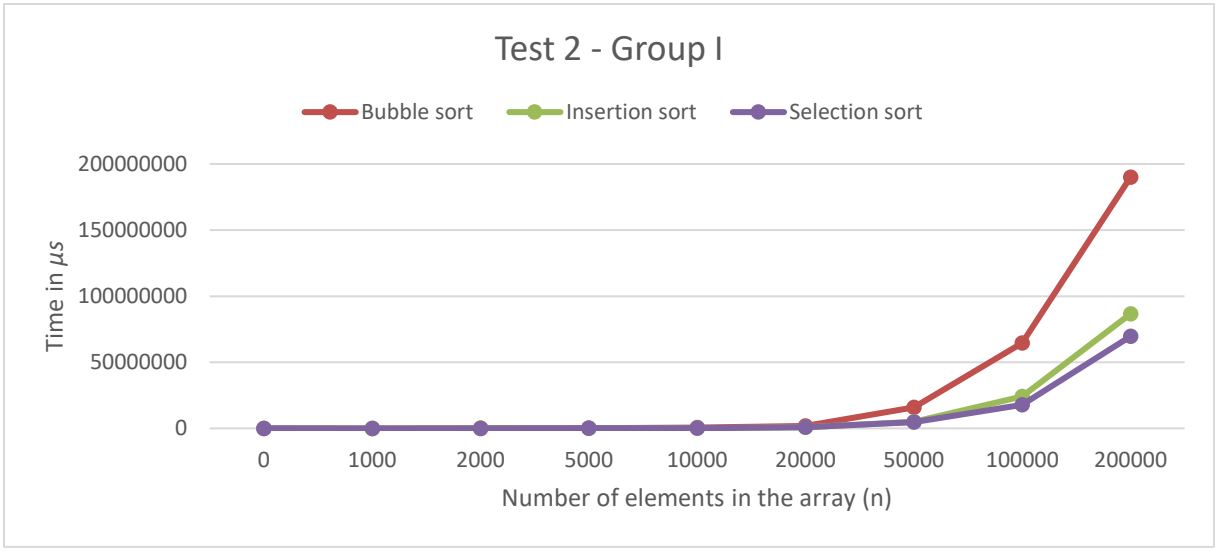
Test 3

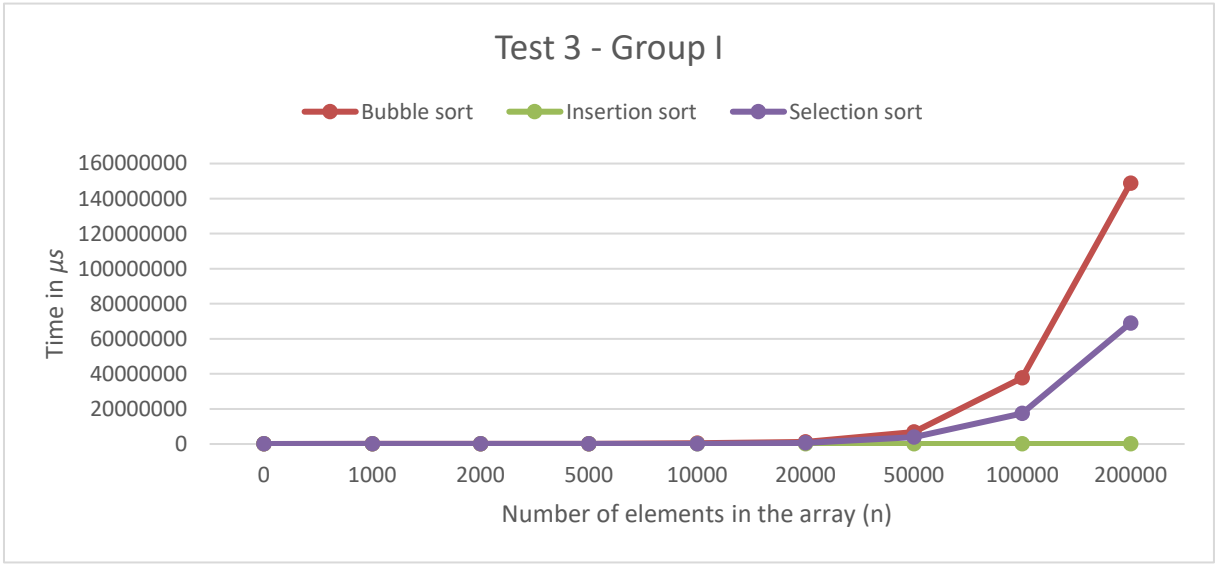
N	Bubble sort	Insertion sort	Selection sort
1000	4000	<1	2000
2000	16000	<1	6000
5000	80000	<1	39000
10000	364000	<1	157000
20000	1165000	<1	653000
50000	6880000	<1	3985000
100000	37649000	<1	17378000
200000	148781000	1000	69007000

