03.05.2022

Step 1: Designing the Experiment:

I created 15 input files for this experiment. input files consist of different inputs with different sizes. I specified which files I used for which algorithm in step 3.

input1:its randomly assigned numbers with size of 5000

input2: its randomly assigned numbers with size of 7500

input3: its randomly assigned numbers with size of 10000

input4: its randomly assigned numbers with size of 15000

input5: its randomly assigned numbers with size of 20000

input6:it starts 5000 and decreasing one by one until to 1

input7:it starts 7500 and decreasing one by one until to 1

input8:it starts 10000 and decreasing one by one until to 1

input9:it starts 15000 and decreasing one by one until to 1

input10:it starts 20000 and decreasing one by one until to 1

input11:it starts 0 and increasing one by one until to 4999

input12:it starts 1 and increasing one by one until to 7500

input13:it starts 1 and increasing one by one until to 10000

input14:it starts 1 and increasing one by one until to 15000

input15:it starts 1 and increasing one by one until to 20000

Deciding Metrics

I counted the actual number of basic operation's executions for metrics cause when ı using physical unit of time its not clear to see performance of algorithms and that way will be more efficent than unit of time variable.

Step 2: Coding and Running:

I selected java language programing for this experiment.For the coding part sometimes ı coded by looking at algorithms sometimes ı used some websites.These the website’s links:

<https://stackoverflow.com/questions/19897911/inserting-and-removing-from-a-max-heap-java>

<https://stackoverflow.com/questions/7559608/median-of-three-values-strategy>

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

<https://stackoverflow.com/questions/43016874/sort-part-of-array-with-heapsort-bug>

<https://linuxhint.com/quick-sort-java-explained/>

<https://leetcode.com/problems/sort-an-array/discuss/1377878/Java-or-Quick-Sort-or-Median-of-3>

<https://www.geeksforgeeks.org/quickselect-algorithm/>

<https://bilgisayarkavramlari.com/2008/08/09/birlestirme-siralamasi-merge-sort/?highlight=merge%20sort>

Let’s examine the time complexity of some algorithms

INSERTİON SORT ALGORİTHM

The two nested loops are an indication that we are dealing with quadratic effort, meaning with time complexity of O(n²)\*. This is the case if both the outer and the inner loop count up to a value that increases linearly with the number of elements.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

### **Average Time Complexity**

The highest power of n in this term is n²; the time complexity for shifting is, therefore, O(n²). This is also called "quadratic time".

For comparison operations, we have one more than shift operations (or the same amount if you move an element to the far left). The time complexity for the comparison operations is, therefore, also O(n²).

The element to be sorted must be placed in the correct position as often as there are elements minus those that are already in the right position – so n-1 times at maximum. Since there is no n² here, but only an n, we speak of "linear time", noted as O(n).

When considering the overall complexity, only the highest level of complexity counts.Therefore follows:

The average time complexity of insertion sort: O(n²)

### **Worst-Case Time Complexity**

In the worst case, the elements are sorted completely descending at the beginning. In each step, all elements of the sorted sub-array must, therefore, be shifted to the right so that the element to be sorted – which is smaller than all elements already sorted in each step – can be placed at the very beginning.

Insortion sort algorithm result for input2 file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

as seen from the photo number of number of basic operation's executions turns out to be very large.

The worst-case time complexity of insertion Sort: O(n²)

**Best-Case Time Complexity**

If the elements already appear in sorted order, there is precisely one comparison in the inner loop and no swap operation at all.

With n elements, that is, n-1 steps (since we start with the second element), we thus come to n-1 comparison operations. Therefore:

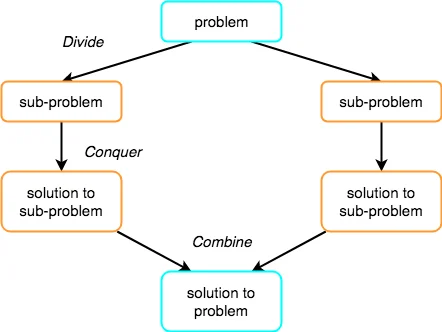
The best-case time complexity of insertion Sort is:O(n)

Merge Sort Algorithm

In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. Divide the problem into multiple small problems.
2. Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. Combine the solutions of the subproblems to find the solution of the actual problem.

****

Merge Sort is quite fast, and has a time complexity of O(n\*logn) It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

Worst Case Time Complexity [ Big-O ]: **O(n\*log n)**

Best Case Time Complexity [Big-omega]: **O(n\*log n)**

Average Time Complexity [Big-theta]: **O(n\*log n)**

Space Complexity: **O(n)**

Quick-Sort Algorithm

Quicksort works according to the "divide and conquer" principle:

First, we divide the elements to be sorted into two sections - one with small elements and one with large elements.

