Kaunas University of Technology
Faculty of Informatics

# Analysis of Sorting Algorithms Complexity. Selection Sort, Heap Sort, and Quick Sort

**Laboratory work report**
**Task variant No. 12**

Orkun MANAP

2021

# Task

Given three sorting methods: Selection Sort, Heap Sort, and Quick Sort. An analysis of the time complexity for each method and comparison of experimental results must be performed.

The time complexity analysis of each algorithm consists of these steps.

- Create and test the software for the implementation and visualization for each of given algorithms.
- Determine experimentally and plot the program execution time as a function of the number of items to be sorted.
- Establish theoretically and plot the relationship between the algorithm execution time (number of operations) and the number of elements under sorting.
- Compare the experimental and theoretical dependence plots of each sorting algorithm.

A comparison of the experimentally determined complexity of the three algorithms must be done in the same graphical window.

Python and PyGame library equipment are chosen for the implementation of algorithms.

# 1 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. The algorithm maintains two subarrays in each array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.[2]

## 1.1 Sorting and visualization

The script *selection.py*, provided in the Appendix 1, sorts the random number sequence [1, 10] in descending order using the selection sort algorithm.

You may select initial data by entering ‘E’ – random sorted initial set by entering ‘R’ and see all the iterations by pressing <Enter>. You can exit the program any time by clicking close button.

Process visualization is given in PyGame window (Figure 1). The figure window consists columns for each variable.

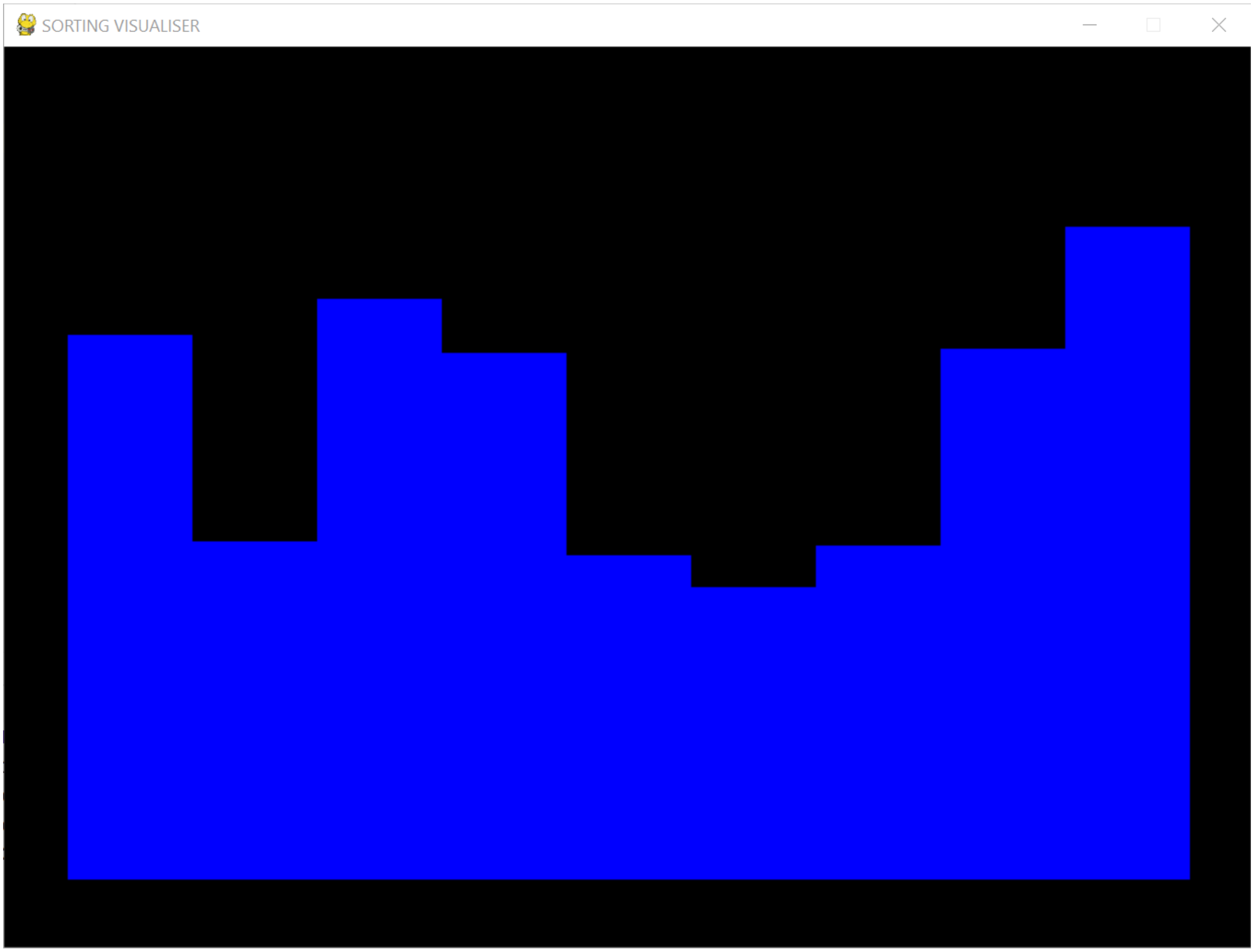

Figure 1. 1.1 Selection sort visualization. 1st step

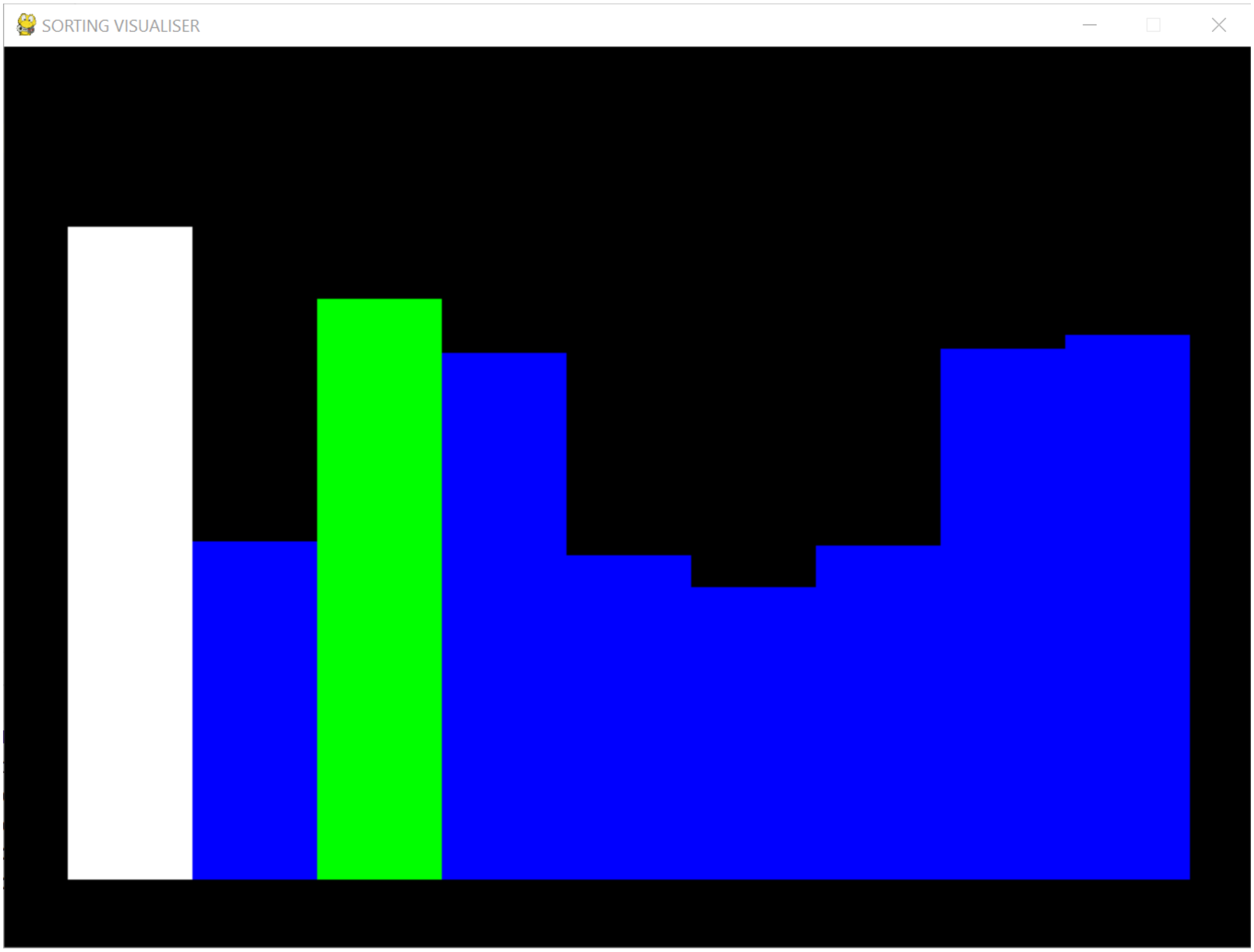

Figure 1.1.2 Selection sort visualization. 2nd step

Finally, we can see the sorted sequence:


Figure 1.1.3 Selection sort visualization. The last step

The initial unsorted array, all sorting algorithm iterations and sorted array are presented in the run window.

```
Press R to create random elements and press E to enter 10 numbers: R
0 [0, 208, 357, 182, 221, 367, 390, 360, 218, 130]
1 [0, 208, 357, 182, 221, 367, 390, 360, 218, 130]
2 [0, 130, 357, 182, 221, 367, 390, 360, 218, 208]
3 [0, 130, 182, 357, 221, 367, 390, 360, 218, 208]
4 [0, 130, 182, 208, 221, 367, 390, 360, 218, 357]
5 [0, 130, 182, 208, 218, 367, 390, 360, 221, 357]
6 [0, 130, 182, 208, 218, 221, 390, 360, 367, 357]
7 [0, 130, 182, 208, 218, 221, 357, 360, 367, 390]
8 [0, 130, 182, 208, 218, 221, 357, 360, 367, 390]
9 [0, 130, 182, 208, 218, 221, 357, 360, 367, 390]
```

Figure 1.1.4 Selection sorting results

## 1.2 Determination of time dependencies

The running time of the algorithm was experimentally defined for the input sequences under sorting containing 500, 1000, …, 32000 elements.

The script *test_selection.py* and function *selection_sort(arr, element_number)* are provided in the Appendix 2.