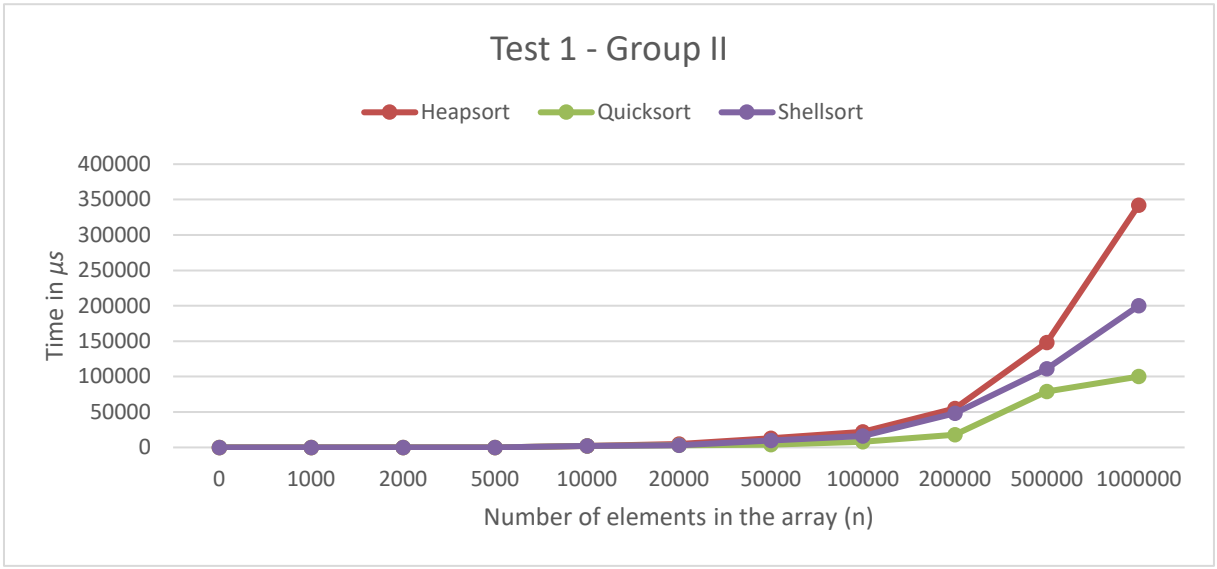
N	Heapsort	Quicksort	Shellsort
1000	<1	<1	<1
2000	<1	<1	<1
5000	1000	1000	<1
10000	2000	1000	<1
20000	3000	2000	1000
50000	7000	3000	2000
100000	19000	6000	5000
200000	48000	19000	12000
500000	100000	43000	30000
1000000	195000	67000	58000