The so-called *pivot element* determines which elements are small and which are large. The pivot element can be any element from the input array. (The *pivot strategy* determines which one is chosen, more on this later.)

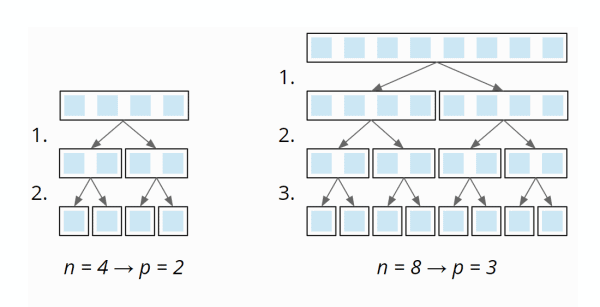
The array is now rearranged so that:

* the elements that are smaller than the pivot element end up in the left section,
* the elements that are larger than the pivot element end up in the right section,
* the pivot element is positioned between the two sections - which also is its final position.

### **Best-Case Time Complexity**

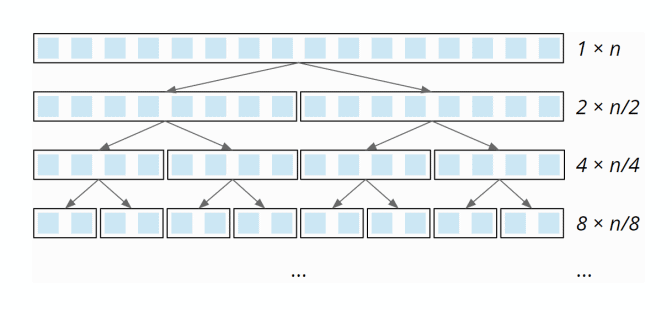
Quicksort achieves optimal performance if we always divide the arrays and subarrays into two partitions of equal size.

Because then, if the number of elements *n* is doubled, we only need one additional partitioning level *p*. The following diagram shows that two partitioning levels are needed with four elements – and only one more with eight elements:



So the number of partitioning levels is log2 n.

At each partitioning level, we have to divide a total of n elements into left and right partitions (1 × n at the first level, 2 × n/2 at the second, 4 × n/4 at the third, etc.):



This partitioning is done – due to the single loop within the partitioning – with linear complexity: When the array size doubles, the partitioning effort doubles as well. The total effort is, therefore, the same at all partitioning levels.

So we have *n* elements times *log2 n* partitioning levels. Therefore:

The best-case time complexity of Quicksort is: *O(n log n)*

### **Average-Case Time Complexity**

Unfortunately, the average time complexity cannot be derived without complicated mathematics, which would go beyond this article's scope.

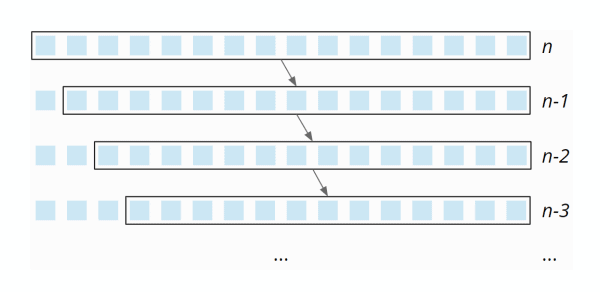
The article concludes that the average number of comparison operations is *1.39 n × log2 n* – so we are still in a quasilinear time. Therefore:

The best-case time complexity of Quicksort is also: *O(n log n)*

### **Worst-case Time Complexity**

If the pivot element is always the smallest or largest element of the (sub)array (e.g. because our input data is already sorted and we always choose the last one as the pivot element), the array would not be divided into two approximately equally sized partitions, but one of length 0 (since no element is larger than the pivot element) and one of length *n-1* (all elements except the pivot element).

Therefore we would need *n* partitioning levels with a partitioning effort of size *n*, *n-1*, *n-2*, etc.:



The partitioning effort decreases linearly from *n* to *0* – on average, it is, therefore, *½ n*. Thus, with *n* partitioning levels, the total effort is *n × ½ n = ½ n²*. Therefore:

The worst-case time complexity of Quicksort is: *O(n²)*

Partial Selection Sort

The concept used in Selection Sort helps us to partially sort the array up to kth smallest(or largest) element for finding the kth smallest(or largest) element in an array. Thus a partial selection sort yields a simple selection algorithm that takes O(k\*n) time to sort the array. This is asymptotically inefficient but can be sufficiently efficient if k is small, and is easy to implement.

