

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

CSE2046

ANALYSIS OF ALGORITHMS

**PROJECT - 1**

Due Date: 11.05.2022

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dept | Student Id | Name Surname |
| 1 | CSE | 150119018 | Umut Emre Önder |
| 2 | CSE | 150119035 | Batuhan Baştürk |

**Distribution of Tasks**

Batuhan Baştürk 🡨 Created SortingAlgorithms.class, preparation of tests, preparation of the report,editing the report

Umut Emre Önder 🡨 Created TestArea.class, preparation of tests, transferring tests to excel, preparation of the report

**Contents**

[Algorithms and Their Explanations 5](#_Toc103122598)

[Insertion Sort 5](#_Toc103122599)

[Algorithm 5](#_Toc103122600)

[Pseudocode & Execution Example 5](#_Toc103122601)

[Space complexity & Time Complexities 6](#_Toc103122602)

[Merge Sort 7](#_Toc103122603)

[Algorithm 7](#_Toc103122604)

[Pseudocode & Execution Example 7](#_Toc103122605)

[Space Complexity & Time Complexities 9](#_Toc103122606)

[Quicksort 9](#_Toc103122607)

[Algorithm 9](#_Toc103122608)

[Pseudocode & Execution Example 10](#_Toc103122609)

[Space & Time Complexities 11](#_Toc103122610)

[Partial Selection Sort 12](#_Toc103122611)

[Algorithm 12](#_Toc103122612)

[Pseudocode & Execution Example 12](#_Toc103122613)

[Space & Time Complexities 13](#_Toc103122614)

[Partial HeapSort 13](#_Toc103122615)

[Algorithm 13](#_Toc103122616)

[Pseudocode & Execution Example 14](#_Toc103122617)

[Time Complexities 15](#_Toc103122618)

[QuickSelect 15](#_Toc103122619)

[Algorithm 15](#_Toc103122620)

[Pseudocode & Execution Example 15](#_Toc103122621)

[Time Complexities 17](#_Toc103122622)

[Big O Notation 18](#_Toc103122623)

[Cases of Algorithms 19](#_Toc103122624)

[Insertion Sort: 19](#_Toc103122625)

[Merge Sort: 20](#_Toc103122626)

[Quick Sort: 20](#_Toc103122627)

[Partial Selection Sort: 20](#_Toc103122628)

[Partial Heap Sort: 21](#_Toc103122629)

[Quick Select: 21](#_Toc103122630)

[Graphical Comparisons 22](#_Toc103122631)

[Insertion Sort: 23](#_Toc103122632)

[1. Short Length 23](#_Toc103122633)

[2. Medium Length 24](#_Toc103122634)

[3. Long Length 25](#_Toc103122635)

[4. Merging Charts 26](#_Toc103122636)

[Merge Sort: 28](#_Toc103122637)

[1. Short Array 28](#_Toc103122638)

[2. Medium Array 29](#_Toc103122639)

[3. Long Array 30](#_Toc103122640)

[4. Merging Charts 32](#_Toc103122641)

[Quick Sort: 33](#_Toc103122642)

[1. Short Array 33](#_Toc103122643)

[2. Medium Array 34](#_Toc103122644)

[3. Long Array 35](#_Toc103122645)

[4. Merging Charts 37](#_Toc103122646)

[Partial Selection Sort: 39](#_Toc103122647)

[1. Short Array 39](#_Toc103122648)

[2. Medium Array 40](#_Toc103122649)

[3. Long Array 41](#_Toc103122650)

[4. Merging Charts 42](#_Toc103122651)

[Partial Heap Sort: 43](#_Toc103122652)

[1. Short Array 43](#_Toc103122653)

[2. Medium Array 44](#_Toc103122654)

[3. Long Array 45](#_Toc103122655)

[4. Merging Charts 47](#_Toc103122656)

[Quick Select: 48](#_Toc103122657)

[1. Short Array 48](#_Toc103122658)

[2. Medium Array 49](#_Toc103122659)

[3. Long Array 50](#_Toc103122660)

[4. Merging Charts 51](#_Toc103122661)

[Quick Select (Median-of-Three): 53](#_Toc103122662)

[1. Short Array 53](#_Toc103122663)

[2. Medium Array 54](#_Toc103122664)

[3. Long Array 55](#_Toc103122665)

[4. Merging Charts 56](#_Toc103122666)

[Duel of Quicks: 57](#_Toc103122667)

[Conclusion 58](#_Toc103122668)

[1. Insertion Sort: 58](#_Toc103122669)

[2. Merge Sort: 58](#_Toc103122670)

[3. Quick Sort: 58](#_Toc103122671)

[4. Partial Selection Sort: 58](#_Toc103122672)

[5. Partial Heap Sort: 58](#_Toc103122673)

[6. Quick Select: 58](#_Toc103122674)

[7. Quick Select (Median-of-Three): 59](#_Toc103122675)

[References & Resources 60](#_Toc103122676)

# Algorithms and Their Explanations

## Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time.

### Algorithm

Insertion sort iterates, consuming one input element per iteration and generating a sorted output list. Insertion sort takes one element from the input data at a time, finds where it belongs in the sorted list, and inserts it there. It keeps going until there are no more input elements.

In-place sorting is done by iterating up the array and growing the sorted list behind it. It compares the value in each array position to the largest value in the sorted list. If the element is larger, it remains in place and moves on to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts them into that correct position.

### Pseudocode & Execution Example

i ← 1

while i < length(A)

    x ← A[i]

    j ← i - 1

    while j >= 0 and A[j] > x

        A[j+1] ← A[j]

        j ← j - 1

    end while

    A[j+1] ← x

    i ← i + 1

end while

### Space complexity & Time Complexities

The simplest worst case input is an array sorted in reverse order.

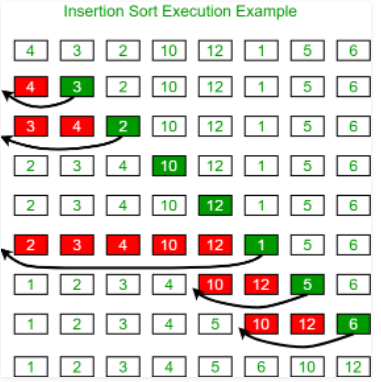
Worst-case performance → O(n2) comparisons and swaps

The best case input is an array that is already sorted. In this case insertion sort has a linear running time.

Best-case performance → O(n) comparisons, O(1) swaps

Average performance → O(n2) comparisons and swaps

Space Complexity → O(1) auxiliary



## Merge Sort

Merge sort is a divide-and-conquer algorithm.

### Algorithm

Divide array A[0..n-1] into two roughly equal halves and duplicate each half in arrays B and C.

Recursively sort arrays B and C

As follows, combine sorted arrays B and C into array A:

Repeat until neither of the arrays contains any more elements:

Compare the first elements in the arrays that haven't been processed yet.

While incrementing the index indicating the unprocessed portion of that array, copy the smaller of the two into A.

Copy the remaining unprocessed elements from the other array into A once all elements in one of the arrays have been processed.

### Pseudocode & Execution Example

**function Mergesort(A[0..n-1])**

//Sorts Array A[0..n-1] by recursive mergesort

//Input: An array A[0..n-1] of orderable elements

//Output: Array A[0..n-1] sorted in nondecreasing order

if n > 1

copy A[0..⌊n/2⌋ - 1] to B[0..⌊n/2⌋ - 1]

copy A[⌊n/2⌋..n - 1] to C[0..⌊n/2⌋ - 1]

Mergesort(B[0..⌊n/2⌋ - 1])

Mergesort(C[0..⌊n/2⌋ - 1])

Merge(B,C,A)

**function Merge(B[0..p - 1],C[0..q - 1],A[0..p + q - 1])**

//Merges two sorted arrays into one sorted array

//Input: Arrays B[0..p - 1] and C[0..q - 1] both sorted

//Output: Sorted array A[0..p + q - 1] of the elements of B and C

i ← 0; j ← 0; k ← 0;

while i < p and j < q do

if B[i] ≤ C[j]

A[k] ← B[i]; i ← i + 1

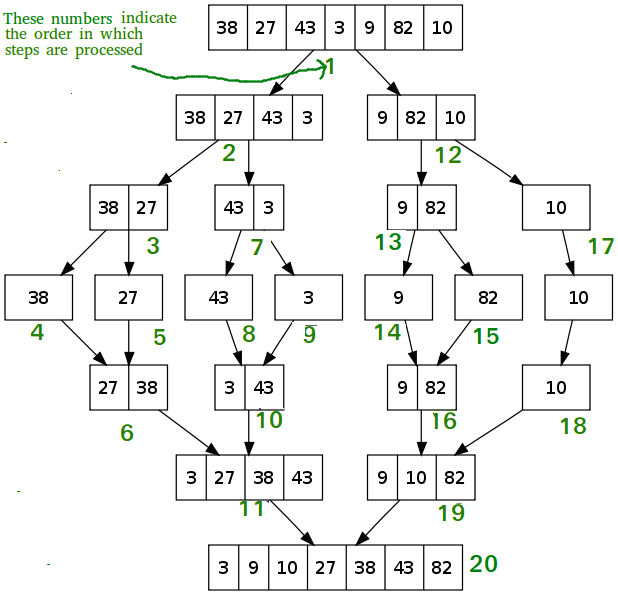
else A[k] ← C[j]; j ← j + 1

k ← k + 1

if i = p

copy C[j..q - 1] to A[k..p + q - 1]

else copy B[i..p -1] to A[k..p + q - 1]



### Space Complexity & Time Complexities

Worst-case performance → O(nlogn)

Best-case performance → O(nlogn)

Average performance → O(nlogn)

Space Complexity → O(n) auxiliary

Recurrence relation time complexity → T(n) = 2T(n/2) + θ(n)

## Quicksort

Quicksort is a divide-and-conquer algorithm.