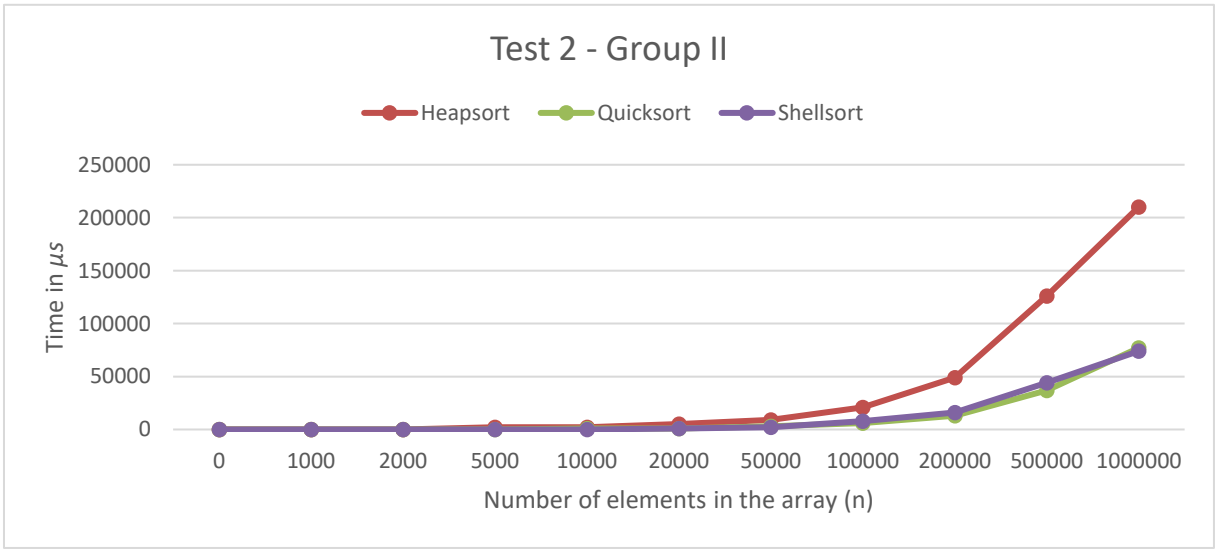
Charts

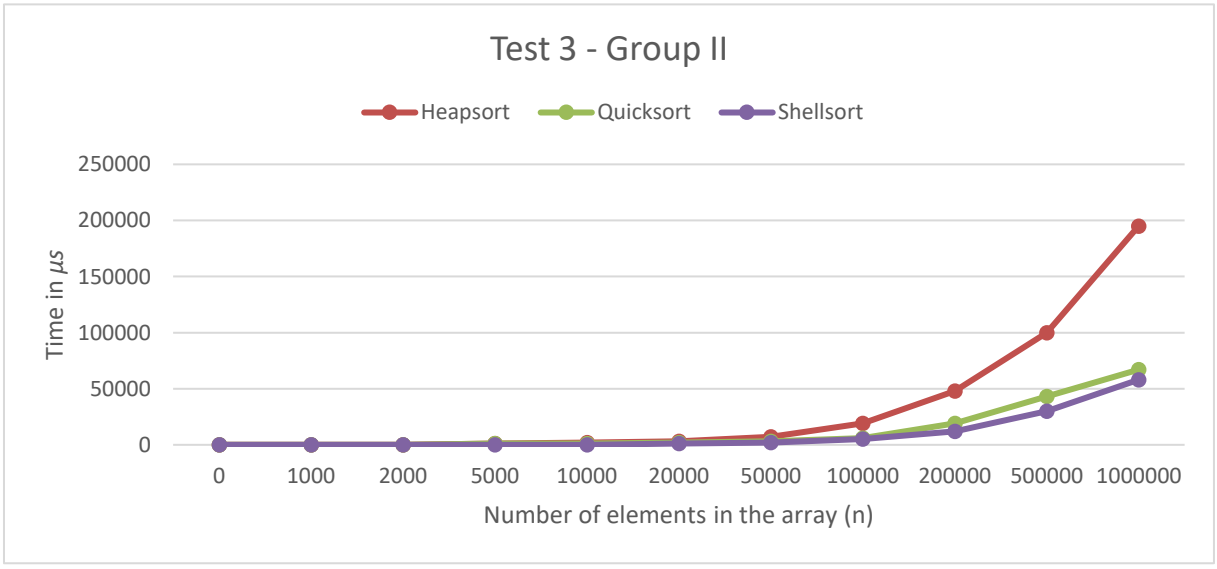

Test 1 - Group I
Bubble sort
Insertion sort
Selection sort
300000000
200000000
100000000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
Number of elements in the array (n)

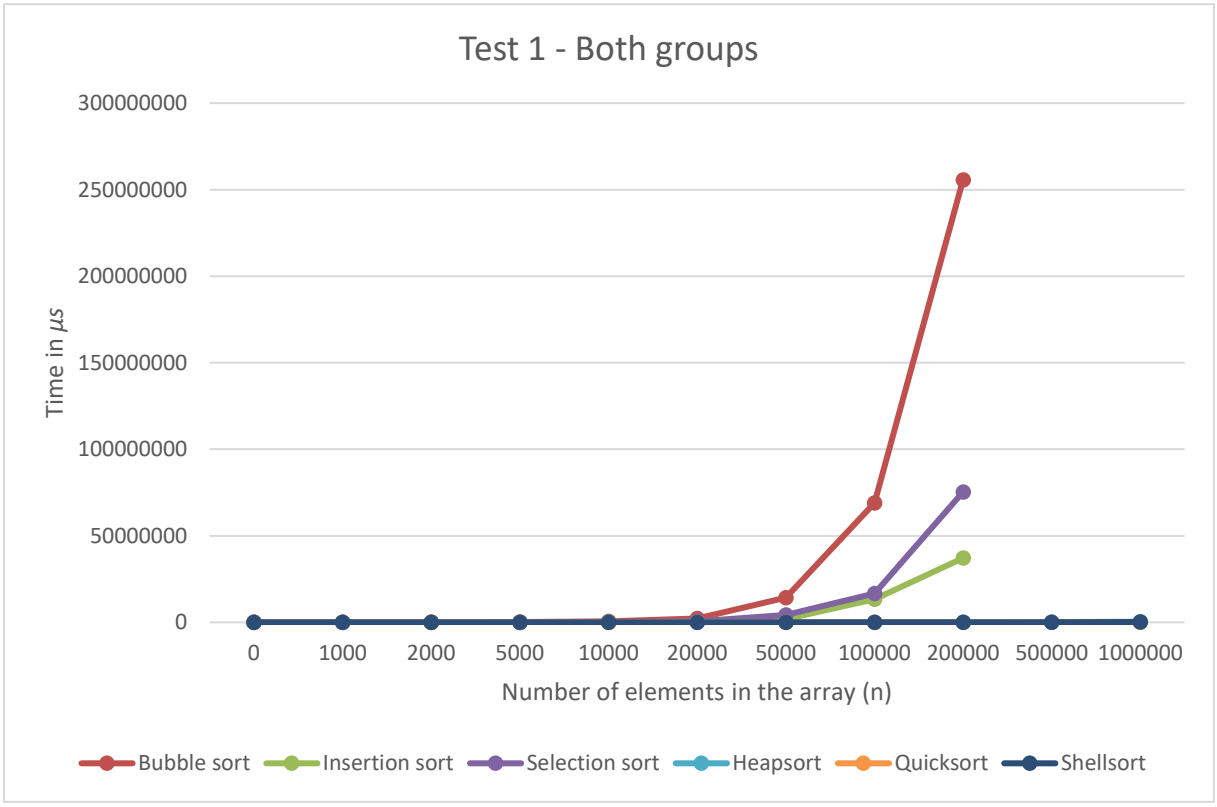

Test 2 - Group I
Bubble sort
Insertion sort
Selection sort
200000000
150000000
100000000
50000000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
Number of elements in the array (n)