The results of 10 experiments and mean time values are presented in command window (Figure 4).

| 500 | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 |
|---|---|---|---|---|---|---|
| 0.009345054626464844 | 0.04595017433166504 | 0.14874649047851562 | 0.5988917350769043 | 2.4668073654174805 | 9.940297842025757 | 40.903899908065796 |
| 0.009037256240844727 | 0.037070512771606445 | 0.15816569328308105 | 0.6073997020721436 | 2.411123752593994 | 9.866627216339111 | 39.917436599731445 |
| 0.009129762649536133 | 0.03748035430908203 | 0.15014410018920898 | 0.6170897483825684 | 2.484412670135498 | 9.647598505020142 | 41.371678829193115 |
| 0.008973121643066406 | 0.0439000129699707 | 0.1507282257080078 | 0.6172254085540771 | 2.4302265644073486 | 10.010490894317627 | 40.67262601852417 |
| 0.00913858413696289 | 0.03726458549499512 | 0.15199804306030273 | 0.6060643196105957 | 2.4528026580810547 | 9.90371036529541 | 40.4600133895874 |
| 0.009031057357788086 | 0.03760647773742676 | 0.14894700050354004 | 0.6106529235839844 | 2.4077467918395996 | 9.981728792190552 | 40.540104389190674 |
| 0.009449958801269531 | 0.03812217712402344 | 0.15613174438476562 | 0.6023504734039307 | 2.429201602935791 | 9.676339387893677 | 40.26688528060913 |
| 0.00899505615234375 | 0.036859989166259766 | 0.1543269157409668 | 0.5866641998291016 | 2.455174684524536 | 9.640969276428223 | 40.163856744766235 |
| 0.00918126106262207 | 0.0368502140045166 | 0.15008926391601562 | 0.5969908237457275 | 2.437690496444702 | 9.753377437591553 | 40.121235847473145 |
| 0.008951663970947266 | 0.04407048225402832 | 0.14876270294189453 | 0.6062183380126953 | 2.4145045280456543 | 9.94438648223877 | 39.143996477127075 |

```
Mean values selection sort:  [0.0010977, 0.0045378, 0.0179813, 0.0718981, 0.2884485, 1.1593934, 4.5988411]
```

Figure 1.2.1 The program execution times under sorting n elements

The dependence of time on the number of elements to be sorted is given in the graph window (Figure 5).

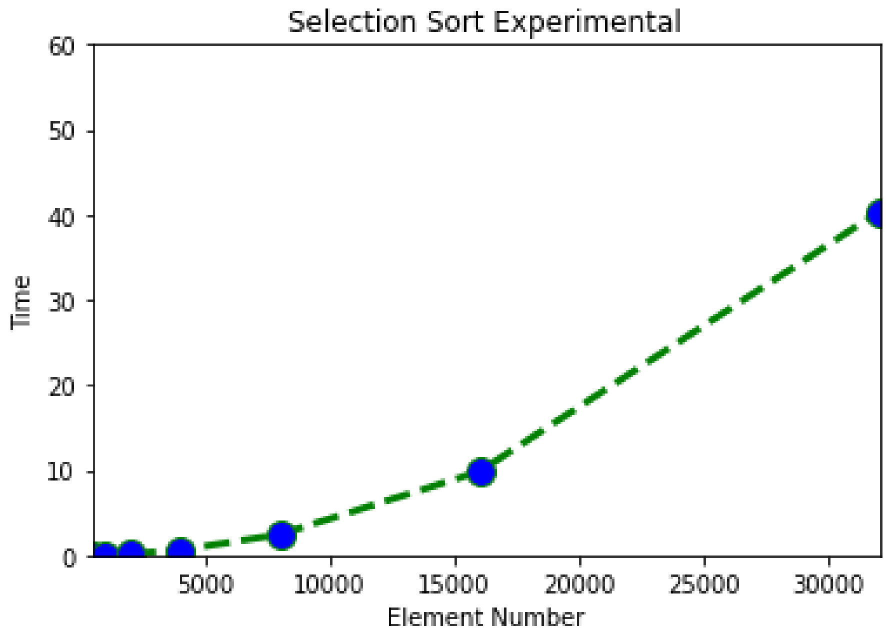


Figure 1.2.2 Experimental time dependence

Time complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc. Time complexity of selection sort is given in the table (Table 1).

Table 1.1 Selection sort time complexity [2]

| **Algorithm** | **Time complexity** | | |
|---|---|---|---|
| | **Best** | **Average** | **Worst** |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |

The number of operations dependences on the number of sorted elements for average and worst cases is given graphically (Figure 6).

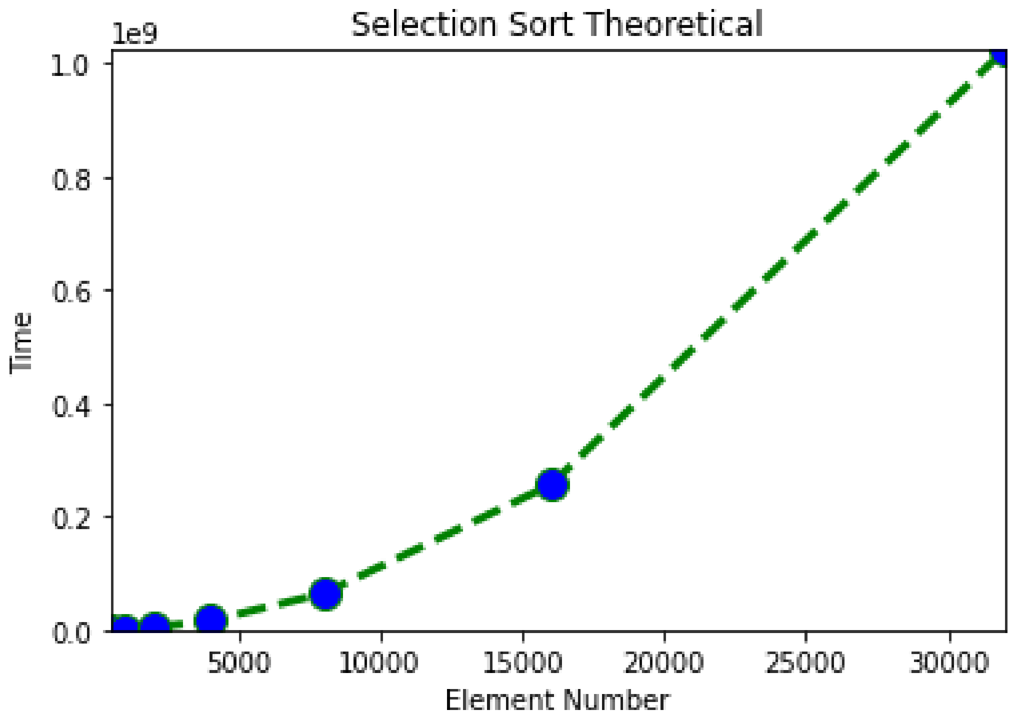


Figure 1.2.3 Selection sort time complexity

We can see that the experimental and theoretical dependencies (Figure 1.2.2, Figure 1.2.3)

# 2 Quick sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways. [1]

## 2.1 Sorting and visualization

The script quick.*py*, provided in the Appendix 3, sorts the random number sequence [1, 10] in ascending order using the selection sort algorithm.

You may select initial data by entering 'E' – random sorted initial set by entering 'R' and see all the iterations by pressing <Enter>. You can exit the program any time by clicking close button.

Process visualization is given in PyGame window (Figure 1). The figure window consists columns for each variable.