### Algorithm

Return immediately if the range has fewer than two elements, as there is nothing else to do. A special-purpose sorting method may be used for other very short lengths, and the rest of these steps may be skipped.

Otherwise, choose a pivot value that occurs within the range.

Partition the range by reordering its elements while determining a point of division so that all elements with values less than the pivot appear before the division and all elements with values greater than the pivot appear after it; elements with values equal to the pivot can appear in either direction. Most partition routines ensure that the value that ends up at the point of division is equal to the pivot, and is now in its final position because at least one instance of the pivot is present.

Apply the quicksort recursively to the sub-range up to the point of division and the sub-range after it, possibly excluding the element equal to the pivot at the point of division from both ranges.

### Pseudocode & Execution Example

/\* This function takes the last element as pivot, places the pivot element at its correct position in a sorted array, and places all smaller (smaller than pivot)to the left of the pivot and all greater elements to the right of the pivot \*/

**function partition (arr[], low, high)**

    // pivot (Element to be placed at right position)

    pivot = arr[high];

    i = (low - 1)  // Index of smaller element and indicates the

                   // right position of pivot found so far

    for (j ← low to j ← high - 1 do{

        // If current element is smaller than the pivot

        if arr[j] < pivot{

            i++;    // increment index of smaller element

            swap arr[i] and arr[j]

  }

    }

    swap arr[i + 1] and arr[high])

    return (i + 1)

/\* low  --> Starting index,  high  --> Ending index \*/

**function quickSort(arr[], low, high)**

    if (low < high){

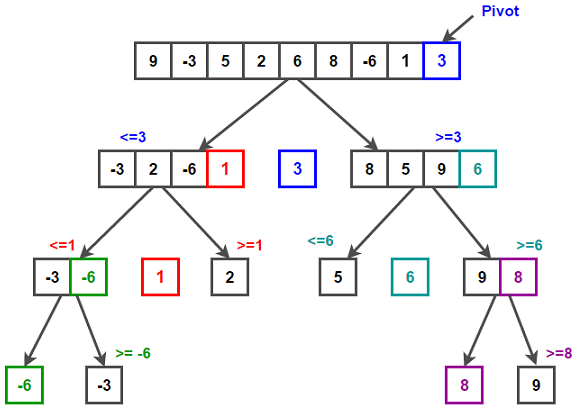
  // pi is partitioning index, arr[pi] is now at right place

        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi

    }

****

### Space & Time Complexities

In general, time complexity can be written as follows.

T(n) = T(k) + T(n-k-1) + θ(n)

Worst-case performance → O(n2)

The worst-case scenario is when the partitioning process always chooses the largest or smallest element as the pivot. When using the above partition strategy, where the last element is always chosen as a pivot, the worst-case scenario is when the array is already sorted in ascending or descending order. Time complexity becomes O(n2).

T(n) = T(0) + T(n-1) + θ(n) which is equivalent to  T(n) = T(n-1) + θ(n)

Best-case performance → O(nlogn)

The best case occurs when the partition process always picks the middle element as the pivot.

T(n) = 2T(n/2) + θ(n)

Average performance → O(nlogn)

T(n) = T(n/9) + T(9n/10) + θ(n)

Worst-case Space Complexity → O(n) auxiliary

When the algorithm encounters its worst-case where for getting a sorted list, we need to make n recursive calls so O(n) is the worst space complexity.

Average case Space Complexity → O(logn)

## Partial Selection Sort

Selection sort is a Brute-Force algorithm.

### Algorithm

The selection sort algorithm sorts an array by repeatedly finding the smallest element from the unsorted part and placing it at the beginning (in ascending order). In a given array, the algorithm keeps two subarrays.

1) A subarray that has already been sorted.

2) The unsorted subarray that remains.

In each iteration of the selection sort, the unsorted subarray's minimum element (in ascending order) is chosen and moved to the sorted subarray.

### Pseudocode & Execution Example

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]

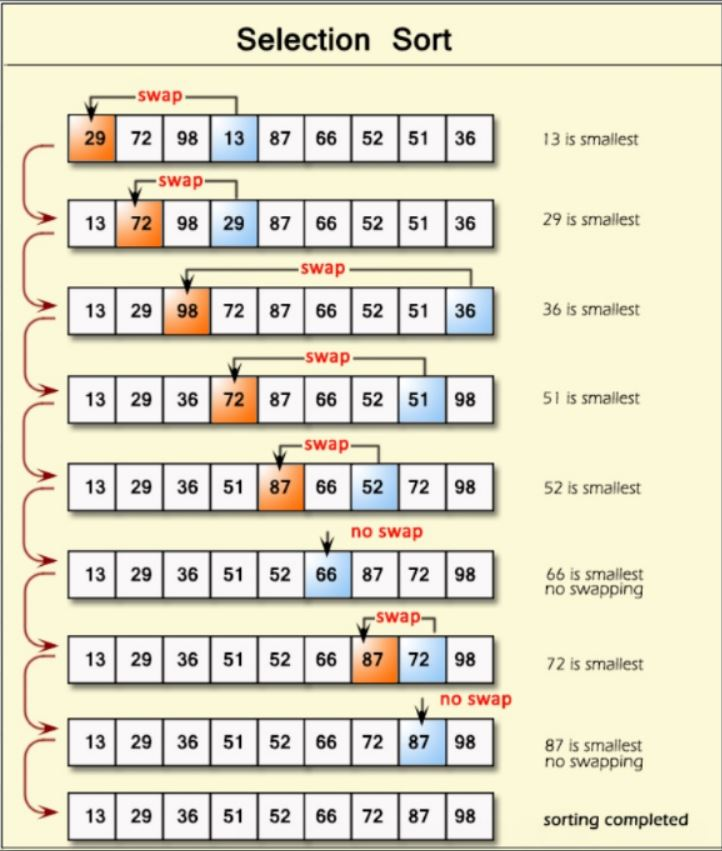
                swap(arr[i] and arr[minIndex]

        }

    }

    return arr[k]

}



### Space & Time Complexities

Worst-case performance → O(k\*n) comparisons

Best-case performance → O(k\*n) comparisons, O(1) swaps

Average performance → O(k\*n) comparisons

Space Complexity → O(1) auxiliary.

## Partial HeapSort

Heapsort is a comparison-based sorting algorithm.

### Algorithm

The partial heap sort algorithm is a simple algorithm consisting of 2 steps.

1- Create max-heap with the given array

2- Perform (n-k) root removal

As a result, the new root of the max-heap (ie the 0th index of the array) becomes the kth smallest element.

### Pseudocode & Execution Example

**Heapsort**(A) {

   BuildHeap(A)

   for i <- length(A) downto k - 1 {

      rootRemoval(A, n)

}

**BuildHeap**(A) {

   heapsize <- length(A)

   for i <- floor( length/2 ) downto 1

      Heapify(A, i)

}

**Heapify**(A, i) {

   largest <- i

   le <- 2i + 1

   ri <- 2i + 2

   if (le<=heapsize) and (A[le]>A[i])

      largest <- le

   if (ri<=heapsize) and (A[ri]>A[largest])

      largest <- ri

   if (largest != i) {

      exchange A[i] <-> A[largest]

      Heapify(A, largest)

   }

}

**rootRemoval**(A, n) {

   A[0] <- A[n-1]

Heapify(A, 0, n-1)

}

### Time Complexities

Worst-case performance → O(nlogn)

Best-case performance → O(nlogn)

Average performance →  O(nlogn)

## QuickSelect

Quickselect is a selection sort algorithm.

### Algorithm

Partition function similar to quick sort

Consider the last element as a pivot and add elements with less value to the left and high value to the right and also change the pivot position to its respective position in the final array.

Find partition part as above then,

If the index of the partitioned element is more than k, then we recur for the left part. If the index is the same as k, we have found the kth smallest element and we return. If the index is less than k, then we recur for the right part.

### Pseudocode & Execution Example

**function partition(list, left, right, pivotIndex) is**

    pivotValue := list[pivotIndex]

    swap list[pivotIndex] and list[right]  *// Move pivot to end*

    storeIndex := left

    for i from left to right − 1 do

        if list[i] < pivotValue then

            swap list[storeIndex] and list[i]

            increment storeIndex

    swap list[right] and list[storeIndex]

*// Move pivot to its final place*

    return storeIndex

*// Returns the k-th smallest element of list within left..right inclusive*

*// (i.e. left <= k <= right).*

**function select(list, left, right, k)** is

    if left = right then   *// If the list contains only one element,*

        return list[left]  *// return that element*

    pivotIndex  := ...     *// select a pivotIndex between left        and right*

*// e.g.,* left + floor(rand() % (right − left + 1))

    pivotIndex  := partition(list, left, right, pivotIndex)

*// The pivot is in its final sorted position*

    if k = pivotIndex then

        return list[k]

    else if k < pivotIndex then

        return select(list, left, pivotIndex − 1, k)

    else

        return select(list, pivotIndex + 1, right, k)

**In addition to quick select if we add median of three pivot selection pseudocode is,**

**function medianOfThree(A,lo,hi)**

mid := ⌊(lo + hi) / 2⌋

if A[mid] < A[lo]

    swap A[lo] with A[mid]

if A[hi] < A[lo]

    swap A[lo] with A[hi]

if A[mid] < A[hi]

    swap A[mid] with A[hi]

pivot := A[hi]

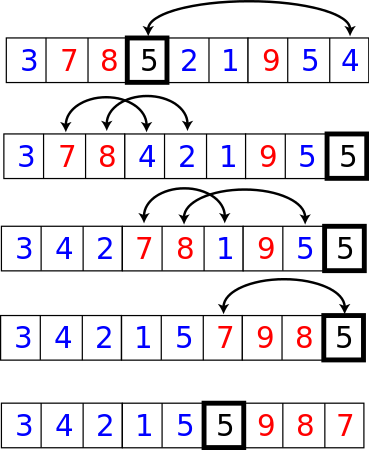
//Median of three pivot selection decreases the chances of worst case happening in quickselect.