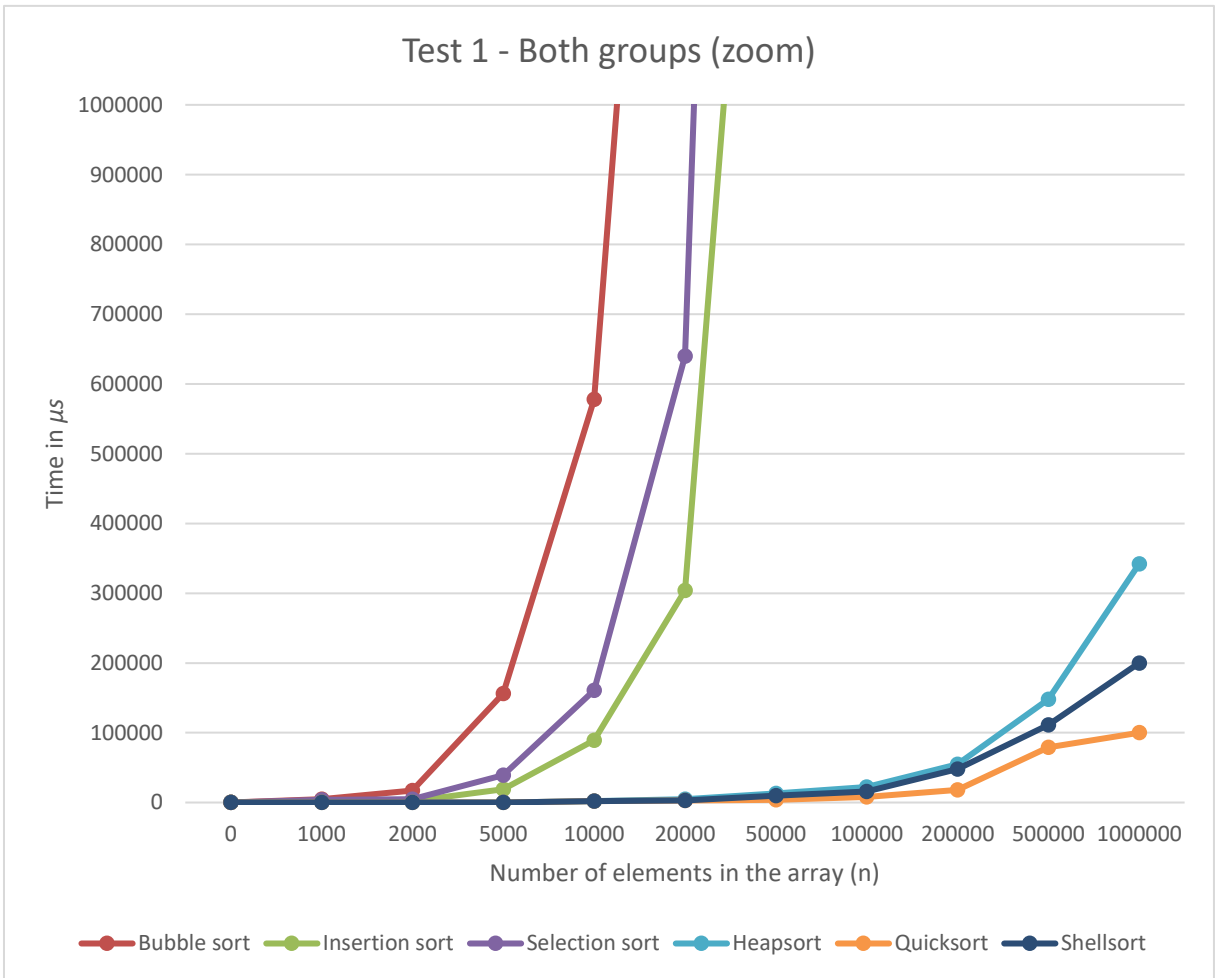

Test 3 - Group I
Bubble sort
Insertion sort
Selection sort
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
Number of elements in the array (n)

