**CSE 2046 – Analysis of Algorithms PROJECT #2**

**Course Instructor: Asisstant Prof. Ömer Korçak**

**Course T.A.: Muhammed Nur Avcil**

**150117828 – Cem Güleç**

1.) Introduction:

In this project, I am aiming to compare some specific algorithms which are targeting to find median of a given array. After giving a brief introduction to those algorithms, I will be plotting those results, comparing every algorithm with each other, and will be commenting on the result that I obtained. Below are the algorithms that are used:

1. Insertion Sort

2. Merge Sort

3. Max-heap removal

4. Quick select using first element as pivot

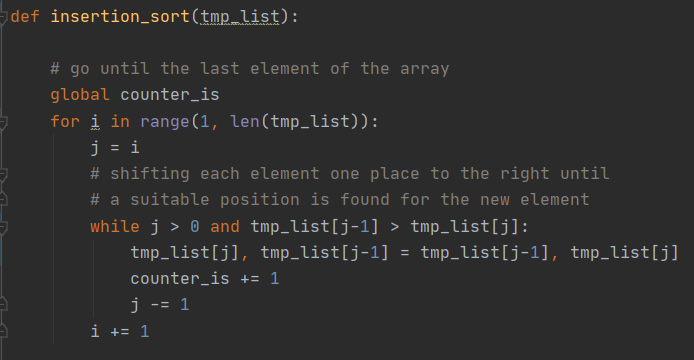
5. Quick select using median-of-three pivot selection.

Comparing those algoritms according to their complexities:

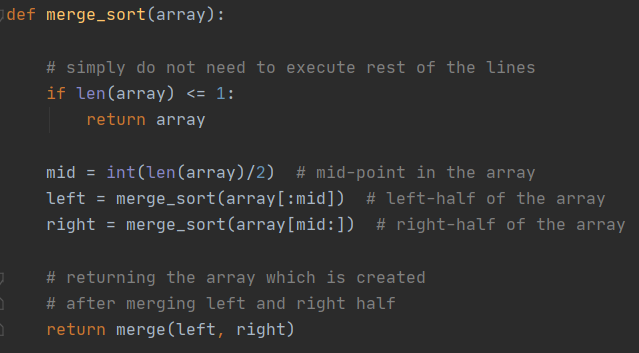
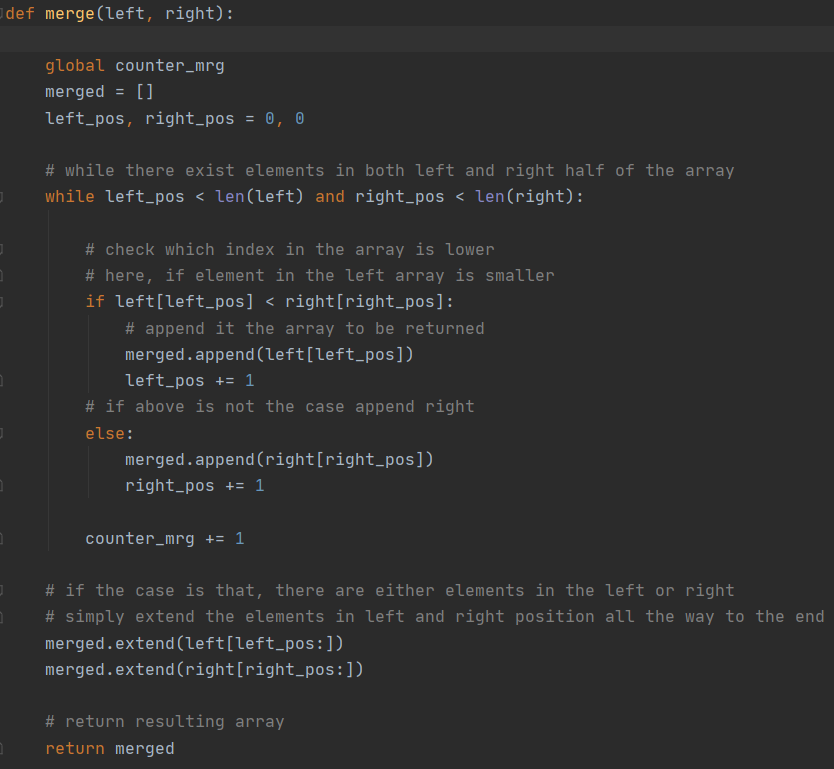
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Case / Algorithm | Insertion Sort | Merge Sort | Max Heap | Quick select 1 | Quick Select 2 |
| Best Case |  |  |  |  |  |
| Average Case |  |  |  |  |  |
| Worst Case |  |  |  |  |  |