Below is the algorithm for partial selection sort:

function partialSelectionSort(arr[0..n], k) {

for i in [0, k) {

minIndex = i

minValue = arr[i]

for j in [i+1, n) {

if (arr[j] < minValue) then

minIndex = j

minValue = arr[j]

swap(arr[i], arr[minIndex])

}

}

return arr[k]

}

Heap Sort Algorithm

The heapsort algorithm consists of two phases: In the first phase, the array to be sorted is converted into a max heap. And in the second phase, the largest element (i.e., the one at the tree root) is removed, and a new max heap is created from the remaining elements.

## Heapsort Time Complexity

## Its consist two part

### **Time Complexity of the heapify() Method**

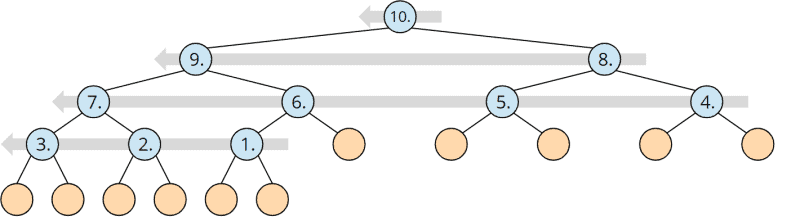
## In the heapify() function, we walk through the tree from top to bottom. The height of a binary tree (the root not being counted) of size n is log2 n at most, i.e., if the number of elements doubles, the tree becomes only one level deeper:

The complexity for the heapify() function is accordingly O(log n).

### **Time Complexity of the buildHeap() Method**

To initially build the heap, the heapify() method is called for each parent node – backward, starting with the last node and ending at the tree root.

A heap of size n has n/2 (rounded down) parent nodes:



Since the complexity of the heapify() method is O(log n) as shown above, the complexity for the buildHeap() method is, therefore, maximum\* O(n log n).

### **Total Time Complexity of Heapsort**

The heapify() method is called *n-1* times. So the total complexity for repairing the heap is also *O(n log n)*.

Both sub-algorithms, therefore, have the same time complexity. Hence:

The time complexity of Heapsort is:*O(n log n)*

Quick Select Algorithm

quickselect is a selection algorithm to find the kth smallest element in an unordered list. It is related to the quicksort sorting algorithm. it is efficient in practice and has good average-case performance, but has poor worst-case performance. Quickselect and its variants are the selection algorithms most often used in efficient real-world implementations.

Quickselect uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for. This reduces the average complexity (nlogn) to (n),with a worst case of O(n^2).

Time Complexity

Like quicksort, quickselect has good average performance, but is sensitive to the pivot that is chosen. If good pivots are chosen, meaning ones that consistently decrease the search set by a given fraction, then the search set decreases in size exponentially and by induction (or summing the [geometric series](https://en.wikipedia.org/wiki/Geometric_series)) one sees that performance is linear, as each step is linear and the overall time is a constant times this (depending on how quickly the search set reduces). However, if bad pivots are consistently chosen, such as decreasing by only a single element each time, then worst-case performance is quadratic: {\displaystyle O(n^{2}).}OO(n^2) .This occurs for example in searching for the maximum element of a set, using the first element as the pivot, and having sorted data. The probability of the worst-case occurring decreases exponentially with n{\displaystyle n,}n,,so quickselect has almost certain O(n) time complexity.

Step 3: Illustrating and Analyzing Results:

Insertion-Sort

I plotted the graph for different sizes,different k values and different inputs.

Input11-input15 for the Best-Case

Input6-input10 for the Worst-Case

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| input size | Best-Case | Worst-Case | Average-Case | |
| 0 | 0 | 0 | 0 |
| 5000 | 0 | 12497500 | 6316680 |
| 7500 | 0 | 28121250 | 14005047 |
| 10000 | 0 | 49995000 | 24975175 |
| 15000 | 0 | 1,12E+08 | 56019326 |
| 20000 | 0 | 2E+08 | 99967602 |

Line ‘unsorted’ on the graph shows us the Average-Case.

Line ‘Sorted in ascending order’ on the graph shows us the Best-Case.