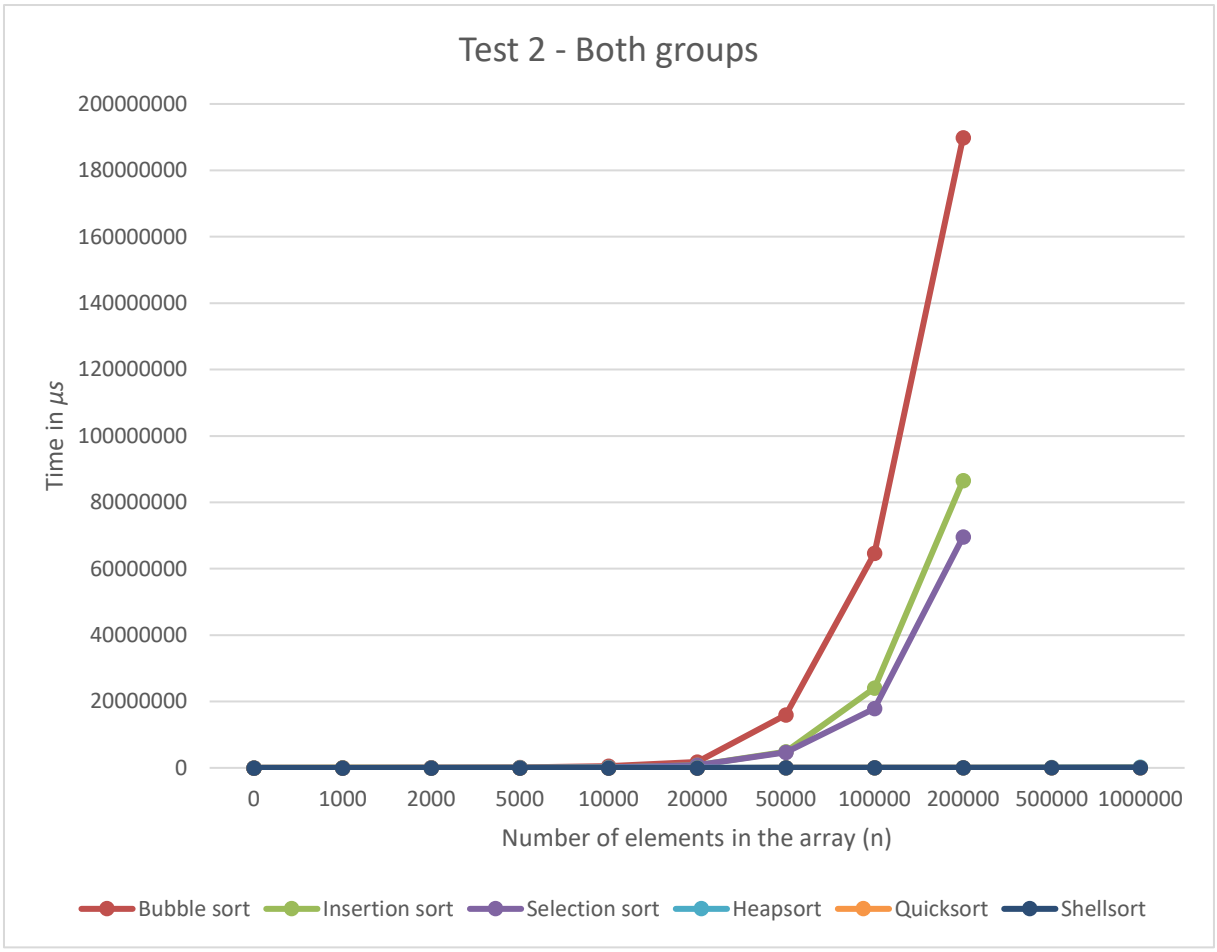

Test 1 - Group II
Heapsort
Quicksort
Shellsort
400000
350000
300000
250000
200000
150000
100000
50000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)