2.) Code Implementation:

2.1) Insertion Sort:

  
  
 Implementation of insertion sort consisting of taking a temporary list which is a copy of the original list and implementing sorting on it. Here, I also used “counter\_is” which will be counting the number of occurrences of the basic operation.

2.2) Merge Sort:



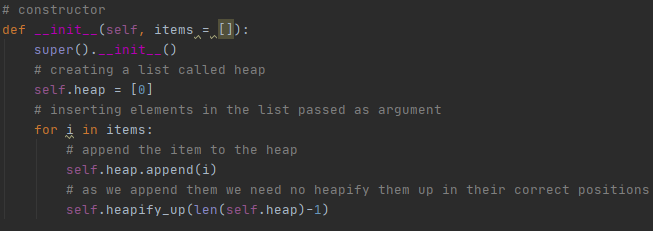
Implementation of merge sort consists of taking a list,  
which is to be sorted. In the main function, I declare   
our mid, left and right arrays. Then, I am calling   
another function which is called “merge” to   
implement sorting on these parts.

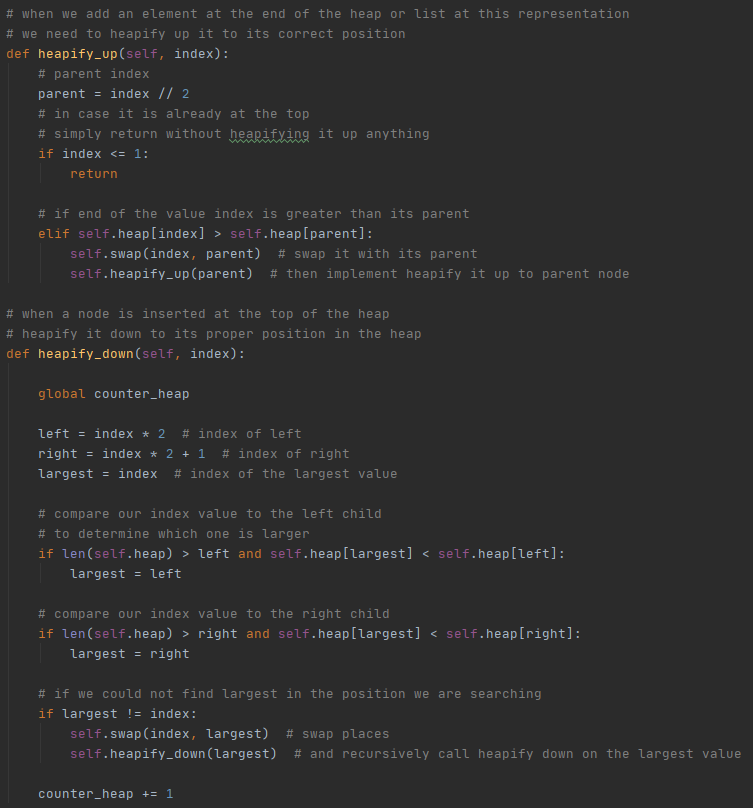
Additionally, “counter\_mrg” is used in order to count number of occurrences of the basic operation in this algorithm.