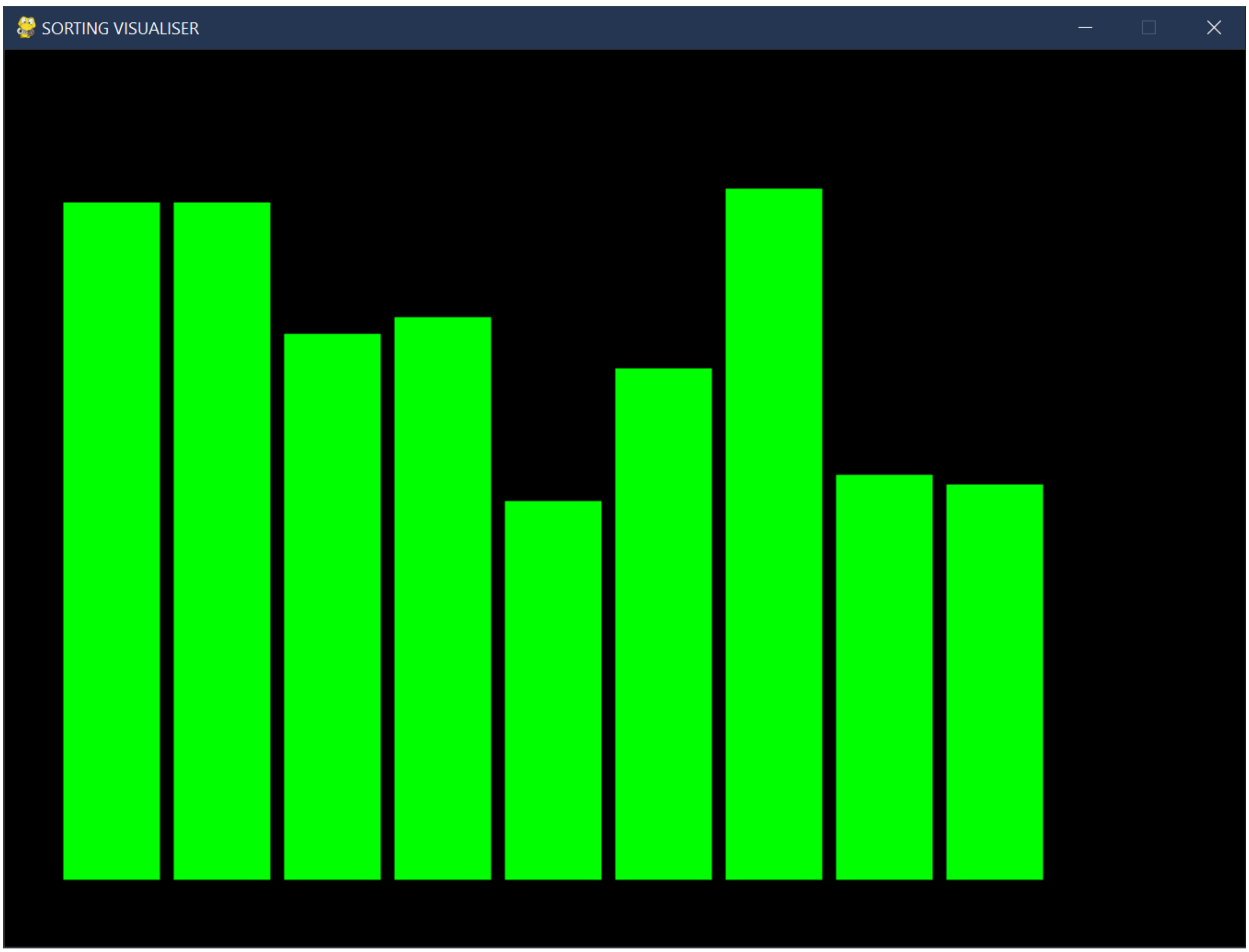


Figure 2.1.1 Selection sort visualization. 1st step

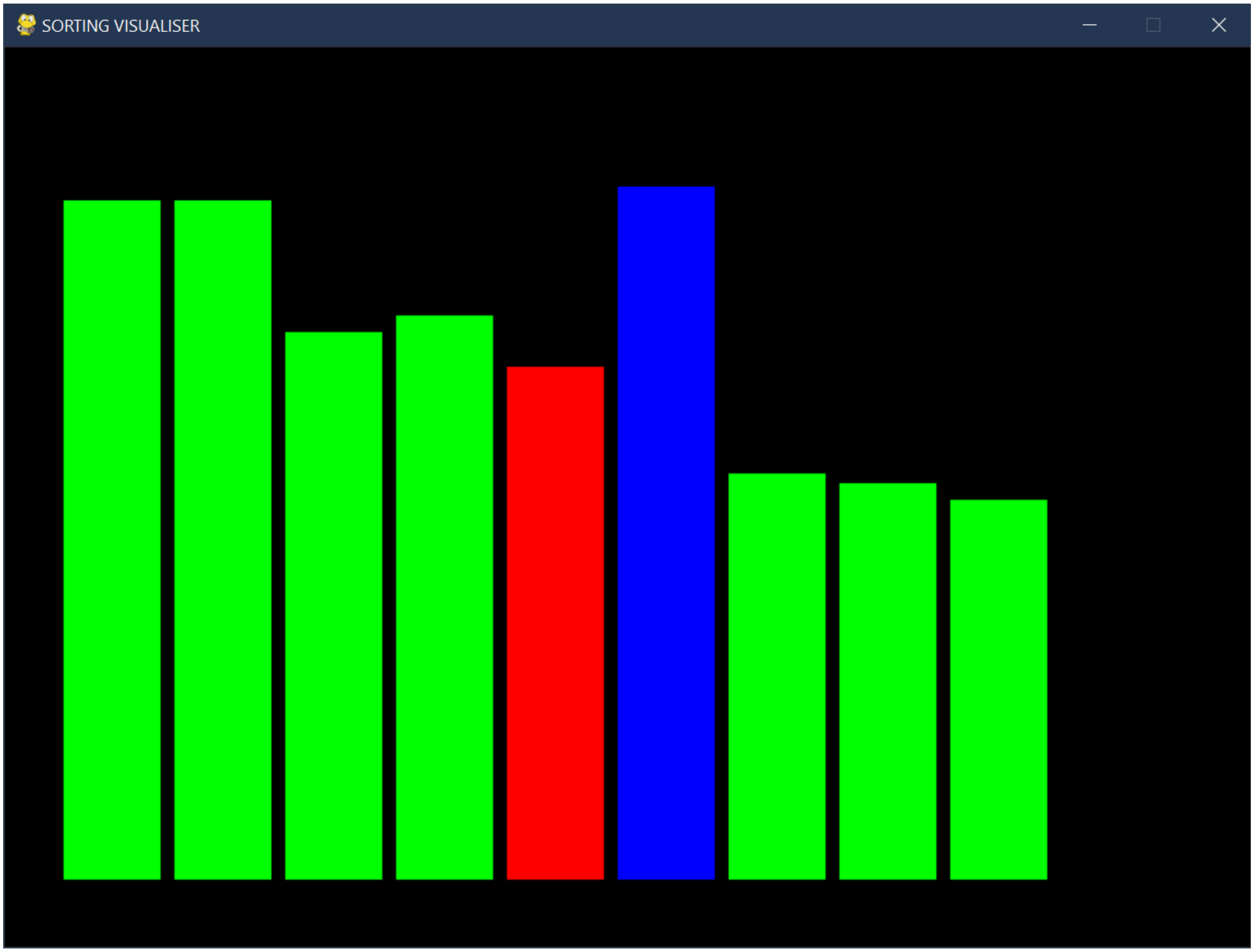


Figure 2.1.2 Selection sort visualization. 2nd step

Finally, we can see the sorted sequence:

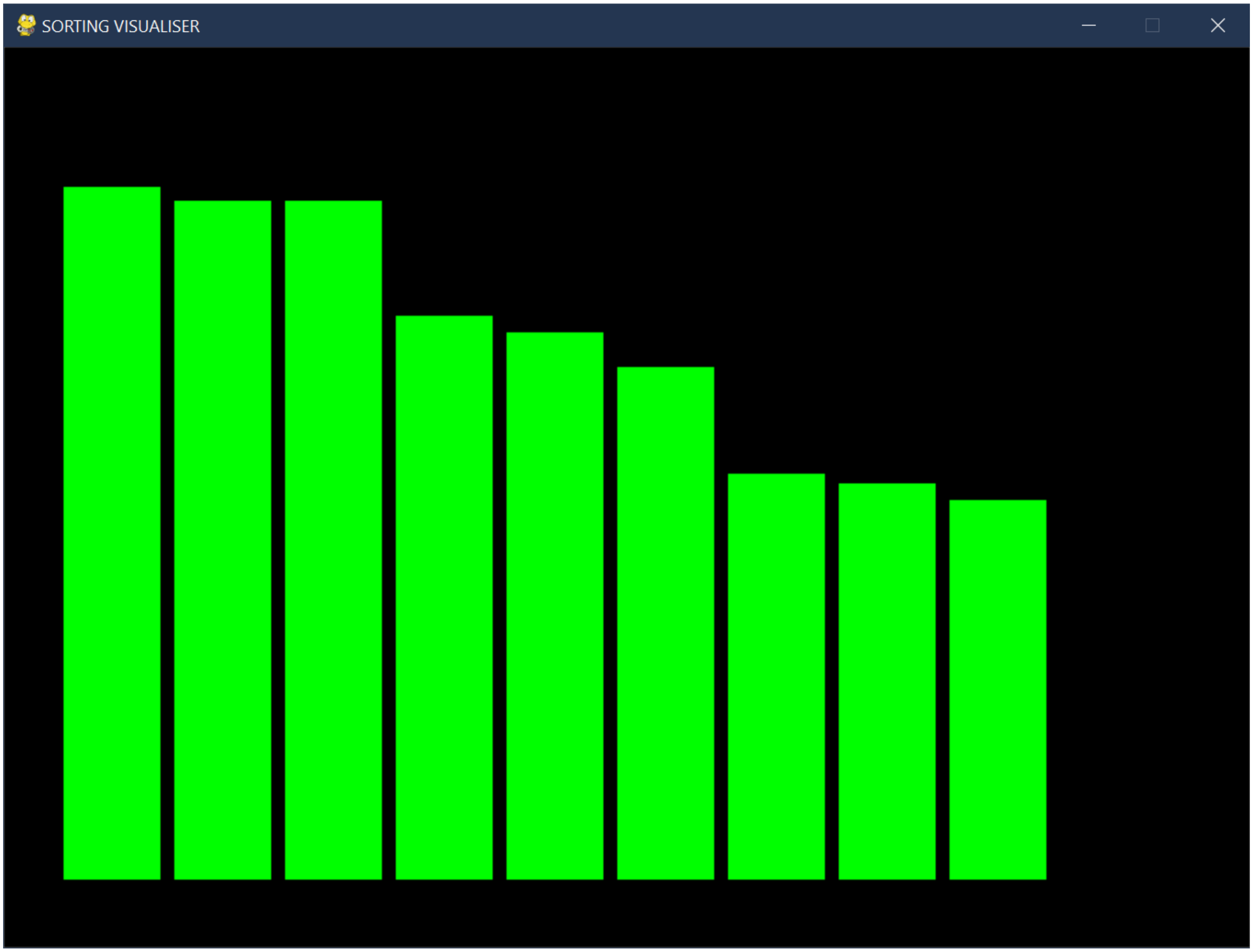


Figure 2.1.3 Selection sort visualization. The last step

The initial unsorted array, all sorting algorithm iterations and sorted array are presented in the run window.

```
[0, 116, 171, 214, 298, 171, 348, 355, 232, 308]
[0, 116, 171, 214, 298, 171, 232, 308, 348, 355]
[0, 116, 171, 214, 298, 171, 232, 308, 348, 355]
[0, 116, 171, 214, 298, 171, 232, 308, 348, 355]
[0, 116, 171, 214, 298, 171, 232, 308, 348, 355]
[0, 116, 171, 214, 171, 232, 298, 308, 348, 355]
[0, 116, 171, 214, 171, 232, 298, 308, 348, 355]
[0, 116, 171, 171, 214, 232, 298, 308, 348, 355]
[0, 116, 171, 171, 214, 232, 298, 308, 348, 355]
```

Figure 2.1.4 Selection sorting results

## 2.2 Determination of time dependencies

The running time of the algorithm was experimentally defined for the input sequences under sorting containing 500, 1000, …, 32000 elements.

The script *test_quicksort.py* and function *quick_sort(arr, l, h)* are provided in the Appendix 4.

The results of 10 experiments and mean time values are presented in command window (Figure 2.2.1).

| 500 | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 |
|---|---|---|---|---|---|---|
| 0.003597259521484375 | 0.002637624740600586 | 0.0070378780364990234 | 0.01915597915649414 | 0.04935812950134277 | 0.1439647674560547 | 0.5078227519989014 |
| 0.0013837814331054688 | 0.002838611602783203 | 0.006513833999633789 | 0.016820192337036133 | 0.05126953125 | 0.14356470108032227 | 0.4946327209472656 |
| 0.0012006759643554688 | 0.002615213394165039 | 0.006696224212646484 | 0.016656160354614258 | 0.05066847801208496 | 0.14736676216125488 | 0.5104091167449951 |
| 0.0011105537414550781 | 0.002790212631225586 | 0.006421327590942383 | 0.019308090209960938 | 0.049275875091552734 | 0.10871577262878418 | 0.4291718006134033 |
| 0.0008225440979003906 | 0.002094268798828125 | 0.005594491958618164 | 0.01644110679626465 | 0.047403812408447266 | 0.1420912742614746 | 0.516512155532836 9 |
| 0.0010704994201660156 | 0.0026226043701171875 | 0.006305694580078125 | 0.017221689224243164 | 0.048888206481933594 | 0.14417481422424316 | 0.4276313781738281 |
| 0.0008220672607421875 | 0.0018672943115234375 | 0.004717826843261719 | 0.011823177337646484 | 0.032120704650878906 | 0.10341334342956543 | 0.4969499111175537 |
| 0.0011112689971923828 | 0.0026237964630126953 | 0.00698542594909668 | 0.01687455177307129 | 0.0510101318359375 | 0.14771389961242676 | 0.49768972396850586 |
| 0.0011305809020996094 | 0.0025854110717773438 | 0.006933927536010742 | 0.016324996948242188 | 0.048233747482299805 | 0.1457991600036621 | 0.49401330947875977 |
| 0.0011856555938720703 | 0.0027022361755371094 | 0.007462739944458008 | 0.01673817634582519 5 | 0.04925370216369629 | 0.14734792709350586 | 0.49597692489624023 |

```
Mean values quick sort:  [0.0001098, 0.0004538, 0.0017981, 0.0071898, 0.0288448, 0.1159393, 0.4598841]
```

Figure 2.2.1 The program execution times under sorting n elements

The dependence of time on the number of elements to be sorted is given in the graph window (Figure 5).