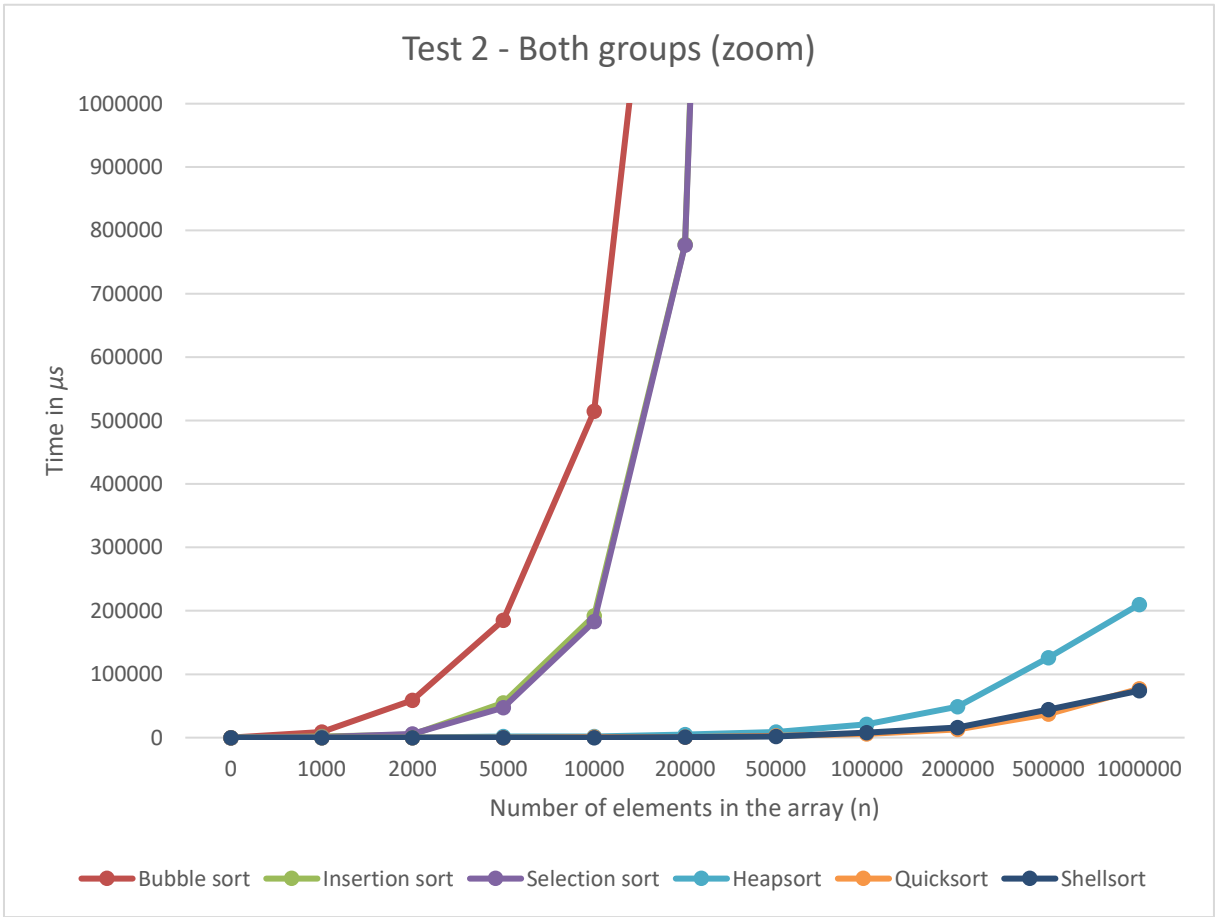

Test 2 - Group II
Heapsort
Quicksort
Shellsort
250000
200000
150000
100000
50000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)