### Time Complexities

Worst-case performance → O(n2)

Best-case performance → Ω(n)

Average performance → θ (n)



# Big O Notation

Big O notation is used in computer science to categorize algorithms based on how their run time or space needs to expand as the input size grows. The remainder term in the prime number theorem is a prominent example of such a difference. In analytic number theory, big O notation is sometimes used to indicate a bound on the difference between an arithmetical function and a better-understood approximation. Many other fields employ Big O notation to generate similar estimations.

Different functions with the same growth rate can be described using the same O notation in Big O notation. The letter O is utilized since a function's growth rate is also known as the function's order. A large O notation description of a function usually only offers an upper constraint on the function's growth rate.

The best, worst, and average cases of a given algorithm in computer science represent the resource utilization at the very least, at the very most, and on average, respectively. Running time, or time complexity, is usually the resource in question, but it might potentially be memory or another resource. The function that takes the fewest steps on n elements of input data is the best case. The function that takes the most steps on input data of size n is the worst case. The average case is a function that takes an average number of steps on n elements of input data.

**Best Case:** In computer science, best-case performance refers to an algorithm's behavior under ideal conditions. For example, when the desired element is the first element in a list, the best-case scenario occurs.

Most academic and commercial enterprises are more concerned with improving average-case complexity and worst-case performance than with best-case performance. Algorithms can also be easily modified to have a good best-case running time by hard-coding solutions to a finite set of inputs, rendering the metric almost meaningless.

**Worst Case:** The worst-case complexity (typically stated in asymptotic notation) in computer science is a measure of the resources (e.g. running time, memory) that an algorithm requires given an arbitrary size input (often denoted as display style). It provides an upper bound on the algorithm's resource requirements.

In the case of running time, the worst-case time-complexity specifies the algorithm's longest-running time given any input of size display style, and so ensures that the method will complete inside the specified time frame. When comparing the efficiency of two algorithms, the order of growth (e.g. linear, logarithmic) of the worst-case complexity is usually utilized.

**Average Case:** The average-case complexity of an algorithm is the amount of a computational resource consumed by the algorithm, averaged over all possible inputs, according to computational complexity theory. It's sometimes compared with worst-case complexity, which considers the algorithm's highest complexity across all conceivable inputs.

While some problems may be intractable in the worst-case scenario, the inputs that cause this behavior are unlikely to occur in practice, so average-case complexity may be a better indicator of an algorithm's effectiveness.

## Cases of Algorithms

In this section, we will explain with examples what types of inputs enter which scenarios of the algorithms in order to prove in which situations the 7 distinct sorting algorithms in the report are more advantageous.

### Insertion Sort:

If and only if the array is already sorted, the insertion sort method works best. The algorithm is generally implemented by inserting the ith index into the indexes between [0, i - 1]. When the operation arrives at an already sorted array, however, no indexes need to be changed.

As a result, while this algorithm's best case is stored in a sorted array, the worst case is stored in a reverse sorted array (because it will make the most comparisons on a reversely sorted array).

### Merge Sort:

In terms of best, worst, and average time complexity, the merge sort method has the same value. As a result, distinguishing between which is the best and which is the worst will be tough, but this is the best version of a merge sort provided the given input is given in such a way that the two split arrays become directly merged when they are already sorted. Because it makes the fewest number of comparisons. However, if such an array is given, and you have to compare n elements in the array with n elements in the other array one by one in order to combine the two arrays that are divided, you will encounter the worst-case scenario of merge-sort. However, this is extremely unlikely to occur and has no bearing. As a result, its speed remains constant regardless of the input format.

### Quick Sort:

Although the Quick-Sort algorithm is commonly faster than other algorithms, it can rarely slow down to the point where the application crashes or memory is exhausted. If the array given to the quick sort method is already sorted or very close to being sorted, there is no doubt that you will receive a StackOverflowError error.

In short, any array is the best and average situation, while an ordered array is the worst-case scenario.

### Partial Selection Sort:

To be honest, this was the algorithm that presented the greatest obstacle during the project. Because the algorithm is really extremely slow.

The Partial Selection Sort algorithm, like the Insertion Sort method, is sorted and reverse sorted array algorithm. We identify the minimum element in each loop in the selection sort and transfer it to the beginning of the array while trying to find the proper spot between [0, I - 1] in the insertion sort. This implies that there are n2 comparisons for each loop.

The only benefit of being partial is that the smaller the k, the faster the operation is because finding the least value k times is enough.

### Partial Heap Sort:

Heap Sort and merge sort has the same character. In heap sort, there is no clear separation between the best and worst scenarios. Because both are processing at the same rate, the shape of the provided array is irrelevant.

### Quick Select:

Quick Select, shares the same personality as quicksort, as well as the almost same name. Because they both use partition logic, a StackOverflowError will be triggered and the application will crash if the pivot selection is picked worst for the array (which is valid for a sorted array for a quick choose method in its default state).

If bad pivots are consistently chosen, such as decreasing by only a single element each time, then worst-case performance is quadratic. 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 so quickselect has almost certain O(n) complexity.

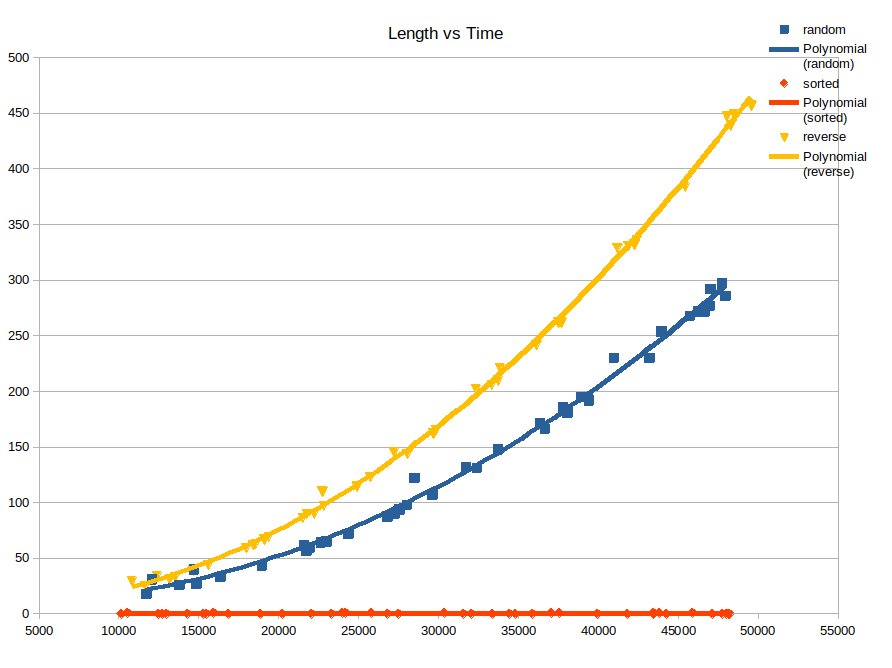
The seven algorithms discussed in this study will be compared. To make this comparison, we selected a variety of datasets, divided them into sections based on the length of the array, the kind of array (random, sorted, reverse ordered), and the position of the k value, and tested each one separately. For each sorting algorithm, we used a total of 120 distinct inputs. We compared some methods with enormous data sets because they are quite fast.

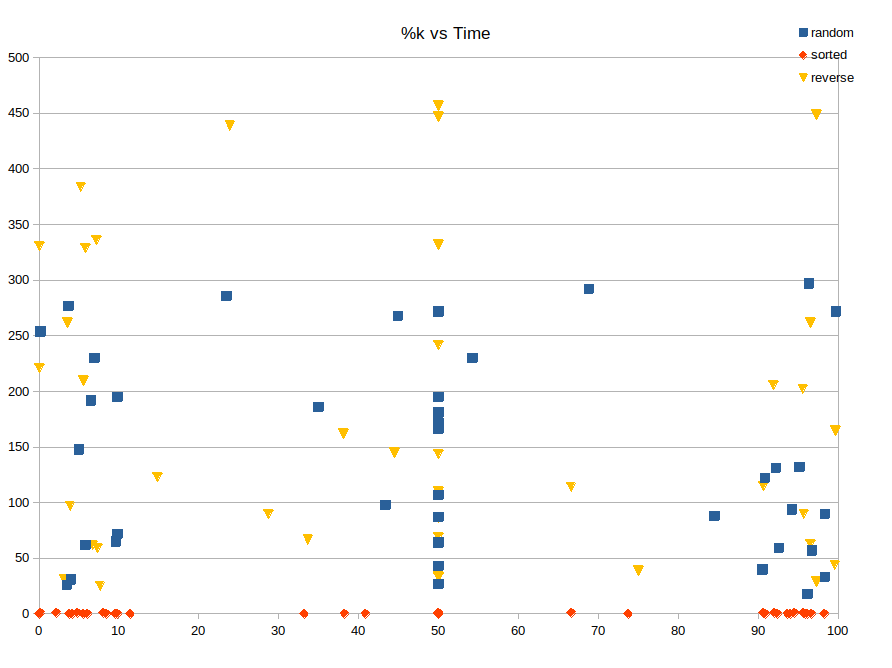
# Graphical Comparisons

In this section, we will make comparisons of 120 different arrays for each algorithm, graphed according to the lengths and positions of their k values. Arrays are divided into short (10k-50k long), medium (50k-250k long), and long (250k-1.25m long), k values are less than 10% of the length, greater than 90%, equal to 50%, and randomly divided into 4. Arrays are also divided into 3 types, random arrays, already sorted, and reverse sorted arrays.