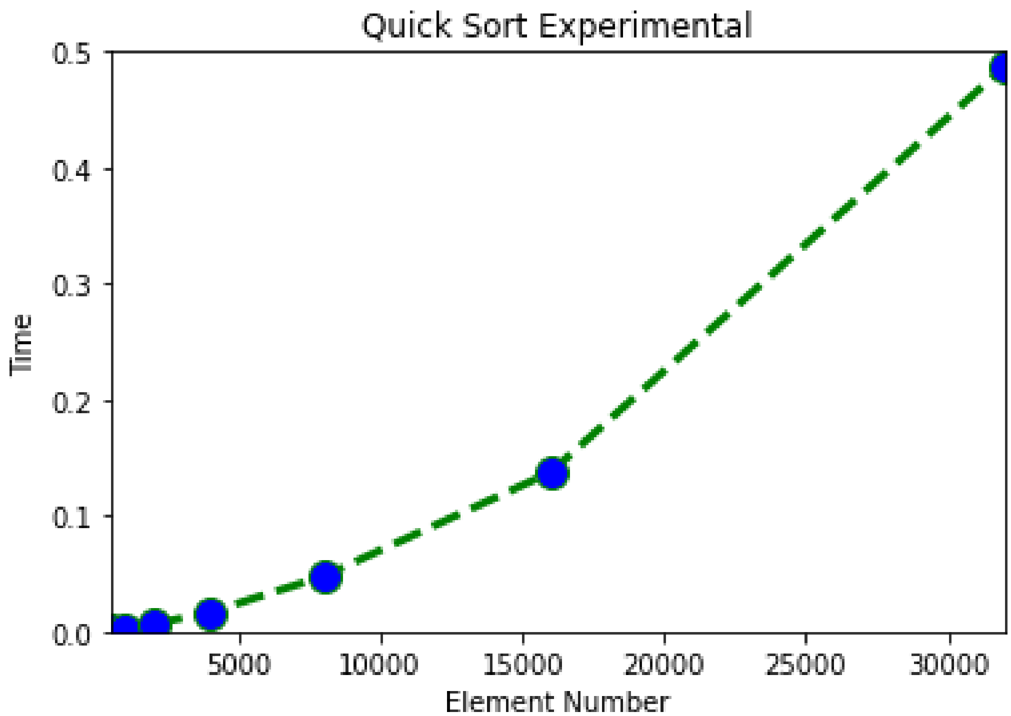


Figure 2.2.2 Experimental time dependence

Time complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc. Time complexity of quick sort is given in the table (Table 1).

Table 1. Quick sort time complexity [1]

| **Algorithm** | **Time complexity** | | |
|---|---|---|---|
| | **Best** | **Average** | **Worst** |
| Quick sort | O(n log n) | O(*n* log *n*) | $O(n^2)$ |

The number of operations dependences on the number of sorted elements for average and worst cases is given graphically (Figure Figure 2.2.2).

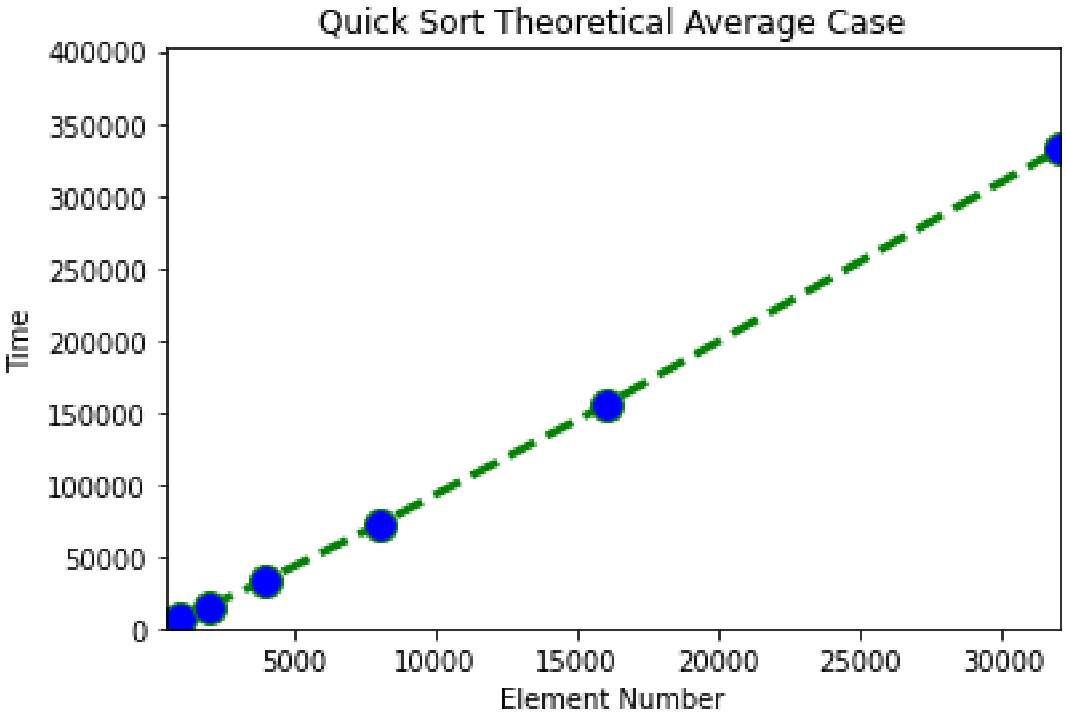

Figure 2.2.3 Quick sort average time complexity

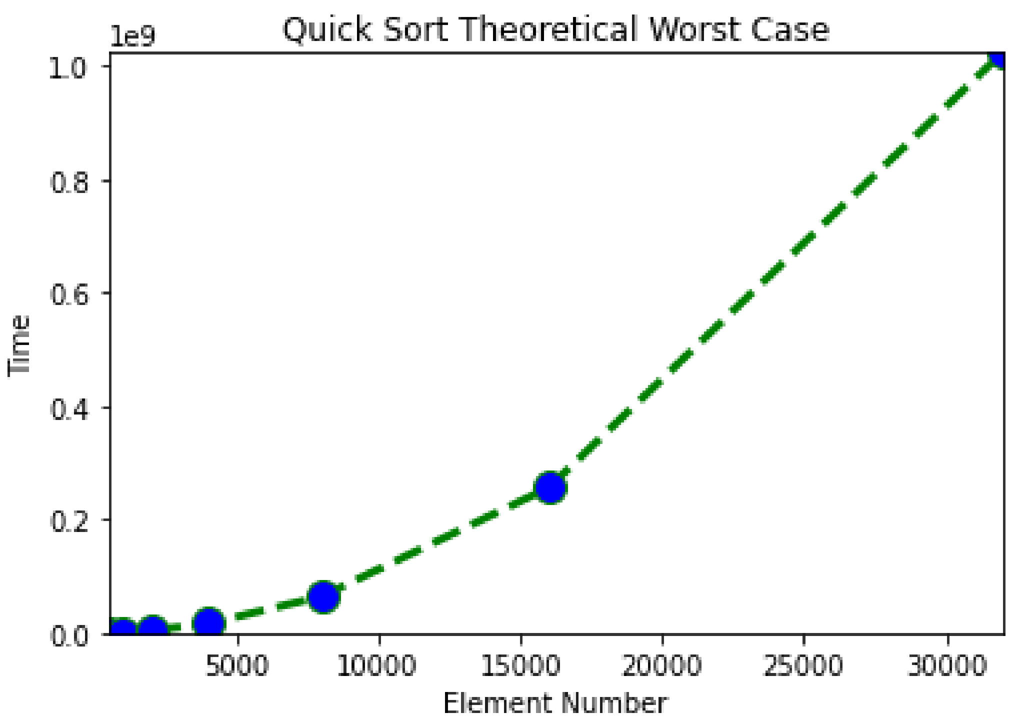


Figure 2.2.4 Quick sort worst time complexity

We can see that the experimental and theoretical dependencies (Figure 2.2.2, Figure 2.2.3, Figure 2.2.4)

# 3 Heap sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements. [3]

## 3.1 Sorting and visualization

The script heap.*py*, provided in the Appendix 5, sorts the random number sequence [1, 10] in ascending order using the selection sort algorithm.

You may select initial data by entering 'E' – random sorted initial set by entering 'R' and see all the iterations by pressing <Enter>. You can exit the program any time by clicking close button.

Process visualization is given in PyGame window (Figure 1). The figure window consists columns for each variable.