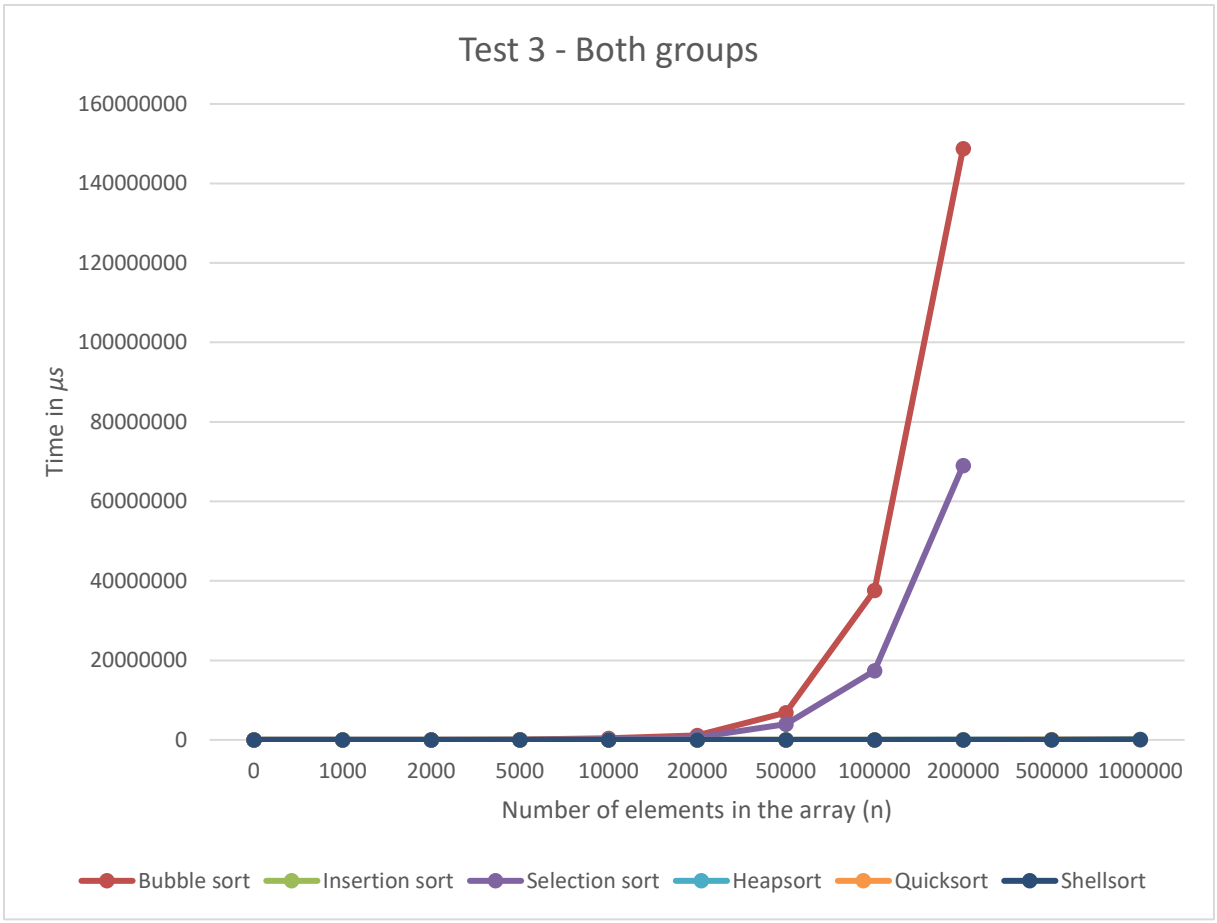

Test 3 - Group II
Heapsort
Quicksort
Shellsort
250000
200000
150000
100000
50000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)

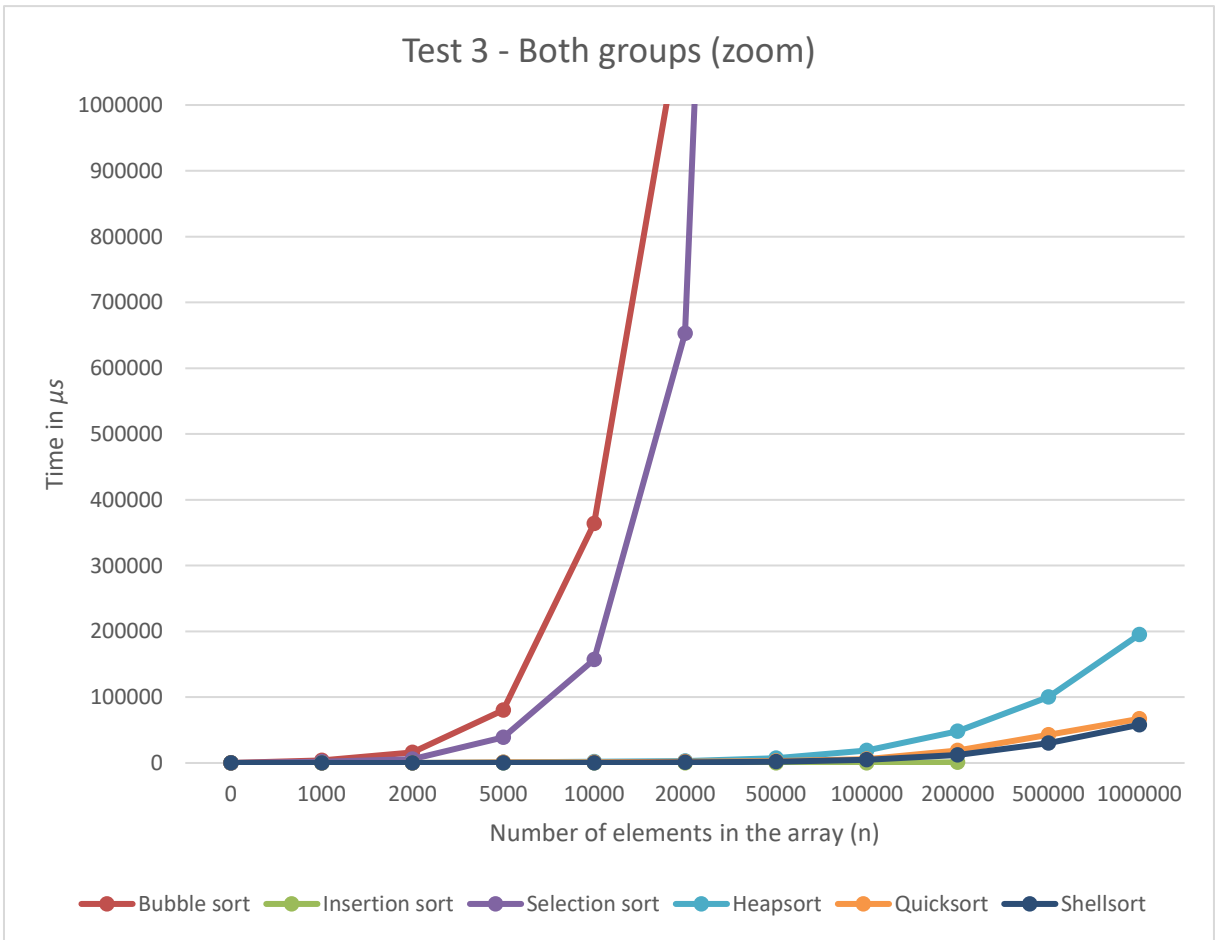

Test 1 - Both groups
300000000
250000000
200000000
150000000
100000000
50000000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)
Bubble sort
Insertion sort
Selection sort
Heapsort
Quicksort
Shellsort