Line ‘Sorted in descending order’ on the graph show us the Worst-Case.

|  |  |
| --- | --- |
| Result or different input sizes and k values |  |
|  |  |
| metin içeren bir resim  Açıklama otomatik olarak oluşturuldu | metin içeren bir resim  Açıklama otomatik olarak oluşturuldu |
|  |  |
| metin içeren bir resim  Açıklama otomatik olarak oluşturuldu | metin içeren bir resim  Açıklama otomatik olarak oluşturuldu |
|  |  |

Merge-Sort

I plotted the graph for different sizes,different k values and different inputs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| size | Unsorted | Sorted in descending order | Sorted in ascending order | |
| 0 | 0 | 0 | 0 |  |
| 5000 | 55256 | 29804 | 32004 |  |
| 7500 | 87135 | 47376 | 49432 |  |
| 10000 | 120458 | 64608 | 69008 |  |
| 15000 | 189330 | 102252 | 106364 |  |
| 20000 | 260797 | 139216 | 148016 |  |

Unsoted🡪 Worst-Case

Sorted in descending order🡪 Best-Case

Result or different input sizes and k values:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Quick-Sort

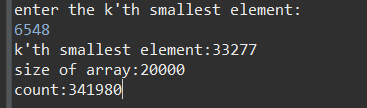
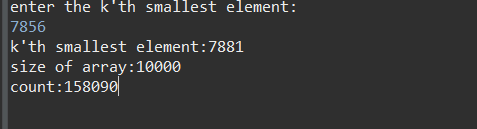
I plotted the graph for different sizes,different k values and different inputs.

Maybe you get StackOverFlowError when you get this message please increase the JVM stack size!!!!!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| size | Unsorted | Sorted in descending order | Sorted in ascending order | | |
| 0 | 0 | 0 | 0 |  |  |
| 5000 | 70208 | 12497500 | 12497500 |  |  |
| 7500 | 115943 | 28121250 | 28121250 |  |  |
| 10000 | 158090 | 49995000 | 49995000 |  |  |
| 15000 | 250149 | 112492500 | 1,12E+08 |  |  |
| 20000 | 341980 | 199990000 | 2E+08 |  |  |

Unsorted 🡪Best Case

I got the same result for Sorted in descending(input6-10) order and Sorted in ascending order(input11-15) files.



metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Partial Selection Sort

Partial selection sort complexity just depends de k values therefore k value indicates number of count.

inputs are selected from randomly generated input files(input1-input5)

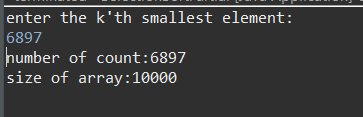
I randomly entered some k values and plotted it.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | size | k values | count | | 0 | 0 | 0 | | 5000 | 2468 | 2468 | | 7500 | 45 | 45 | | 10000 | 6897 | 6897 | | 15000 | 9658 | 9658 | | 20000 | 16589 | 16589 | |  |

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Heap Sort

I plotted the graph for different sizes,different k values and different inputs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| size | Unsorted | Sorted in descending order | Sorted in ascending order | |
| 0 | 0 | 0 | 0 |  |
| 5000 | 43692 | 51810 | 56809 |  |
| 7500 | 76370 | 81810 | 89309 |  |
| 10000 | 95834 | 113618 | 123617 |  |
| 15000 | 165575 | 178618 | 1,94E+05 |  |
| 20000 | 223440 | 247234 | 2,67E+05 |  |

Quick Select (Array partitioning)

I kept the input size fixed(10000) and ı entered the some random k values for the randomly created input file(unsorted),descending order input file and ascending order input file and plotted it.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| k values | Unsorted | Sorted in descending order | Sorted in ascending order | |
| 0 | 0 | 0 | 0 |  |
| 348 | 12505 | 6717444 | 3419274 |  |
| 819 | 30803 | 15037659 | 7854210 |  |
| 2411 | 28882 | 36591747 | 21202334 |  |
| 4589 | 27052 | 49657569 | 3,54E+07 |  |
| 7367 | 47136 | 38796722 | 4,65E+07 |  |
| 9182 | 39096 | 15029297 | 49660847 |  |

Unsorted ->>>> Best-Case

Sorted in ascending order->>>Worst Case

For the unsorted input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in descending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in ascending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

This time I kept the k value constant and changed the input size then plotted it.