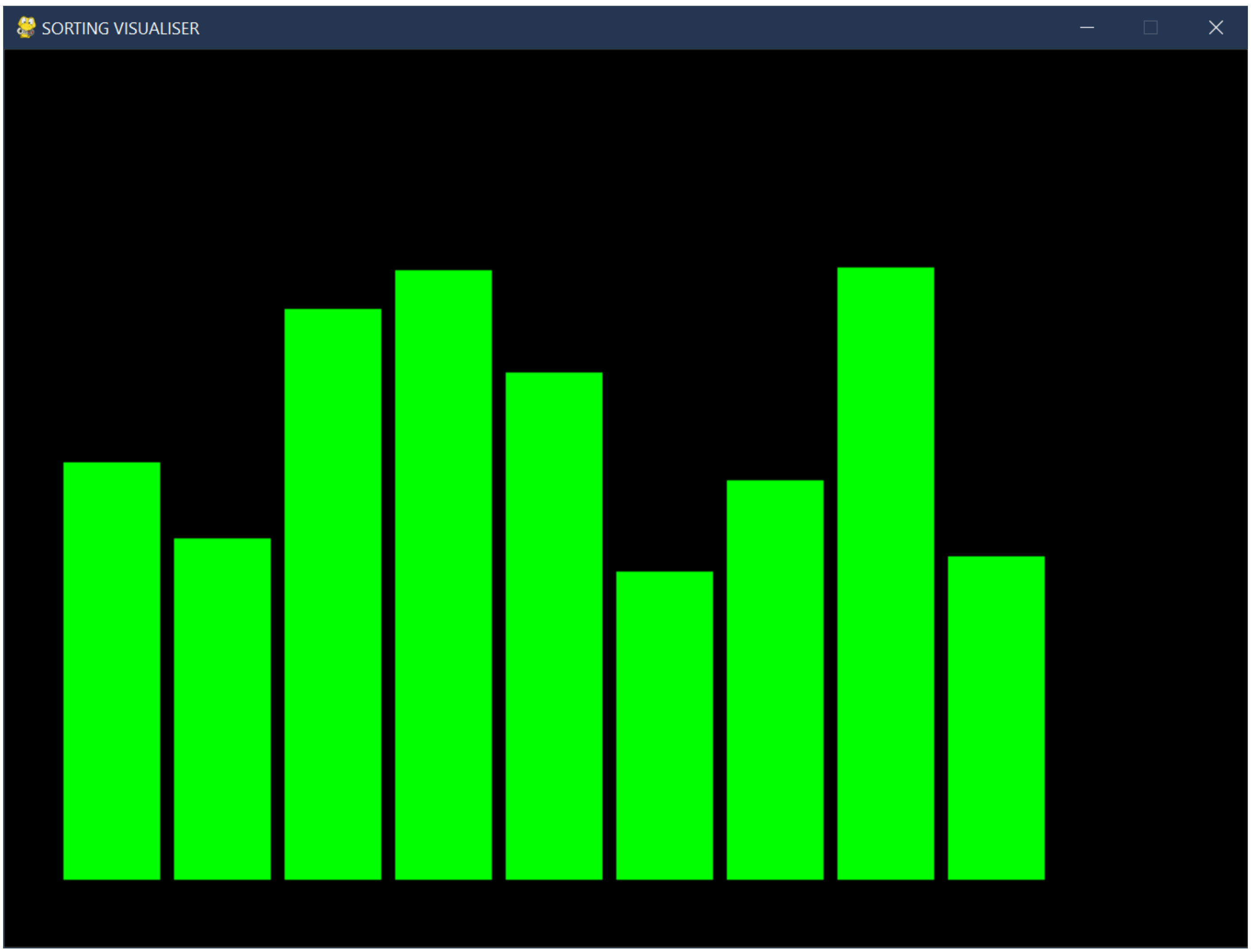


Figure 3.1.1 Heap sort visualization. 1st step

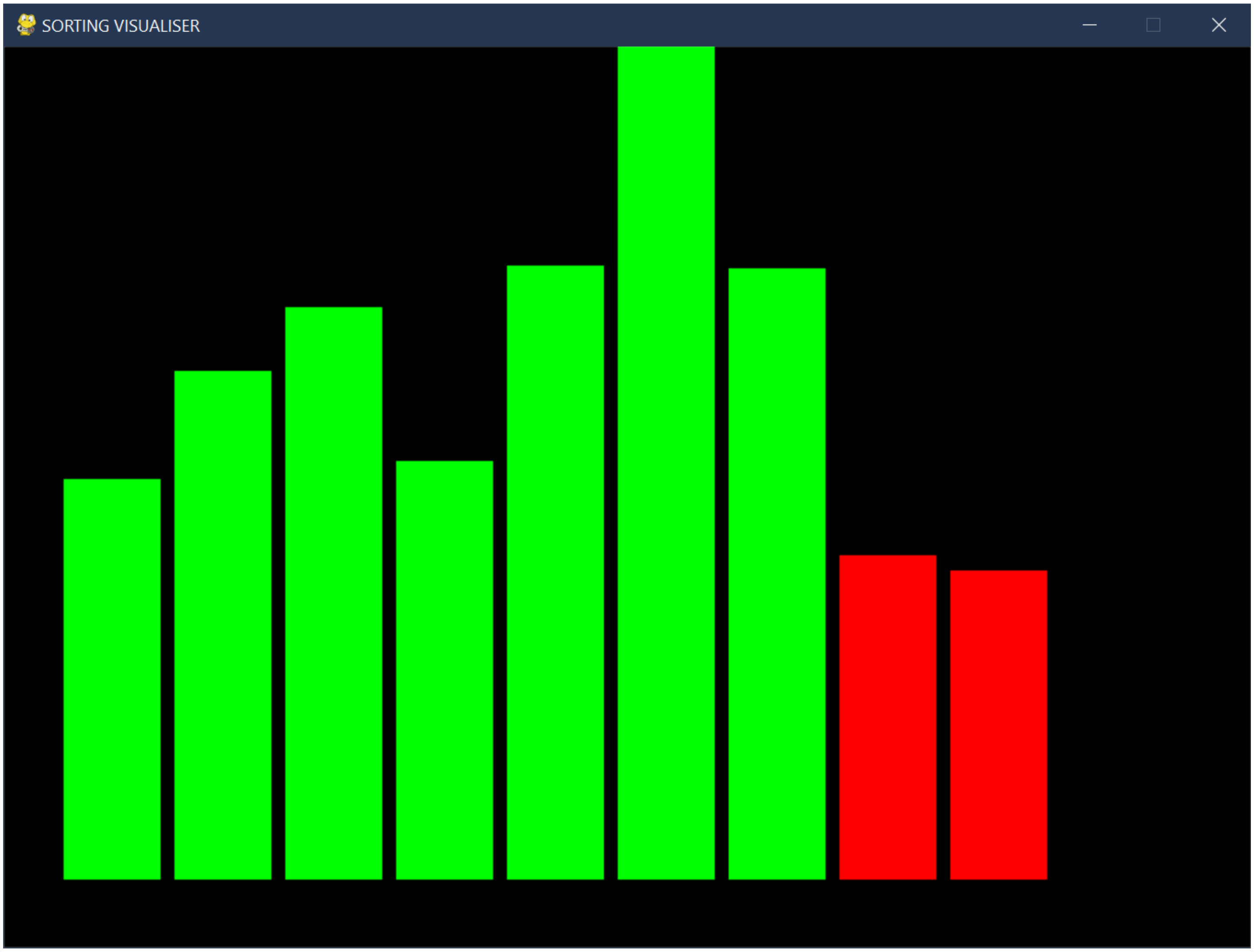

Figure 3.1.2 Heap sort visualization. 2nd step

Finally, we can see the sorted sequence:

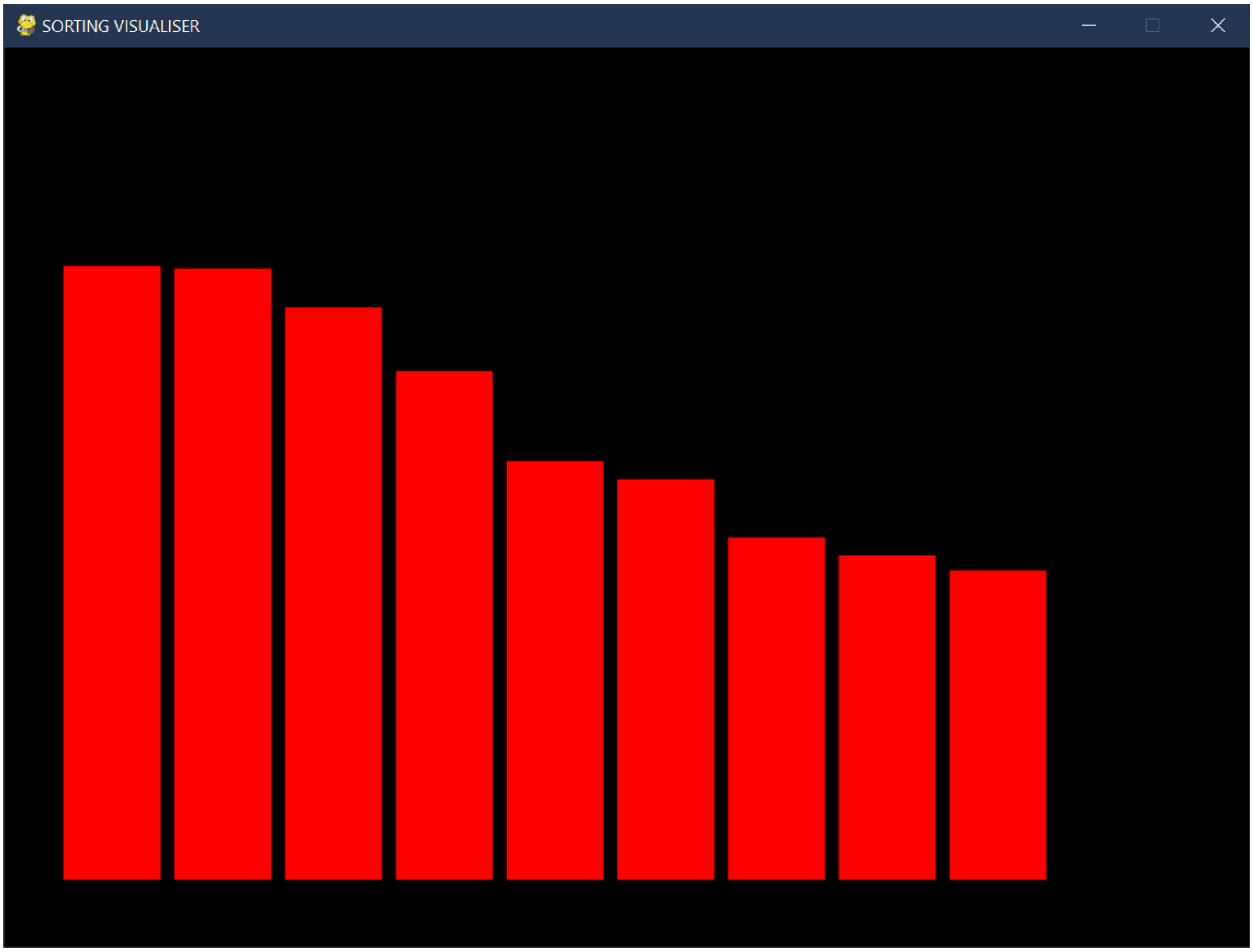


Figure 3.1.3 Heap sort visualization. The last step

The initial unsorted array, all sorting algorithm iterations and sorted array are presented in the run window.

```
[0, 299, 354, 188, 160, 234, 378, 312, 158, 367]
[0, 299, 354, 188, 367, 234, 378, 312, 158, 160]
[0, 299, 354, 188, 367, 234, 378, 312, 158, 160]
[0, 299, 354, 312, 367, 234, 378, 188, 158, 160]
[0, 299, 354, 312, 367, 234, 378, 188, 158, 160]
[0, 299, 378, 312, 367, 234, 354, 188, 158, 160]
[0, 299, 378, 312, 367, 234, 354, 188, 158, 160]
[0, 367, 378, 312, 299, 234, 354, 188, 158, 160]
[0, 367, 378, 312, 299, 234, 354, 188, 158, 160]
[378, 367, 0, 312, 299, 234, 354, 188, 158, 160]
[378, 367, 354, 312, 299, 234, 0, 188, 158, 160]
[160, 367, 354, 312, 299, 234, 0, 188, 158, 378]
[367, 160, 354, 312, 299, 234, 0, 188, 158, 378]
[367, 312, 354, 160, 299, 234, 0, 188, 158, 378]
[367, 312, 354, 188, 299, 234, 0, 160, 158, 378]
[158, 312, 354, 188, 299, 234, 0, 160, 367, 378]
[354, 312, 158, 188, 299, 234, 0, 160, 367, 378]
[354, 312, 234, 188, 299, 158, 0, 160, 367, 378]
[160, 312, 234, 188, 299, 158, 0, 354, 367, 378]
[312, 160, 234, 188, 299, 158, 0, 354, 367, 378]
[312, 299, 234, 188, 160, 158, 0, 354, 367, 378]
[0, 299, 234, 188, 160, 158, 312, 354, 367, 378]
[299, 0, 234, 188, 160, 158, 312, 354, 367, 378]
[299, 188, 234, 0, 160, 158, 312, 354, 367, 378]
[158, 188, 234, 0, 160, 299, 312, 354, 367, 378]
[234, 188, 158, 0, 160, 299, 312, 354, 367, 378]
[160, 188, 158, 0, 234, 299, 312, 354, 367, 378]
[188, 160, 158, 0, 234, 299, 312, 354, 367, 378]
[0, 160, 158, 188, 234, 299, 312, 354, 367, 378]
[160, 0, 158, 188, 234, 299, 312, 354, 367, 378]
[158, 0, 160, 188, 234, 299, 312, 354, 367, 378]
[0, 158, 160, 188, 234, 299, 312, 354, 367, 378]
```

Figure 3.1.4 Heap sorting results

## 3.2 Determination of time dependencies

The running time of the algorithm was experimentally defined for the input sequences under sorting containing 500, 1000, …, 32000 elements.