Test 1 - Both groups (zoom)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)
Bubble sort
Insertion sort
Selection sort
Heapsort
Quicksort
Shellsort


Test 2 - Both groups
200000000
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)
Bubble sort
Insertion sort
Selection sort
Heapsort
Quicksort
Shellsort


Test 2 - Both groups (zoom)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Time in μs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)
Bubble sort
Insertion sort
Selection sort
Heapsort
Quicksort
Shellsort


Test 3 - Both groups
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
Time in µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)
Bubble sort
Insertion sort
Selection sort
Heapsort
Quicksort
Shellsort


Test 3 - Both groups (zoom)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Time in µs
0
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
Number of elements in the array (n)
Bubble sort
Insertion sort
Selection sort
Heapsort
Quicksort
Shellsort

Analysis of results and conclusions

Group I:

By the rate of growth of the graphs, we can easily conclude that these functions are of the form $An^2 + Bn + C$ -> their time complexity is $O(n^2)$. However, to prove this, let's analyze the sample results of these functions and compare to the growth of the class $O(n^2)$ to see if our assumption holds true:

Bubble sort:

For n = 1000: 5000 µs

For n = 2000: 17000 µs -> time increase of 3.4 times (slower $than$ $O(n^2)$)

For n = 5000: 156000 µs -> time increase of about 9.2 times (faster $than$ $O(n^2)$)

For n = 10000: 578000 µs -> time increase of about 3.7 times (slower $than$ $O(n^2)$)

Insertion sort:

For n = 1000: 1000 µs

For n = 2000: 2000 µs -> increase in time by 2 times (much slower $than\ O(n^2)$)

For n = 5000: 19000 µs -> time increase of about 9.5 times (faster $than\ O(n^2)$)

For n = 10000: 89000 µs -> time increase of about 4.7 times (faster $than\ O(n^2)$)

Selection sort:

For n = 1000: 4000 µs

For n = 2000: 5000 µs -> time increase of 1.25 times (much slower $than\ O(n^2)$)

For n = 5000: 39000 µs -> time increase of about 7.8 times (faster $than\ O(n^2)$)

For n = 10000: 161000 µs -> time increase of about 4.12 times (slightly faster $than\ O(n^2)$)

A certain regularity can be observed here, these algorithms perform faster or slower due to various factors (though the speed of the machine or the sorted state of the array). However, they more or less equalize to the level of $O(n^2)$, also note that we ignore constants in the class.

Therefore, it can be concluded that the **complexity class of Group I algorithms is $O(n^2)$** and it coincides with publicly available knowledge. This occurs for all algorithms, however, in the case of sorted data, **the time complexity class of the sort by insertion algorithm tends to $O(n)$.**

Group II:

It is not at all that easy to tell from the speed of growth of group 2 graphs by their class. However, we can see from the collective graphs that **they are much faster than Group I algorithms.** Based on common knowledge, Shell's algorithm has roughly the complexity of the $O(n^{\frac{3}{4}})$ and the rest of the algorithms $O(n * log\, n)$. However, the Quicksort algorithm is able to relegate itself to the class of $O(n^2)$ and Shell's algorithm at best to $O(n * log\, n)$ and $O(n^2)$ at worst. Here we have quite a big gap, but we can more or less compare them to known complexities. So let's try to roughly define the complexity classes of these algorithms.

Heapsort:

For n = 1000: 2000 µs

For n = 2000: 5000 µs -> time increase of 2.5 times (faster $than\, O(n * \log n)$)

For n = 5000: 13000 µs -> time increase of about 2.6 times (slightly faster $than\, O(n * \log n)$)

For n = 10000: 22000 µs -> time increase of about 1.7 times (slower $than\, O(n * \log n)$)

Shellsort:

For n = 1000: 2000 µs

For n = 2000: 3000 µs -> time increase of 1.5 times (slower $than$ $O(n^{\frac{3}{4}})$)

For n = 5000: 4000 µs -> time increase of about 1.3 times (slower $than$ $O(n^{\frac{3}{4}})$)

For n = 10000: 8000 µs -> time increase of about 2 times (faster $than$ $O(n^{\frac{3}{4}})$)

Quicksort:

For n = 1000: 2000 µs

For n = 2000: 3000 µs -> time increase of 1.5 times (slower $than$ $O(n * \log n)$)

For n = 5000: 10000 µs -> time increase of about 3.3 times (same as $O(n * \log n)$)

For n = 10000: 16000 µs -> time increase of about 1.6 times (slower $than$ $O(n * \log n)$)

As with the previous group, the speed of the times fluctuates. However, they more or less align with the complexity classes we are familiar with.

Therefore, it can be concluded that the **complexity class of Group II algorithms is $\boldsymbol{O(n * log\ n)}$** With the exception of Shell's algorithm, which is more or less $O(n^{\frac{3}{4}})$.

<u>Note, however, that these are only approximations. Everything depends on the environment and on the operations that introduce constants into the equation, and these are ignored by the complexity class. However, it gives us some insight and understanding of the complexity of sorting algorithms.</u>