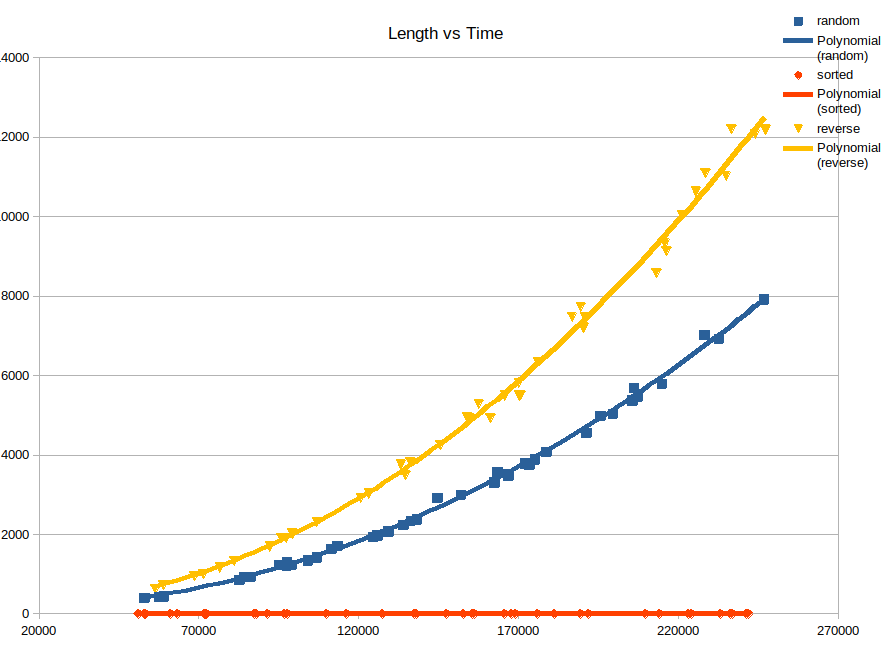
## Insertion Sort:

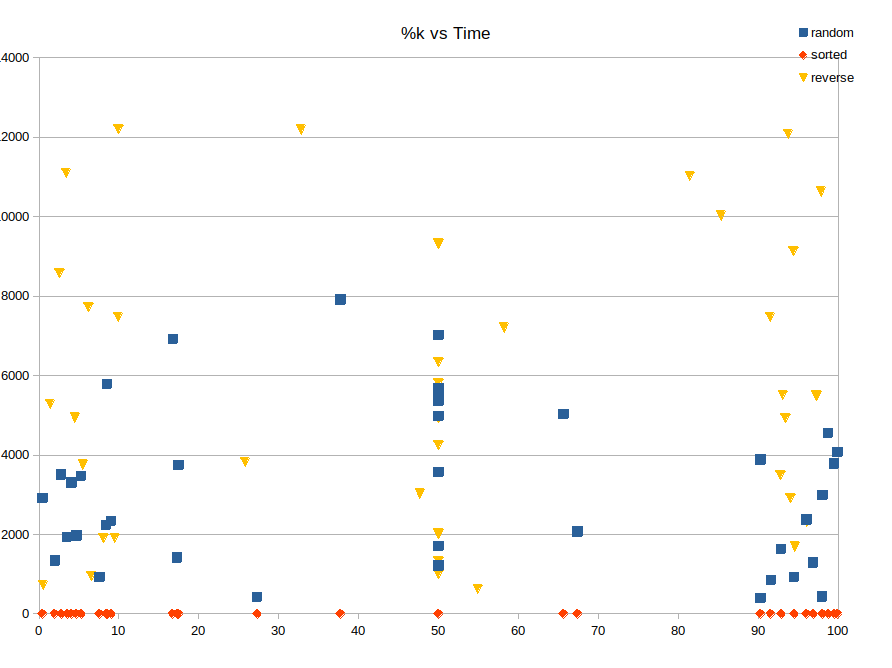
### Short Length



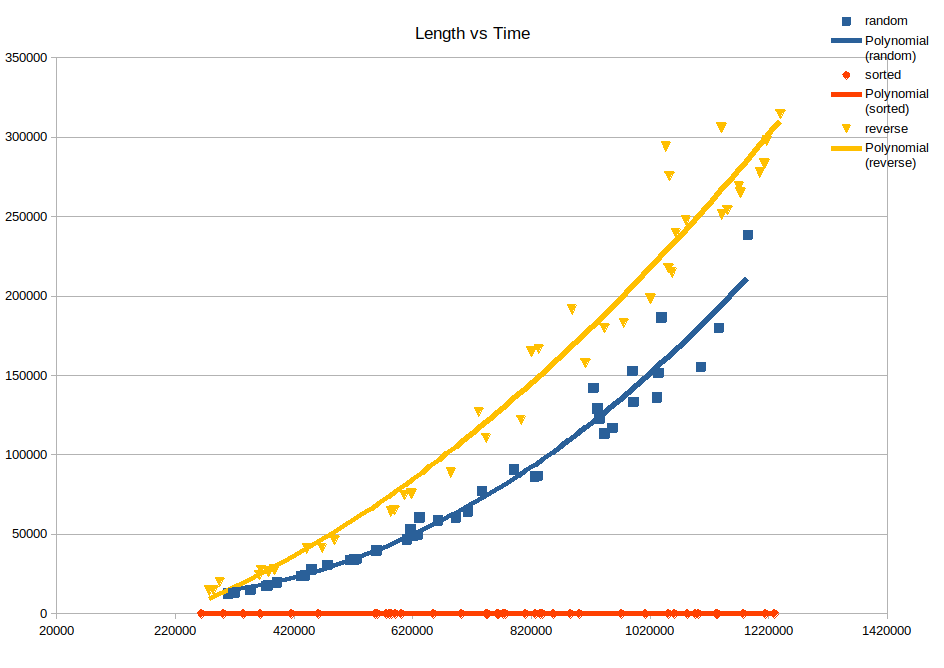


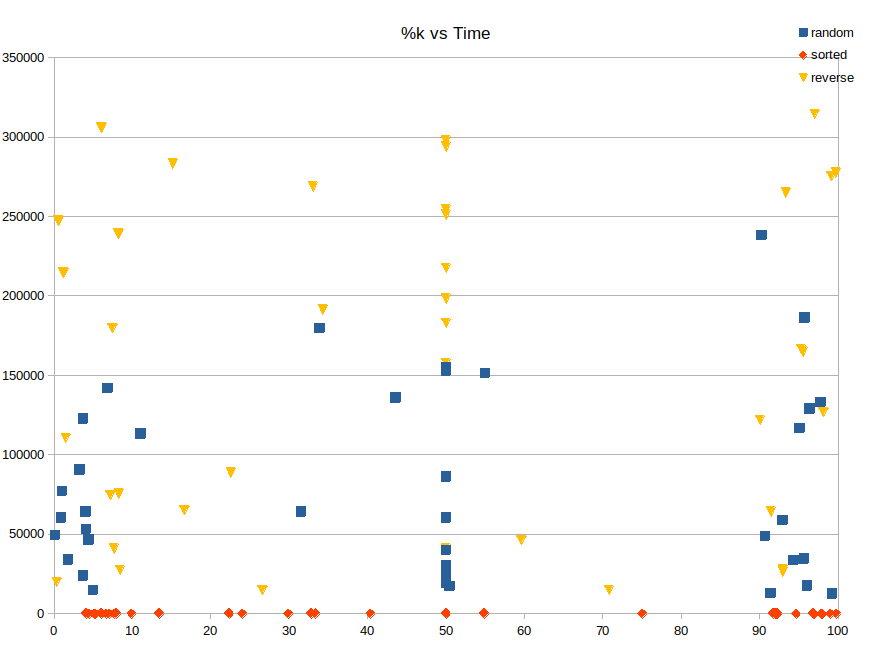
### Medium Length





### Long Length

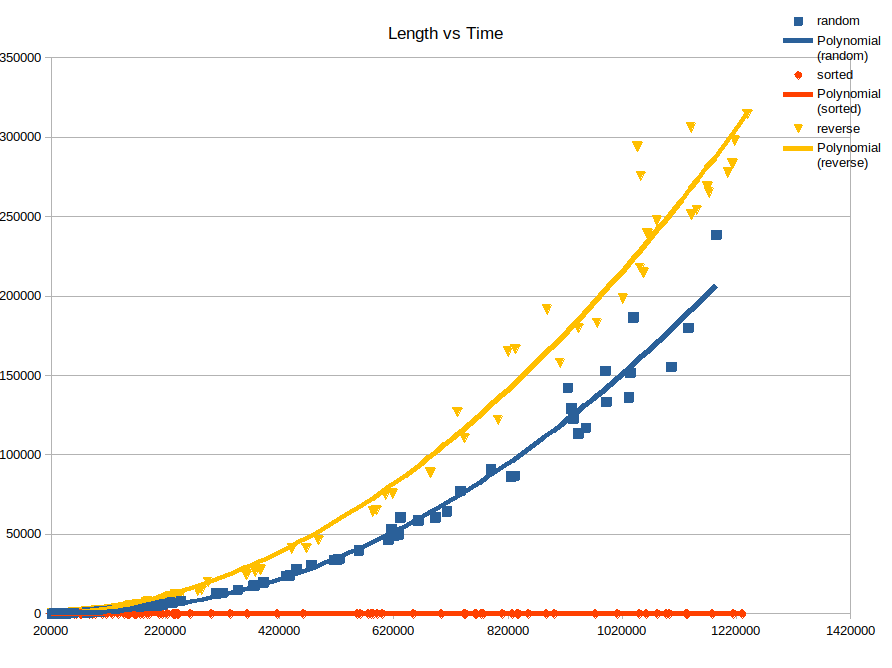


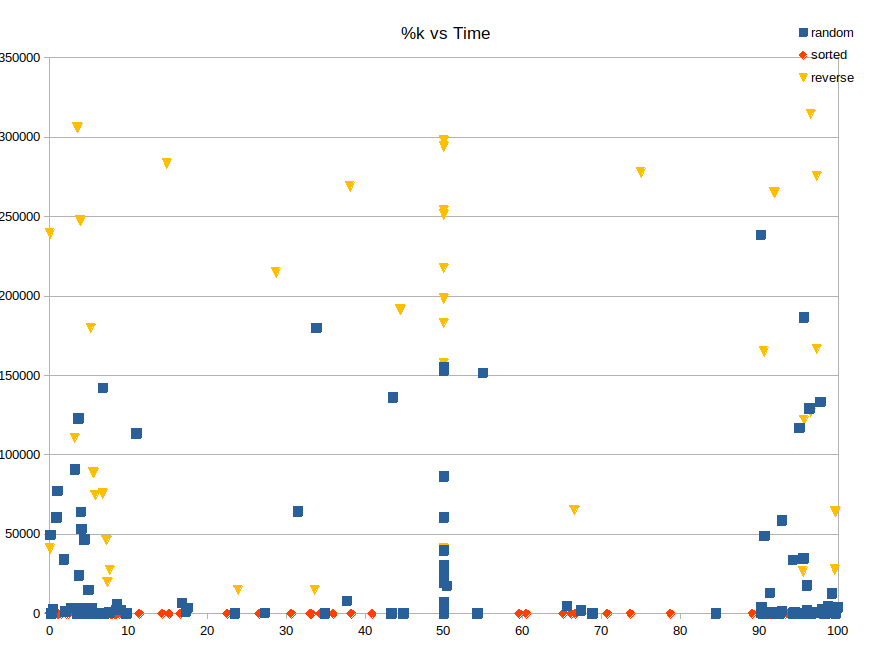


Insertion sort algorithm, as seen in the graphics, and as we mentioned in the case section above, regardless of the range of the length of the array, the reverse sorted array takes the longest time, then the random array, and finally the sorted array, which returns results in almost 0ms.

As we can see in the %k vs Time graphs, there is no noticeable change in the speeds according to the position of k, since it is not a partial algorithm and it must first sort the entire array to return the result.

### Merging Charts

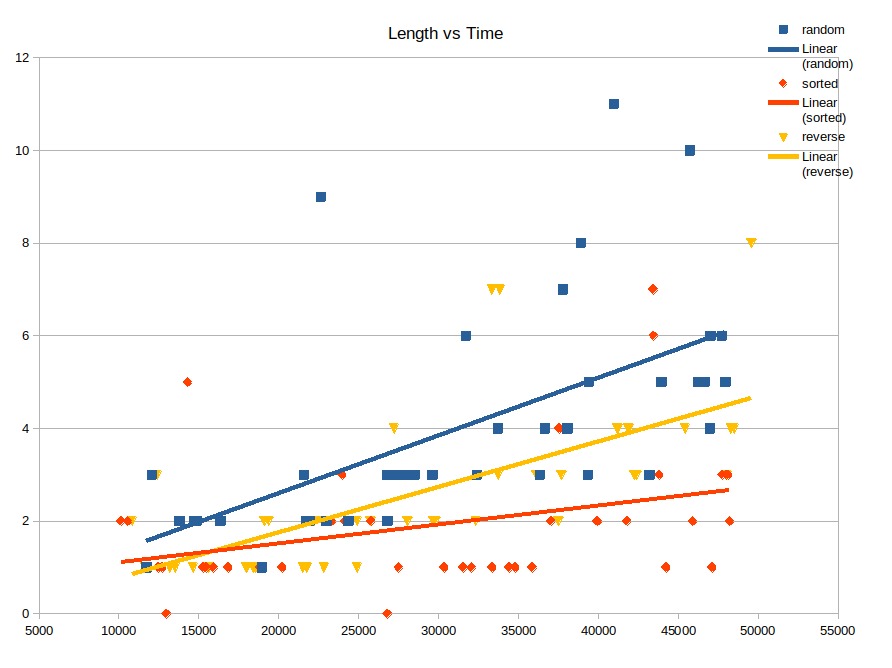


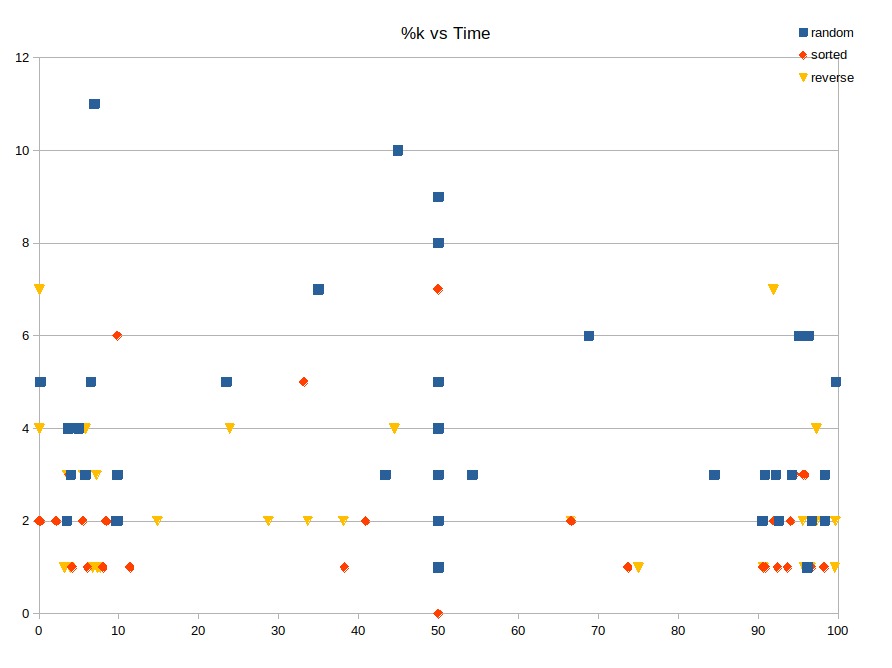


Insertion sort algorithm, as we mentioned in the first part of the report, is an algorithm that works at O(n2) in the worst and average cases, and O(1) in the best cases. When we examine the graph, an increasing exponential movement is seen in reverse and average cases, while a constant speed is seen in the best case. In other words, the arrays tested with theoretical values confirm each other.

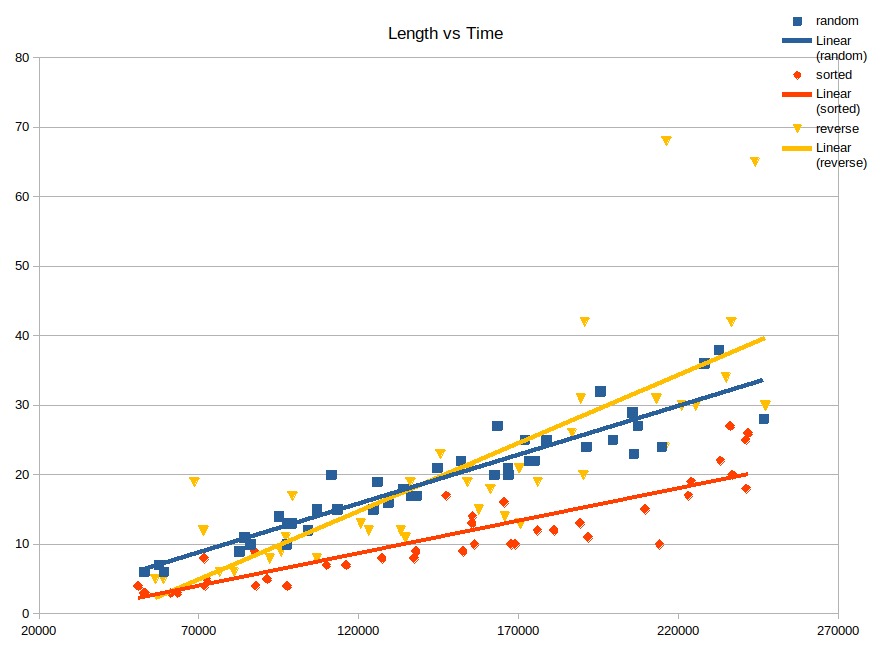
## Merge Sort:

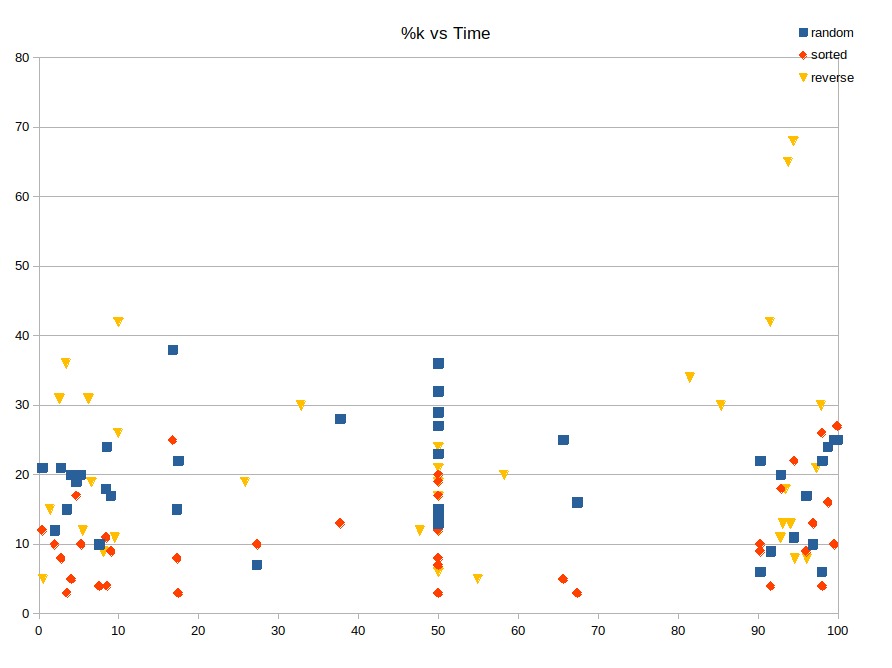
### Short Array



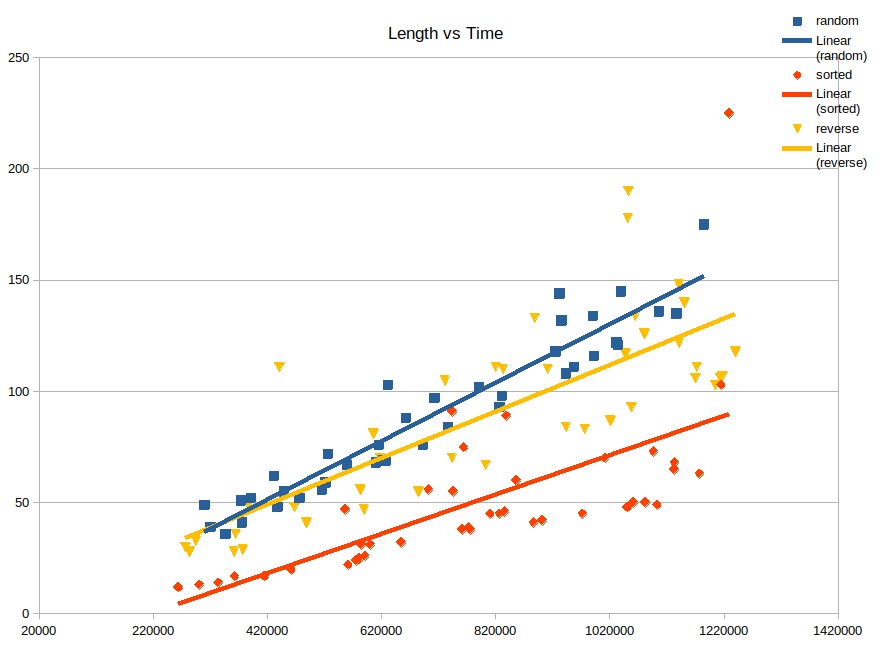


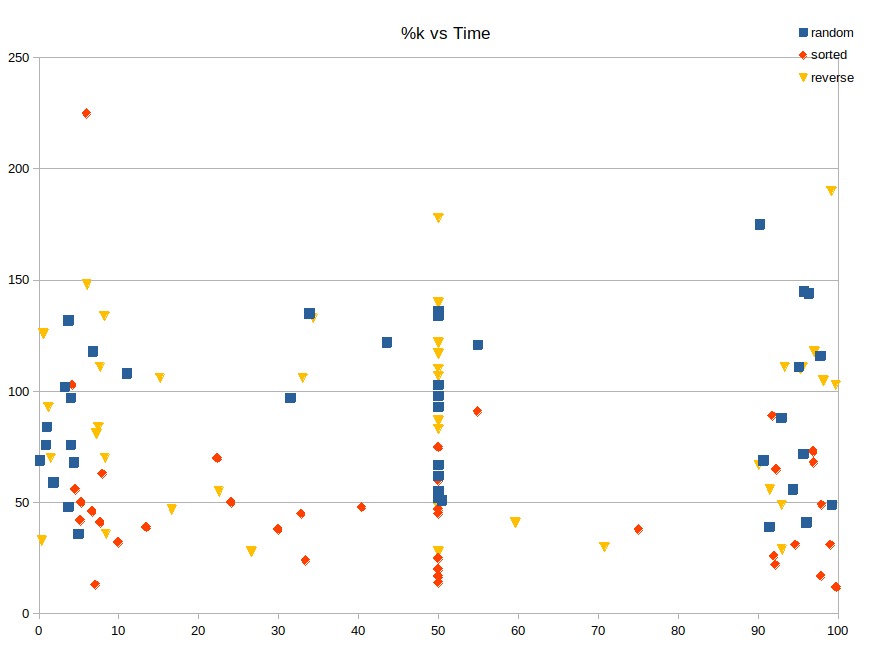
### Medium Array





### Long Array



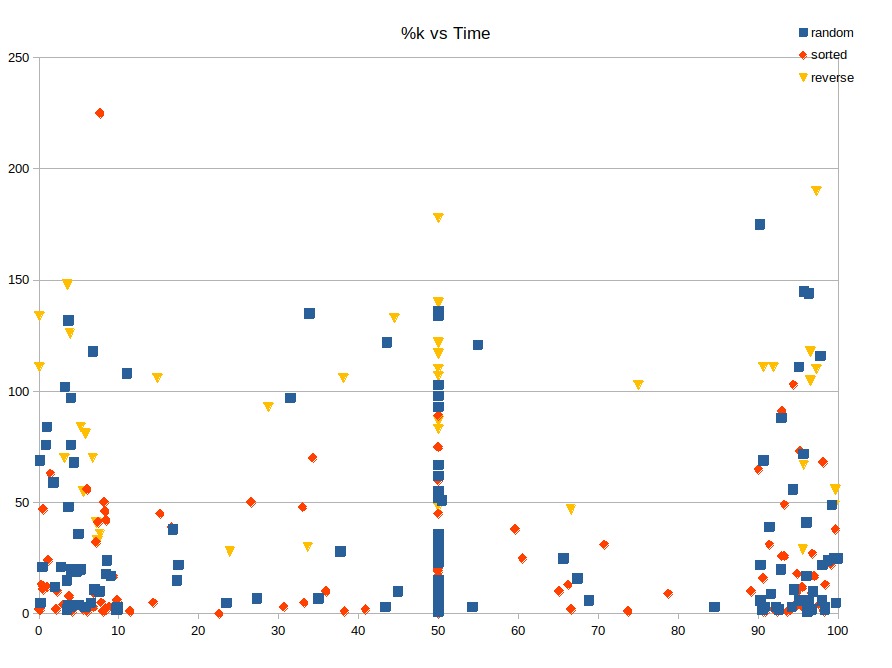


Since there is not much difference between the times, we can say that the base time complexities of these algorithms are the same. But as we can see, we can say that the fastest version is in sorted array. Since the 2 arrays it will merge are already sorted, it makes the least number of comparisons.

As we can see in the %k vs Time graphs, there is no noticeable change in the speeds according to the position of k, since it is not a partial algorithm and it must first sort the entire array to return the result.

### Merging Charts

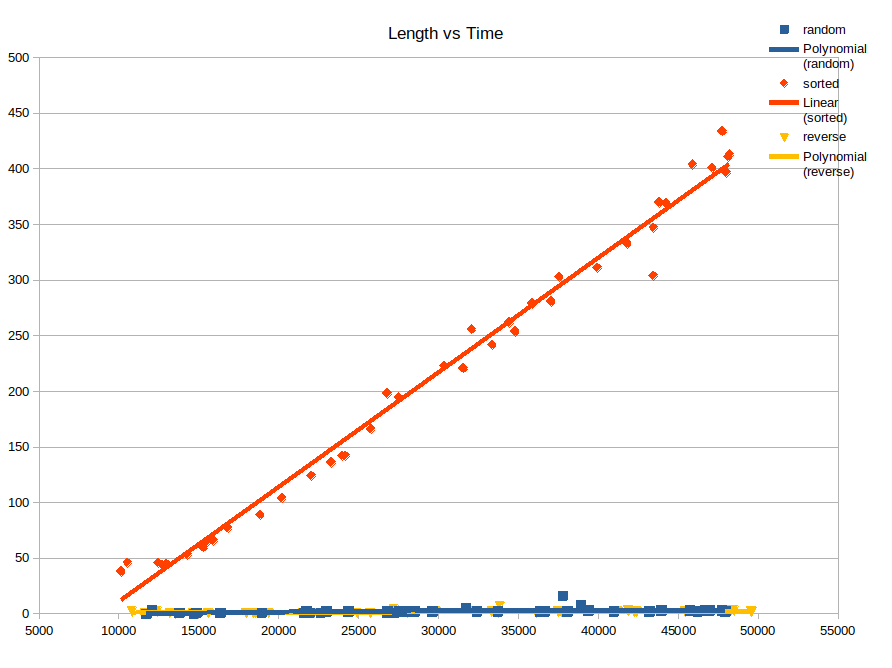
****

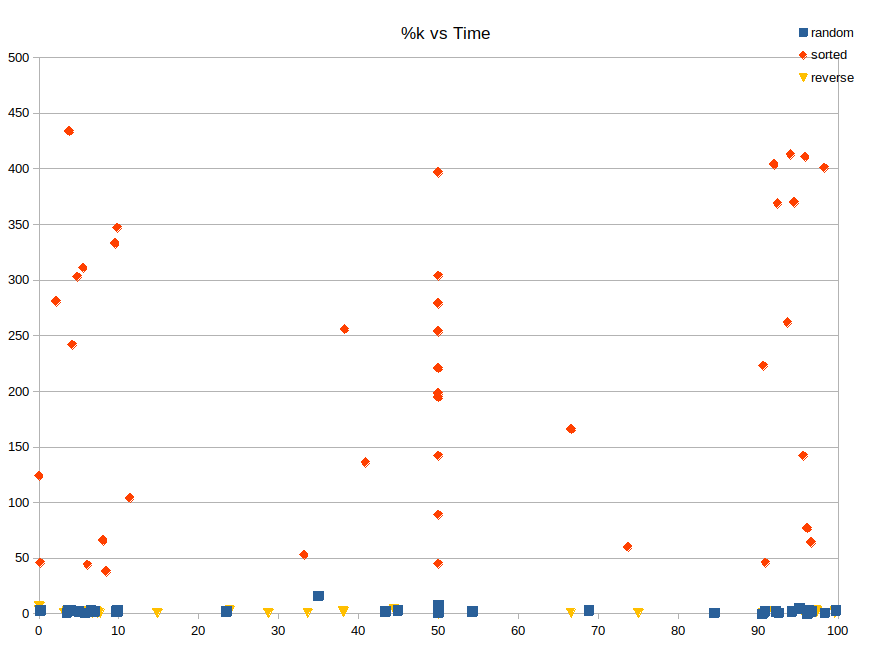
****

Merge sort algorithm, as we mentioned in the first part of the report, is an algorithm that works at O(nlogn) in every case. When we examine the graph, we can notice that the times are very close to each other.From this we can say that the base time complexity **(**O(nlogn)**)**is equal in each case. In other words, the arrays tested with theoretical values confirm each other.