The script *test_* Heap*.py* and function Heap*_sort(arr, root, size)* are provided in the Appendix 6.

The results of 10 experiments and mean time values are presented in command window (Figure 3.2.1).

| 500 | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 |
|---|---|---|---|---|---|---|
| 0.002338409423828125 | 0.004712820053100586 | 0.011251449584960938 | 0.02330780029296875 | 0.05552935600280762 | 0.1134941577911377 | 0.24627208709716797 |
| 0.001984119415283203 | 0.004503488540649414 | 0.010151386260986328 | 0.02267146110534668 | 0.04928421974182129 | 0.11096787452697754 | 0.23343443870544434 |
| 0.0019183158874511719 | 0.004343986511230469 | 0.009921550750732422 | 0.022973060607910156 | 0.048294782638549805 | 0.10708379745483398 | 0.23292016983032227 |
| 0.0019092559814453125 | 0.004369497299194336 | 0.009808540344238281 | 0.022284984588623047 | 0.04907035827636719 | 0.11217570304870605 | 0.23640131950378418 |
| 0.0019216537475585938 | 0.004375457763671875 | 0.010329484939575195 | 0.021850824356079 1 | 0.04917740821838379 | 0.10750126838684082 | 0.23468661308288574 |
| 0.0019001960754394531 | 0.004415750503540039 | 0.010116338729858398 | 0.022077560424804688 | 0.04829907417297363 | 0.11283397674560547 | 0.23293352127075195 |
| 0.001924753189086914 | 0.004365682601928711 | 0.009790420532226562 | 0.02215719223022461 | 0.053632259368896484 | 0.10660791397094727 | 0.23740673065185547 |
| 0.00194549560546875 | 0.004420280456542969 | 0.00980377197265625 | 0.022035598754882812 | 0.05502057075500488 | 0.10882329940795898 | 0.23281645774841309 |
| 0.0019168853759765625 | 0.004431962966918945 | 0.010500431060791016 | 0.021817922592163086 | 0.04905557632446289 | 0.10733485221862793 | 0.23501014709472656 |
| 0.0019352436065673828 | 0.004378318786621094 | 0.009841680526733398 | 0.026933670043945312 | 0.04851675033569336 | 0.10688948631286621 | 0.23488640785217285 |

Mean values heap sort: [0.0001098, 0.0004538, 0.0017981, 0.0071898, 0.0288448, 0.1159393, 0.4598841]

Figure 3.2.1 The program execution times under sorting n elements

The dependence of time on the number of elements to be sorted is given in the graph window (Figure 5).

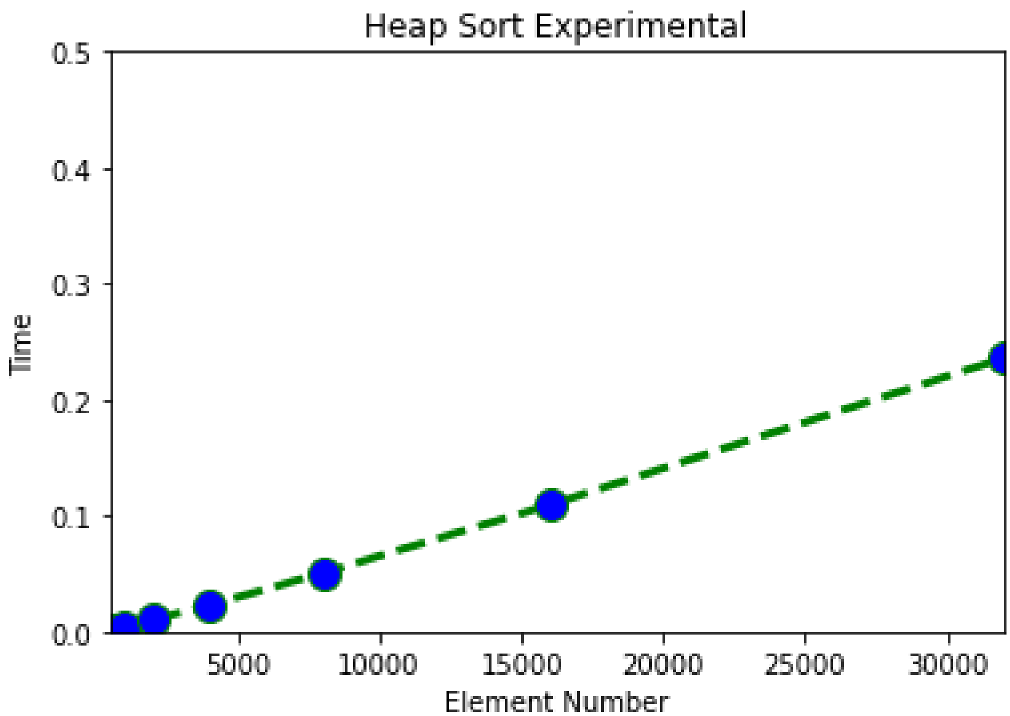


Figure 3.2.2 Experimental time dependence

Time complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc. [3] Time complexity of bubble sort is given in the table (Table 1).

Table 1. Heap sort time complexity

| **Algorithm** | **Time complexity** | | |
|---|---|---|---|
| | **Best** | **Average** | **Worst** |
| Bubble sort | O(n log n) | O(n log n) | O(n log n) |

The number of operations dependences on the number of sorted elements for average and worst cases is given graphically (Figure 3.2.2).

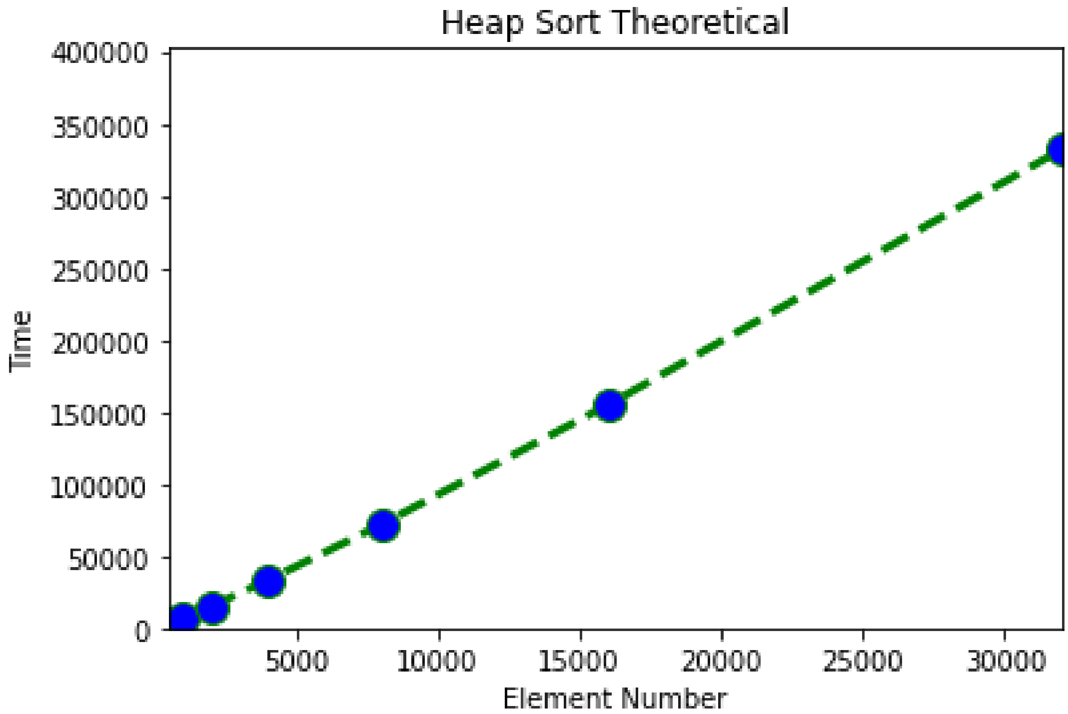


Figure 3.2.3 Heap sort time complexity

We can see that the experimental and theoretical dependencies (Figure 3.2.1, 3.2.2)

## 4 Comparative analysis

The script that allows us to visually compare the experimental execution times of the three methods is presented in Appendix 7. We can see comparison results in figure window (Figure 4.1).