2.3) Max heap removal:

In this algorithm, I firstly created a class which is called “max\_heap”. Implementation is based on array representation of max heap. Dependent on the class there are several functions which are used occasionally to implement sorting inside the max heap, which are named as “heapify\_up” and “heapify\_down”. These functions used in order to fit an element into the correct place in the heap.  
 On the other hand, 2 more functions named as “pop” and “peek” used here.   
  
Pop function: Allows me to pop an element from the heap, then sort rest of the elements in order to find a correct position to themselves, while not violating max-heap rules.

Peek function: Allows me to see the root of the heap.

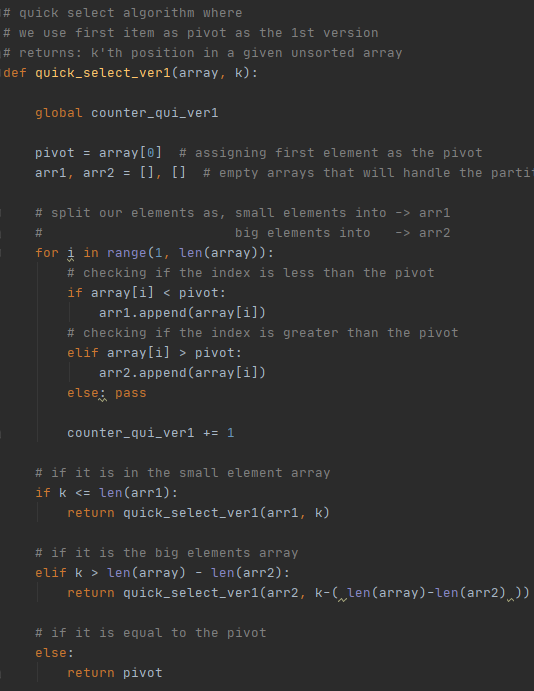
In the constructor, items that are taken as an input are appended to the heap one by one. Also, used “heapify\_up” function to put them into the correct place.



As in the image left, functions are defined which takes index as an input. Then, in these functions, this specific index is checked whether it is violating the rules or not.

If it violates anything, it needs to be put in the correct position.

Additionally, “counter\_heap” is used in order to count the number of occurrences of the basic operation in this algorithm.

2.4) Quick select algorithm:

In this algorithm, pivot is chosen to be the first element of the array given.

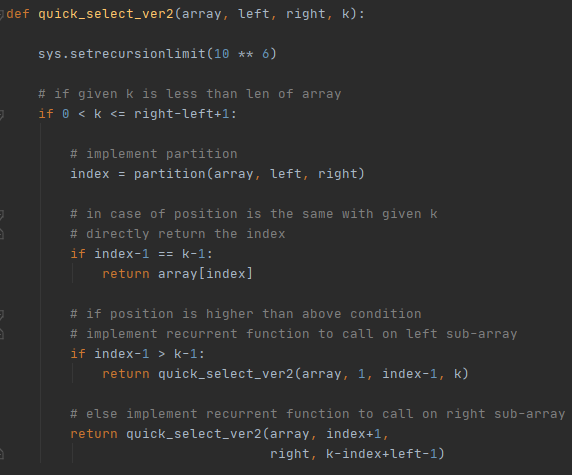
After having the pivot known, I splitted it into 2 arrays depending on whether an element is less than or greater than the pivot value.

After splitting is done, then depending on the case that are declared as in the image, recurrently calling our quick select function to find the correct value to be returned.

Additionally, here “counter\_qui\_ver1” is used in order to count the number of occurrences of the basic operation in this algorithm.

2.5) Quick select algorithm (using median-of-three pivot selection):



In this algorithm, first, I have declared the mid, left, and right indexes which will be used here as the pivots. Then, differently from algorithm which is used in (2.4), implemented partition according to those 3 values given.

Additionally here, “counter\_qui\_ver2” is used in order to count the number of occurrences of the basic operation in the algorithm.