K value:2187 (arbitrarily chosen)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sizes | Unsorted | Sorted in descending order | Sorted in ascending order | |  |
| 0 | 0 | 0 | 0 |  |  |
| 5000 | 11037 | 12301875 | 8542422 |  |  |
| 7500 | 34507 | 23236875 | 14009922 |  |  |
| 10000 | 27155 | 34171875 | 19477422 |  |  |
| 15000 | 48321 | 56041875 | 3,04E+07 |  |  |
| 20000 | 83968 | 77911875 | 4,13E+07 |  |  |

Unsorted Best-Case

Sorted in descending order Worst-Case

For the unsorted input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in descending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in ascending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

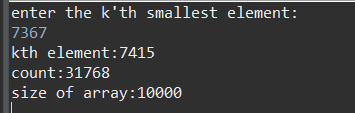
Quick Select Algorithm (Median-of-three pivot selection)

Similar to Quick Select Algorithm based array partitioning I kept the input size fixed(10000) and ı entered the some random k values (like at the top) for the randomly created input file(unsorted),descending order input file and ascending order input file and plotted it.

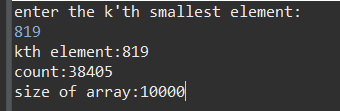
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| k values | Unsorted | Sorted in descending order | Sorted in ascending order | |
| 0 | 0 | 0 | 0 |  |
| 348 | 17871 | 39186 | 19985 |  |
| 819 | 18385 | 38405 | 19986 |  |
| 2411 | 17122 | 37143 | 19973 |  |
| 4589 | 27471 | 34004 | 2,00E+04 |  |
| 7367 | 31768 | 30934 | 2,00E+04 |  |
| 9182 | 24014 | 33097 | 19985 |  |

For the unsorted input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in descending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in ascending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

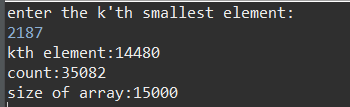
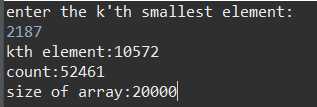
Açıklama otomatik olarak oluşturuldu

This time I kept the k value constant and changed the input size then plotted it.

K value:2187

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sizes | Unsorted | Sorted in descending order | Sorted in ascending order | |
| 0 | 0 | 0 | 0 |  |
| 5000 | 10992 | 16379 | 9372 |  |
| 7500 | 22447 | 27394 | 14984 |  |
| 10000 | 17710 | 36675 | 19371 |  |
| 15000 | 35082 | 57378 | 3,00E+04 |  |
| 20000 | 52461 | 74954 | 3,94E+04 |  |

For the unsorted input file:



For the sorted in descending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

For the sorted in ascending order input file:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Compare The Some Algorithms

Insertion Sort vs Merge Sort

I compared the first two algorithm with using unsorted input file(average case) and ı plotted it

Input1-input5

|  |  |  |
| --- | --- | --- |
| sizes | Insertion Sort | Merge Sort |
| 0 | 0 | 0 |
| 5000 | 6316680 | 55256 |
| 7500 | 14005047 | 87135 |
| 10000 | 24975175 | 120458 |
| 15000 | 56019326 | 189330 |
| 20000 | 99967602 | 260797 |

Time complexity for the insortion sort(average case):O(n^2)

Time complexity for the merge sort:O(nlogn)

The findings obtained according to the experiment meets the theoretical expectations.Considering average time complexity of both algorithm we can say that Merge Sort is efficient in terms of time.

Insertion Sort Vs Quick Sort

I decided to compare worst case time complexity of both algorithm

Time complexity for the insortion sort(worst case):O(n^2)

Time complexity for the quick sort(worts case):O(n^2)

|  |  |  |
| --- | --- | --- |
| input size | Insertion Sort | Quick Sort |
| 0 | 0 | 0 |
| 5000 | 12497500 | 12497500 |
| 7500 | 28121250 | 28121250 |
| 10000 | 49995000 | 49995000 |
| 15000 | 1,12E+08 | 1,12E+08 |
| 20000 | 2E+08 | 2E+08 |

For the quicksort already sorted and reverse sorted inputs are the worst cases also for insertion sort worst case is already reverse sorted input.The fundings of experiment support this expectations. As seen in the graph above.

Quick Select (Array Partitioning vs Median of Three Pivot Selection)

I compared the both algorithm with using unsorted input file(gives us Best-Case time complexity) as a result the findings are very close to each other.

Best Case time complexity is O(n).

|  |  |  |
| --- | --- | --- |
| k values | Array Partitioning | Median of Three |
| 348 | 17871 | 12505 |
| 819 | 18385 | 30803 |
| 2411 | 17122 | 28882 |
| 4589 | 27471 | 27052 |
| 7367 | 31768 | 47136 |
| 9182 | 24014 | 39096 |

FARUK AKDEMİR

150119012