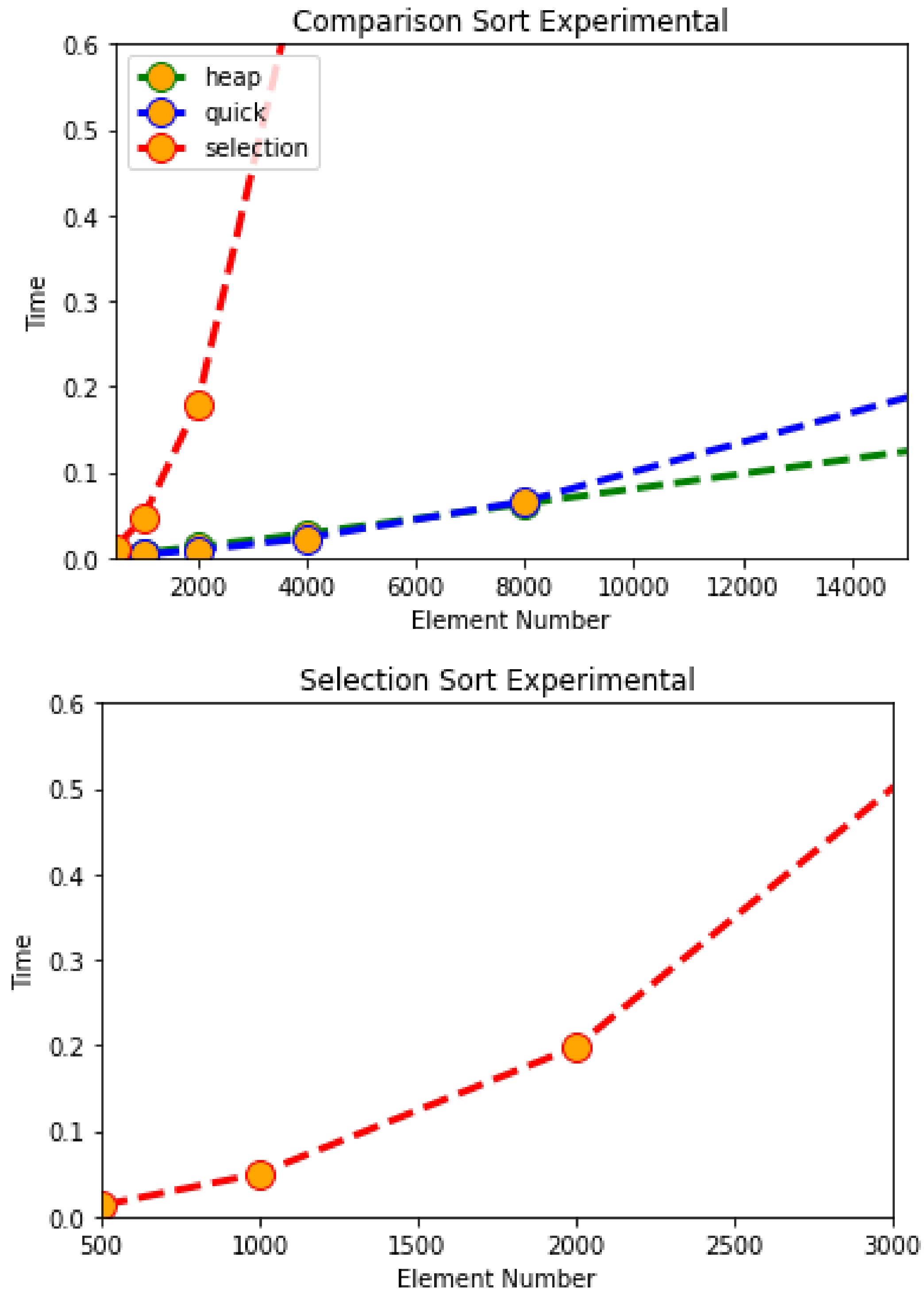


Figure 4.1. Comparison of the sorting methods**. Red is Selection**, **Blue is Quick**, and **Green is Heap** sorting algorithm.

# Conclusions

In this report, Heap Sort, Quick Sort, Selection Sort, sorting methods are examined with experimental values and compared with theoretical values.

As can be seen from the tests, the average time complexity of the algorithms is as follows, nlogn for Heap Sort, n ^ 2 for Selection Sort, and nlogn for Quick Sort.

As can be seen from the tests performed with experimental data, the graph obtained for each algorithm with the average values obtained is similar to the graph obtained with the theoretical values. The reason for the small changes in the graphics is due to the performance change of other operations on the computer (such as web browsing) while testing.

As can be seen from the tests performed, Selection Sort is slower than other algorithms. One of the biggest reasons for this is that the selection sort is easy to apply but inefficient. It has quadratic complexity in all circumstances (in the worst-case scenario as well as in the best scenario). Heapsort and Quicksort average time complexity yielded similar results. Both algorithms use the in-place sorting method. The Heap Sort algorithm gives nlogn complexity even in the worst-case scenario, while the QuickSort algorithm yields n ^ 2. But in general, the QuickSort algorithm works faster than Heap Sort. The reason for this is that the QuickSort algorithm does not change elements unnecessarily. If a bad situation is encountered, it can be switched from the QuickSort algorithm to the Heap Sort algorithm.

# References


1. Quick Sort – Quick Sort Explanation and time complexity
2. Selection Sort – Selection Sort Explanation and time complexity
3. Heap Sort – Heap Sort Explanation and time complexity

# Appendices

## Appendix 1. Selection Sort Animation

```
import pygame
import random

pygame.font.init()
my_font = pygame.font.SysFont('Comic Sans MS', 30)

ELEMENT_NUMBER = 10

pygame.font.init()

screen = pygame.display.set_mode((900, 650))

pygame.display.set_caption("SORTING VISUALISER")
run = True

width = 900
length = 600
array = [0] * ELEMENT_NUMBER
arr_clr = [(0, 255, 0)] * ELEMENT_NUMBER
# White sorted, blue unsorted, red moved, green selected
clr = [(0, 255, 0), (255, 0, 0), (0, 0, 255), (255, 255, 255)]

counter = 0


def generate_arr():
    decision = input('Press R to create random elements and press E to enter 10 numbers: ')
    if decision == 'R':
        for i in range(1, ELEMENT_NUMBER):
            arr_clr[i] = clr[0]
            array[i] = random.randrange(100, 400)

    elif decision == 'E':
        for i in range(1, ELEMENT_NUMBER):
            arr_clr[i] = clr[0]
            array[i] = int(input('Enter number: '))


def refill():
    screen.fill((0, 0, 0))
    draw()
    pygame.display.update()
    pygame.time.delay(30)


def selection_sort(arr):
    global counter
    # if counter < ELEMENT_NUMBER:
    for i in range(ELEMENT_NUMBER):
        print(i, array)
        min_idx = i
        arr_clr[i] = clr[1]
        refill()
        for j in range(i + 1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j
            arr_clr[min_idx] = clr[0]
```

```
            arr_clr[j] = clr[2]
            arr_clr[i] = clr[3]
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

        refill()

        counter += 1


def draw():
    if ELEMENT_NUMBER > width:
        element_width = int(ELEMENT_NUMBER / width)
    else:
        element_width = int(width / ELEMENT_NUMBER)
    for i in range(1, ELEMENT_NUMBER):
        pygame.draw.line(screen, arr_clr[i], ((element_width * i), length),
                          ((element_width * i),
                           array[i]),
                          element_width)


generate_arr()


def run_code(element_number):
    global run
    global ELEMENT_NUMBER
    ELEMENT_NUMBER = element_number
    while run:
        for event in pygame.event.get():

            if event.type == pygame.QUIT:
                run = False
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN:
                    selection_sort(array)
        draw()
        pygame.display.update()

    pygame.quit()


run_code(10)
print(len(array))
```

## Appendix 2. Selection Sort Testing

```
import time
import random
from prettytable import PrettyTable
import matplotlib.pyplot as plt

element_numbers = [500, 1000, 2000, 4000, 8000, 16000, 32000]
test_iteration = []
array = []
mean = [0, 0, 0, 0, 0, 0, 0]
iteration = 10


def generate_arr(ELEMENT_NUMBER):
    global array
    array = []
    for numb in range(ELEMENT_NUMBER):
        array.append(random.randint(100, 400))


def selection_sort(arr, ELEMENT_NUMBER):
    for i in range(ELEMENT_NUMBER):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]


for j in range(iteration):
    run_time = []
    for i in element_numbers:
        generate_arr(i)
        start_time = time.time()
        selection_sort(array, i)
        run_time.append(time.time() - start_time)
    test_iteration.append(run_time)

t = PrettyTable(['500', '1000', '2000', '4000', '8000', '16000', '32000'])

for val in test_iteration:
    t.add_row(val)

for i in test_iteration:
    count = 0
    for j in i:
        mean[count] += mean[count] + j
        count += 1

print(t)

for i in range(0, 7):
    mean[i] = mean[i] / iteration

print('Mean values: ', mean)

x = element_numbers
y = mean

plt.plot(x, y, color='green', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='blue', markersize=12)
```

```
plt.xlim(500, 32000)
plt.ylim(0, 60)

plt.xlabel('Element Number')
plt.ylabel('Time')

plt.title('Selection Sort Experimental')

plt.show()
```

## Appendix 3. Quick Sort Animation

```
import pygame
import random

pygame.font.init()

screen = pygame.display.set_mode((900, 650))

pygame.display.set_caption("SORTING VISUALISER")
run = True

width = 900
length = 600
array = [0] * 10
arr_clr = [(0, 255, 0)] * 10
clr = [(0, 255, 0), (255, 0, 0), (0, 0, 255), (255, 255, 255)]

counter = 0
step = 0


def generate_arr():
    decision = input('Press R to create random elements and press E to enter 10 
numbers: ')
    if decision == 'R':
        for i in range(1, 10):
            arr_clr[i] = clr[0]
            array[i] = random.randrange(100, 400)

    elif decision == 'E':
        for i in range(1, 10):
            arr_clr[i] = clr[0]
            array[i] = int(input('Enter number: '))


def refill():
    screen.fill((0, 0, 0))
    draw()
    pygame.display.update()
    pygame.time.delay(30)


def quicksort(arr, l, r):
    global counter
    global step
    if l < r:
        pi = partition(arr, l, r)
        print(arr)
        if counter == step:
            quicksort(arr, l, pi - 1)
            counter += 1

            refill()
            for i in range(0, pi + 1):
                arr_clr[i] = clr[3]
            quicksort(arr, pi + 1, r)


def partition(arr, low, high):
    global counter
    pygame.event.pump()
```

```
    pivot = arr[high]
    arr_clr[high] = clr[2]
    i = low - 1

    for j in range(low, high):
        # if low < counter < high:
        arr_clr[j] = clr[1]
        refill()
        arr_clr[high] = clr[2]
        arr_clr[j] = clr[0]
        arr_clr[i] = clr[0]
        if arr[j] < pivot:
            i = i + 1
            arr_clr[i] = clr[1]
            arr[i], arr[j] = arr[j], arr[i]
    refill()
    arr_clr[i] = clr[0]
    arr_clr[high] = clr[0]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    return i + 1


def draw():
    element_width = 70

    for i in range(1, 10):
        pygame.draw.line(screen, arr_clr[i], (80 * i - 3, length),
                                 (80 * i - 3,
                                  array[i]),
                                 element_width)


generate_arr()

while run:
    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            run = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RETURN:
                quicksort(array, 1, len(array) - 1)
                step += 1
    draw()
    pygame.display.update()

pygame.quit()
```