3) Input Selection:

In phase of choosing which input types to be used, I seeked on some specific features on it. First of all, I implemented different type of scenarios in the inputs, which are as listed in the below:

3.1) Input type where all of the number are same.

3.2) Input type where the list is consisting of fully random numbers, between some specific value. (which I will specify in below section.)

3.3) Input type where the list of numbers are sorted.

3.4) Input type where the list of numbers in reverse sorted way.

3.5) Input type where first 10% portion of the input list is repeated over and over again.

3.6) Input type where from beginning 20% of the list is sorted but rest of the list is consisting of unsorted numbers.

Reason for picking those particular scenarios is that, to see how the algorithm will react on different occasions. In a particular scenario, some of the algorithm might end up having their worst cases, some of them are not.

On the other hand, I created different scenarios where the input sizes different, which are also listed in the below:

For the input sizes: 100, 500, 1000, 2000, 4000, 5000, 10000, 50000  
is used combining them with different input scenario type.

4) Implementation of the inputs:

There are 6 formats to be implement with 9 different input size, which makes a total of 54 inputs.  
  
Example input format: “100\_sorted\_5k” means that the input size is 100, it is a sorted list and the value ranges are between 1000 – 5000.  
  
Other than this example, “same” used for (3.1), “random” used for (3.2), “sorted” used for (3.3), “Rsorted” used for (3.4), “repeat” used for (3.5), “lportion” used for (3.6)

5) Technical details about the computer (which is used as test environment):

For writing the problem itself, I have used Python as a language. Additional to that, an extra problem handles the input array generation, which is also written in Python.

My computer which is used as the test environment has Intel 7200U 2.5 GHz, running in Windows operating system. Since I have worked alone in this project only this computer is used to test algorithms and get the corresponding results.

6) Results

Insertion Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size / Input | 3.1 | 3.2 | 3.3 | 3.4 | 3.5 | 3.6 |
| 100 | 0.0000198 | 0.0002343 | 0.0000107 | 0.0001232 | 0.0002389 | 0.0002504 |
| 500 | 0.0000839 | 0.0040529 | 0.0000504 | 0.0008725 | 0.0042006 | 0.0053015 |
| 1000 | 0.0000871 | 0.0173663 | 0.0001076 | 0.0028417 | 0.0172836 | 0.0209538 |
| 2000 | 0.0002410 | 0.0663209 | 0.0001869 | 0.0144340 | 0.0752209 | 0.0827621 |
| 5000 | 0.0004901 | 0.4669822 | 0.0004820 | 0.0724326 | 0.4740214 | 0.5265168 |
| 10000 | 0.0009163 | 1.8085526 | 0.0010286 | 0.2783200 | 1.9172699 | 2.1278930 |
| 50000 | 0.0045832 | 5.2045566 | 0.0047236 | 11.1601156 | 49.3508224 | 63.2240390 |

Complexity comparison: 3.1 > 3.3 > 3.2 > 3.4 > 3.5 > 3.6

Merge Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size / Input | 3.1 | 3.2 | 3.3 | 3.4 | 3.5 | 3.6 |
| 100 | 0.0004301 | 0.0003333 | 0.0002387 | 0.0002345 | 0.0005595 | 0.0004464 |
| 500 | 0.0013002 | 0.0017910 | 0.0012542 | 0.0012911 | 0.0015854 | 0.0019273 |
| 1000 | 0.0028398 | 0.0039606 | 0.0025932 | 0.0026667 | 0.0034520 | 0.0074305 |
| 2000 | 0.0059219 | 0.0090495 | 0.0068848 | 0.0073441 | 0.0080277 | 0.0092157 |
| 5000 | 0.0163011 | 0.0249020 | 0.0168525 | 0.0162953 | 0.0201169 | 0.0240648 |
| 10000 | 0.0336373 | 0.0560159 | 0.0364436 | 0.0336587 | 0.0441034 | 0.0494537 |
| 50000 | 0.1884555 | 0.1260609 | 0.2010002 | 0.1849502 | 0.2492394 | 0.3051851 |