## Appendix 4. Quick Sort Testing

```
import time
import random
from prettytable import PrettyTable
import matplotlib.pyplot as plt

element_numbers = [500, 1000, 2000, 4000, 8000, 16000, 32000]
array = []
mean_quick = [0, 0, 0, 0, 0, 0, 0]
iteration = 10

def generate_arr(ELEMENT_NUMBER):
    global array
    array = []
    for numb in range(ELEMENT_NUMBER):
        array.append(random.randint(100, 400))

def quick_sort(arr, l, h):
    # Create an auxiliary stack
    size = h - l + 1
    stack = [0] * size

    # initialize top of stack
    top = -1

    # push initial values of l and h to stack
    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h

    # Keep popping from stack while is not empty
    while top >= 0:

        # Pop h and l
        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1

        # Set pivot element at its correct position in
        # sorted array
        p = partition(array, l, h)

        # If there are elements on left side of pivot,
        # then push left side to stack
        if p - 1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p - 1

        # If there are elements on right side of pivot,
        # then push right side to stack
        if p + 1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h
```

```python
def partition(arr, l, h):
    i = (l - 1)
    x = array[h]

    for j in range(l, h):
        if array[j] <= x:
            # increment index of smaller element
            i = i + 1
            array[i], array[j] = array[j], array[i]

    array[i + 1], array[h] = array[h], array[i + 1]
    return (i + 1)



test_iteration_quick = []
for j in range(iteration):
    run_time = []
    for i in element_numbers:
        generate_arr(i)
        start_time = time.time()
        quick_sort(array, 0, i-1)
        run_time.append(time.time() - start_time)
    test_iteration_quick.append(run_time)


t = PrettyTable(['500', '1000', '2000', '4000', '8000', '16000', '32000'])

for val in test_iteration:
    t.add_row(val)


for i in range(7):
  for j in range(10):
    mean_quick[i] = test_iteration_quick[j][i]
  mean_quick[i] = mean_quick[i] / iteration


print('Mean values quick sort: ', mean_quick)


print(t)

x = element_numbers
y = mean_quick


plt.plot(x, y, color='blue', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='orange', markersize=12)

plt.xlim(500, 32000)
plt.ylim(0, .1)

plt.xlabel('Element Number')
plt.ylabel('Time')

plt.title(‘Quick Sort Experimental')

plt.show()
```

**Appendix 5. Heap Sort Animation**

```
import pygame
import random

pygame.font.init()

screen = pygame.display.set_mode((900, 650))

pygame.display.set_caption("SORTING VISUALISER")
run = True

width = 900
length = 600
array = [0] * 10
arr_clr = [(0, 255, 0)] * 10
clr = [(0, 255, 0), (255, 0, 0), (0, 0, 255), (255, 255, 255)]

counter = 0
step = 0


def generate_arr():
    decision = input('Press R to create random elements and press E to enter 10
numbers: ')
    if decision == 'R':
        for i in range(1, 10):
            arr_clr[i] = clr[0]
            array[i] = random.randrange(100, 400)

    elif decision == 'E':
        for i in range(1, 10):
            arr_clr[i] = clr[0]
            array[i] = int(input('Enter number: '))


def refill():
    screen.fill((0, 0, 0))
    draw()
    pygame.display.update()
    pygame.time.delay(30)


def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        pygame.event.pump()
        heapify(arr, i, n)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        arr_clr[i] = clr[1]
        refill()
        heapify(arr, 0, i)


def heapify(arr, root, size):
    print(array)
    global counter
    left = root * 2 + 1
    right = root * 2 + 2
    largest = root
```

```
    if left < size and arr[left] > arr[largest]:
        largest = left
    if right < size and arr[right] > arr[largest]:
        largest = right
    if largest != root:
        arr_clr[largest] = clr[2]
        arr_clr[root] = clr[2]
        arr[largest], arr[root] = arr[root], arr[largest]
        refill()
        arr_clr[largest] = clr[0]
        arr_clr[root] = clr[0]
        heapify(arr, largest, size)
        refill()




def draw():
    element_width = 70

    for i in range(1, 10):
        pygame.draw.line(screen, arr_clr[i], (80 * i - 3, length),
                         (80 * i - 3,
                          array[i]),
                         element_width)


generate_arr()

while run:
    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            run = False
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RETURN:
                heap_sort(array)
                step += 1
    draw()
    pygame.display.update()

pygame.quit()
```

## Appendix 6. Heap Sort Testing

```
import time
import random
from prettytable import PrettyTable
import matplotlib.pyplot as plt

element_numbers = [500, 1000, 2000, 4000, 8000, 16000, 32000]
array = []
mean_heap = [0, 0, 0, 0, 0, 0, 0]
iteration = 10

def generate_arr(ELEMENT_NUMBER):
    global array
    array = []
    for numb in range(ELEMENT_NUMBER):
        array.append(random.randint(100, 400))


def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, i, n)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, 0, i)


def heapify(arr, root, size):
    left = root * 2 + 1
    right = root * 2 + 2
    largest = root

    if left < size and arr[left] > arr[largest]:
        largest = left
    if right < size and arr[right] > arr[largest]:
        largest = right
    if largest != root:
        arr[largest], arr[root] = arr[root], arr[largest]
        heapify(arr, largest, size)

test_iteration_heap = []
for j in range(iteration):
    run_time = []
    for i in element_numbers:
        generate_arr(i)
        start_time = time.time()
        heap_sort(array)
        run_time.append(time.time() - start_time)
    test_iteration_heap.append(run_time)

t = PrettyTable(['500', '1000', '2000', '4000', '8000', '16000', '32000'])

for val in test_iteration:
    t.add_row(val)

for i in range(7):
  for j in range(10):
    mean_heap[i] = test_iteration_heap[j][i]
  mean_heap[i] = mean_heap[i] / iteration
```

```
print('Mean values heap sort: ', mean_heap)


print(t)

x = element_numbers
y = mean_heap

plt.plot(x, y, color='green', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='orange', markersize=12)

         marker='o', markerfacecolor='orange', markersize=12)

plt.xlim(500, 32000)
plt.ylim(0, .1)

plt.xlabel('Element Number')
plt.ylabel('Time')

plt.title('Heap Sort Experimental')

plt.show()
```

## Appendix 7. Comparison of Experimental Results

```
import time
import random
from prettytable import PrettyTable
import matplotlib.pyplot as plt

element_numbers = [500, 1000, 2000, 4000, 8000, 16000, 32000]
array = []
mean_quick = [0, 0, 0, 0, 0, 0, 0]
mean_heap = [0, 0, 0, 0, 0, 0, 0]
mean_selection = [0, 0, 0, 0, 0, 0, 0]
iteration = 10

def generate_arr(ELEMENT_NUMBER):
    global array
    array = []
    for numb in range(ELEMENT_NUMBER):
        array.append(random.randint(100, 400))


def selection_sort(arr, ELEMENT_NUMBER):
    for i in range(ELEMENT_NUMBER):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def quick_sort(arr, l, h):
    # Create an auxiliary stack
    size = h - l + 1
    stack = [0] * size

    # initialize top of stack
    top = -1

    # push initial values of l and h to stack
    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h

    # Keep popping from stack while is not empty
    while top >= 0:

        # Pop h and l
        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1

        # Set pivot element at its correct position in
        # sorted array
        p = partition(array, l, h)

        # If there are elements on left side of pivot,
        # then push left side to stack
        if p - 1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
```

```
            stack[top] = p - 1

        # If there are elements on right side of pivot,
        # then push right side to stack
        if p + 1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h


def partition(arr, l, h):
    i = (l - 1)
    x = array[h]

    for j in range(l, h):
        if array[j] <= x:
            # increment index of smaller element
            i = i + 1
            array[i], array[j] = array[j], array[i]

    array[i + 1], array[h] = array[h], array[i + 1]
    return (i + 1)

def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, i, n)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, 0, i)


def heapify(arr, root, size):
    left = root * 2 + 1
    right = root * 2 + 2
    largest = root

    if left < size and arr[left] > arr[largest]:
        largest = left
    if right < size and arr[right] > arr[largest]:
        largest = right
    if largest != root:
        arr[largest], arr[root] = arr[root], arr[largest]
        heapify(arr, largest, size)

test_iteration_heap = []
for j in range(iteration):
    run_time = []
    for i in element_numbers:
        generate_arr(i)
        start_time = time.time()
        heap_sort(array)
        run_time.append(time.time() - start_time)
    test_iteration_heap.append(run_time)

test_iteration_quick = []
for j in range(iteration):
    run_time = []
    for i in element_numbers:
        generate_arr(i)
```

```
        start_time = time.time()
        quick_sort(array, 0, i-1)
        run_time.append(time.time() - start_time)
    test_iteration_quick.append(run_time)

test_iteration_selection = []
for j in range(iteration):
    run_time = []
    for i in element_numbers:
        generate_arr(i)
        start_time = time.time()
        selection_sort(array, i)
        run_time.append(time.time() - start_time)
    test_iteration_selection.append(run_time)

t = PrettyTable(['500', '1000', '2000', '4000', '8000', '16000', '32000'])

for val in test_iteration:
    t.add_row(val)

for i in range(7):
  for j in range(10):
    mean_selection[i] = test_iteration_selection[j][i]
  mean_selection[i] = mean_selection[i] / iteration

for i in range(7):
  for j in range(10):
    mean_quick[i] = test_iteration_quick[j][i]
  mean_quick[i] = mean_quick[i] / iteration

for i in range(7):
  for j in range(10):
    mean_heap[i] = test_iteration_heap[j][i]
  mean_heap[i] = mean_heap[i] / iteration

print('Mean values quick sort: ', mean_quick)

print('Mean values heap sort: ', mean_heap)

print('Mean values selection sort: ', mean_selection)

print(t)

x = element_numbers
y1 = mean_heap
y2 = mean_quick
y3 = mean_selection

plt.plot(x, y1, color='green', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='orange', markersize=12)

plt.plot(x, y2, color='blue', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='orange', markersize=12)

plt.plot(x, y3, color='red', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='orange', markersize=12)

plt.xlim(500, 32000)
plt.ylim(0, .1)

plt.xlabel('Element Number')
```

```python
plt.ylabel('Time')

#plt.title('Selection Sort Experimental')
plt.title('Comparison Sort Experimental')

plt.show()

x = element_numbers
y = []

for i in x:
  y.append(i ** 2)

plt.plot(x, y, color='green', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='blue', markersize=12)

plt.xlim(500, 32000)
plt.ylim(0, 1024000000)

plt.xlabel('Element Number')
plt.ylabel('Time')

#plt.title('Selection Sort Experimental')
plt.title('Selection Sort Theoretical')

plt.show()

x = element_numbers
y = []

for i in x:
  y.append(i * i)


print(y)

plt.plot(x, y, color='green', linestyle='dashed', linewidth=3,
         marker='o', markerfacecolor='blue', markersize=12)

plt.xlim(500, 32000)
plt.ylim(0, 1024000000)

plt.xlabel('Element Number')
plt.ylabel('Time')

plt.title('Quick Sort Theoretical Worst Case')

plt.show()

x = np.linspace(-5,5,100)

x
```