Complexity comparison: 3.2 > 3.4 > 3.1 > 3.3 > 3.5 > 3.6

Max Heap Removal

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size / Input | 3.1 | 3.2 | 3.3 | 3.4 | 3.5 | 3.6 |
| 100 | 0.0001268 | 0.0002926 | 0.0002672 | 0.0003179 | 0.0002837 | 0.0002857 |
| 500 | 0.0003330 | 0.0020892 | 0.0018761 | 0.0019562 | 0.0019754 | 0.0022855 |
| 1000 | 0.0007327 | 0.0052388 | 0.0045898 | 0.0042853 | 0.0043716 | 0.0079787 |
| 2000 | 0.0014597 | 0.0111236 | 0.0096660 | 0.0111403 | 0.0121436 | 0.0102550 |
| 5000 | 0.0040421 | 0.0312227 | 0.0273684 | 0.0289549 | 0.0278978 | 0.0280445 |
| 10000 | 0.0075910 | 0.0610727 | 0.0599115 | 0.0632248 | 0.0593542 | 0.0613781 |
| 50000 | 0.0395567 | 0.1311557 | 0.3514281 | 0.3631937 | 0.3885011 | 0.3838904 |

Complexity comparison: 3.1 > 3.2 > 3.3 > 3.4 > 3.6 > 3.5

Quick select using first element as pivot

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size / Input | 3.1 | 3.2 | 3.3 | 3.4 | 3.5 | 3.6 |
| 100 | 0.0000267 | 0.0000789 | 0.0007824 | 0.0007143 | 0.0000960 | 0.0003990 |
| 500 | 0.0001093 | 0.0003357 | 0.0194777 | 0.0161823 | 0.0018956 | 0.0086527 |
| 1000 | 0.0001513 | 0.0005535 | 0.0714470 | 0.0675652 | 0.0072840 | 0.0455118 |
| 2000 | 0.0002987 | 0.0012347 | 0.2666317 | 0.2491232 | 0.0255870 | 0.1226245 |
| 5000 | 0.0007065 | 0.0036844 | 1.1485418 | 1.0407835 | 0.1236154 | 0.5166389 |
| 10000 | 0.0014549 | 0.0071542 | 3.1153901 | 2.5818274 | 0.2991289 | 1.3216586 |
| 50000 | 0.0072842 | 0.0171002 | 18.1145067 | 15.4541213 | 1.7367442 | 8.1465071 |

Complexity comparison: 3.1 > 3.2 > 3.5 > 3.6 > 3.4 > 3.3

Quick select using median-of-three pivot selection

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Size / Input | 3.1 | 3.2 | 3.3 | 3.4 | 3.5 | 3.6 |
| 100 | 0.0014713 | 0.0001178 | 0.0007267 | 0.0003792 | 0.0001269 | 0.0001414 |
| 500 | 0.0182377 | 0.0005945 | 0.0212925 | 0.0034035 | 0.0021336 | 0.0009201 |
| 1000 | 0.0761306 | 0.0007540 | 0.0817669 | 0.0102169 | 0.0083510 | 0.0017476 |
| 2000 | 0.3456877 | 0.0027057 | 0.3075901 | 0.0324619 | 0.0483625 | 0.0034182 |
| 5000 | 1.9401605 | 0.0066443 | 1.9497969 | 0.1013902 | 0.2894433 | 0.0077343 |
| 10000 | 7.7364655 | 0.0219370 | 8.6385514 | 0.2556395 | 0.8254595 | 0.0240011 |
| 50000 | 40.487596 | 0.0819370 | 61.0240111 | 5.4287402 | 28.4044447 | 0.3230944 |

Complexity comparison: 3.2 > 3.6 > 3.4 > 3.5 > 3.1 > 3.3