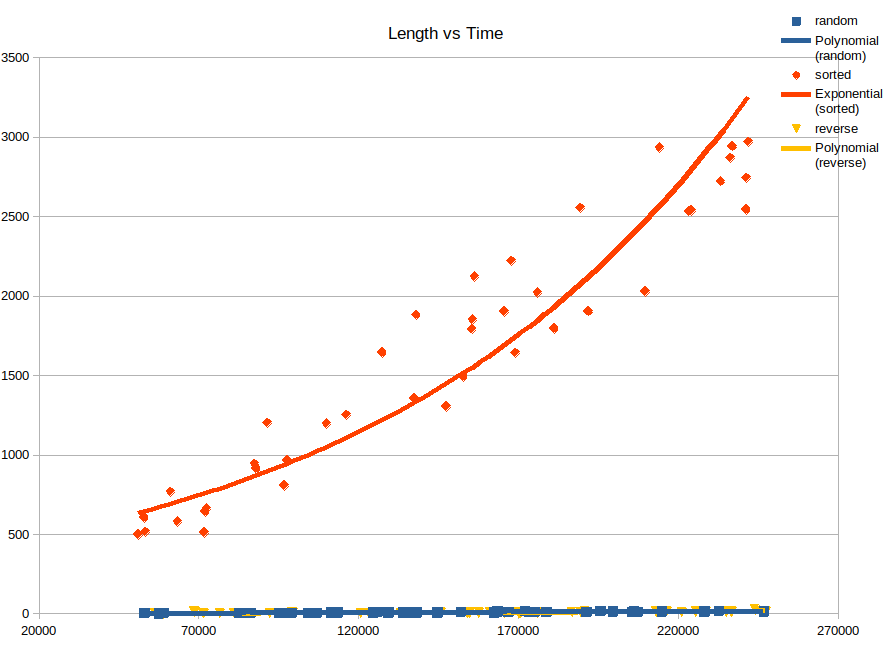
## Quick Sort:

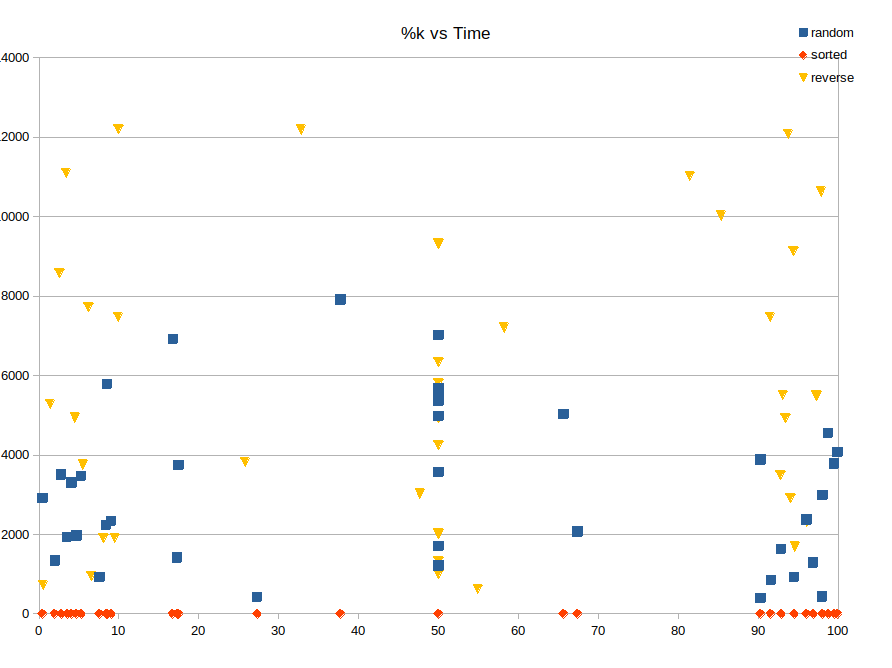
### Short Array



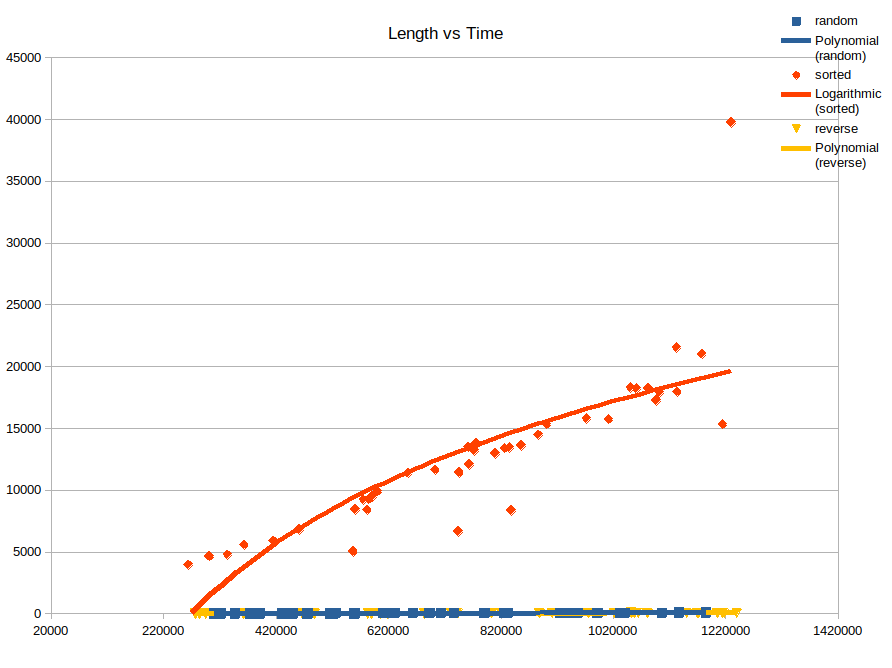


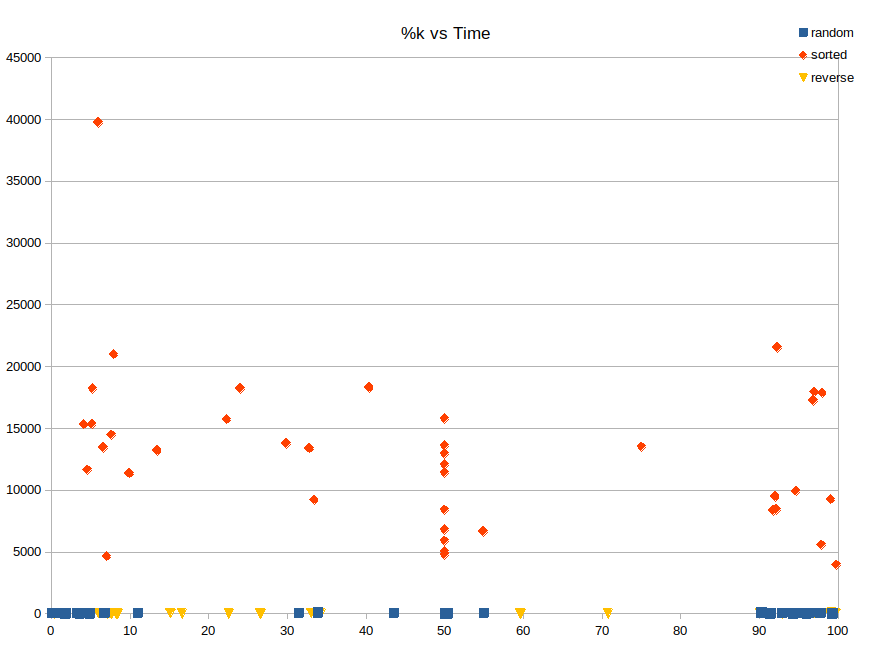
### Medium Array





### Long Array

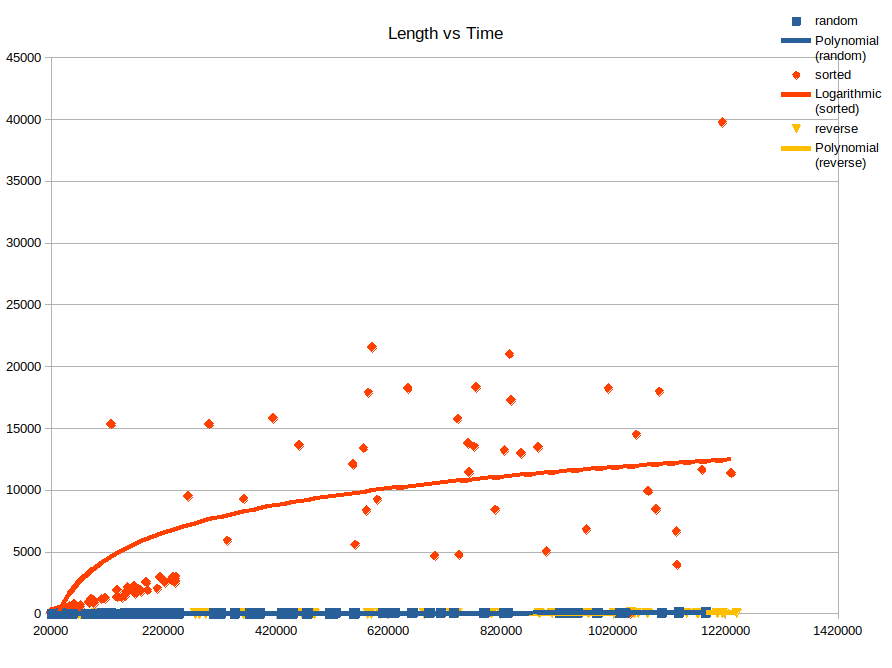


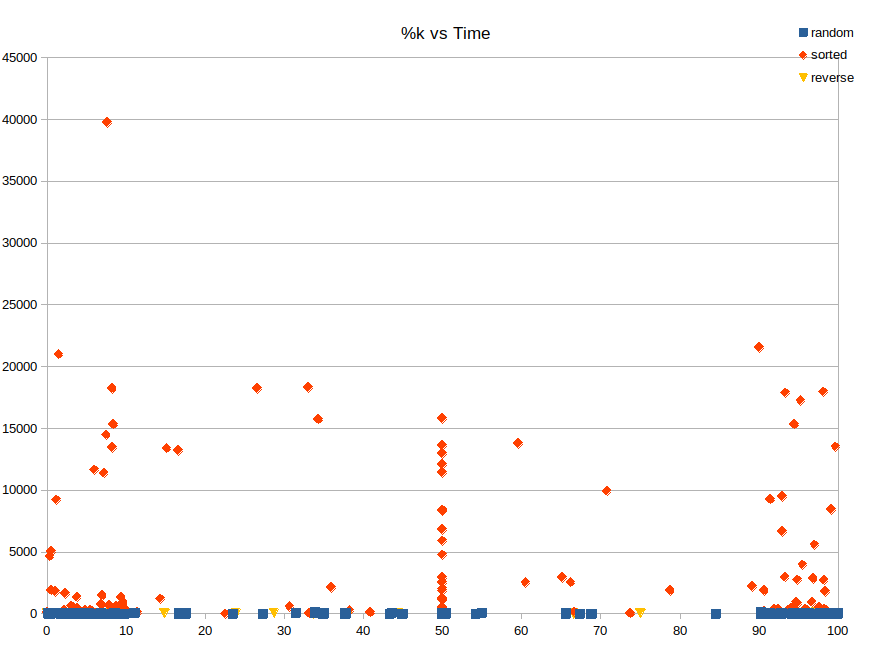


Although Quick Sort is a very fast sorting algorithm that truly lives up to its name, it is a very slow algorithm because of the slowdown of the partition algorithm in arrays that are already sorted or very close to being sorted. As can be seen in the graphics, it worked very slowly in the sorted array.

When we make a length comparison, we can see that the given length does not change much for random and reverse sorted arrays. However, the random array has one difference from the reverse sorted array, that sometimes there are already sorted sections in a random array. That's why we can rarely see that it works slower than a reverse sorted array.

### Merging Charts





At the beginning of the report, we said that the quicksort runs at O(n2) speed in the worst case, and in the other part of the report, the worst-case consists of arrays that are sorted or very close to being sorted. However, the graphs show us a logarithmic rise, not an exponential rise.

This may be because of the nightmare of all algorithms using the partition method: StackOverflowError, the one with the error message, not the site. StackOverflowError is an error we get when too many recursive functions are called and there is no more space in java's account. Since the partition algorithm progresses very slowly in the sorted array, every algorithm that uses quicksort and another same kind of partition may face this error and there is no way to solve it.

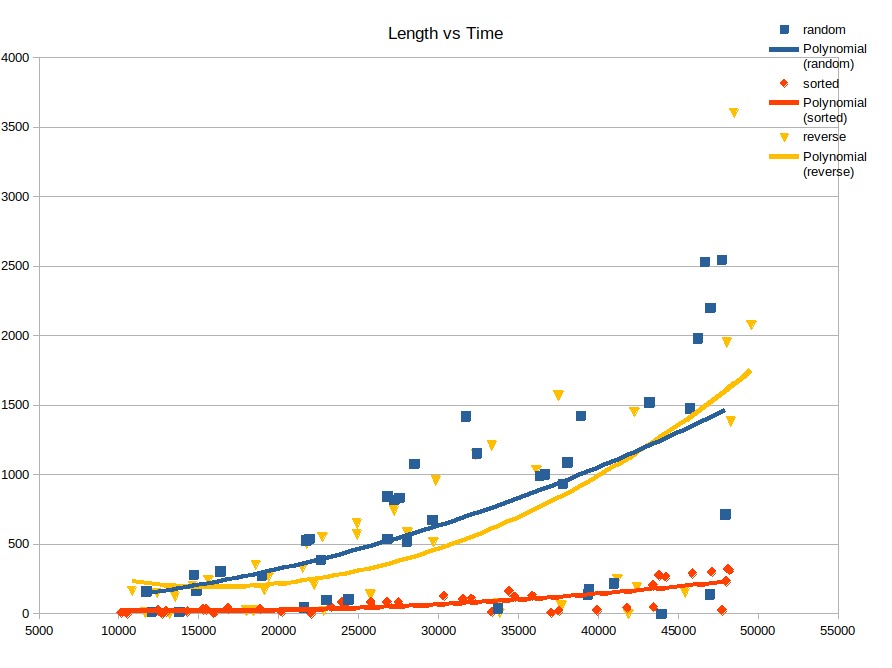
That's why we couldn't do anything but stop the program from getting corrupted with try-catch commands, and then sort it using the Arrays.sort() built-in function. In other words, when the length of the array gets longer, it's not really quicksorted, it's the Arrays.sort() function.

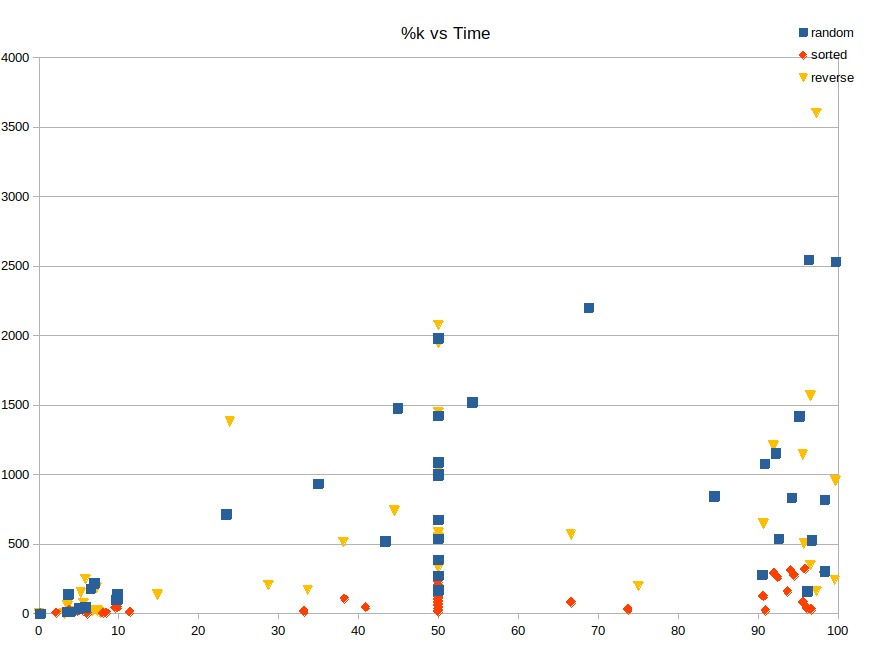
In fact, if you look at the Length vs Time chart in the merging charts section, there is 1 red dot at 40000 ms. The array, which somehow got rid of the error, was sorted with quicksort and the result was returned. If all of them could avoid getting errors, the results would be that high and an exponential graph would appear.

We can clearly see in all graphs that the %k value does not mean anything for an algorithm like quicksort.

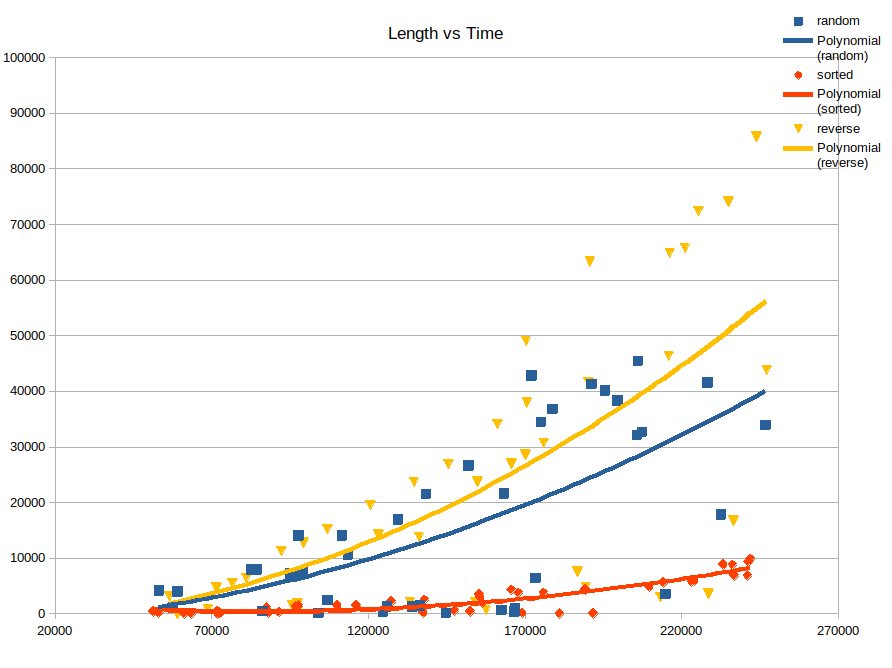
## Partial Selection Sort:

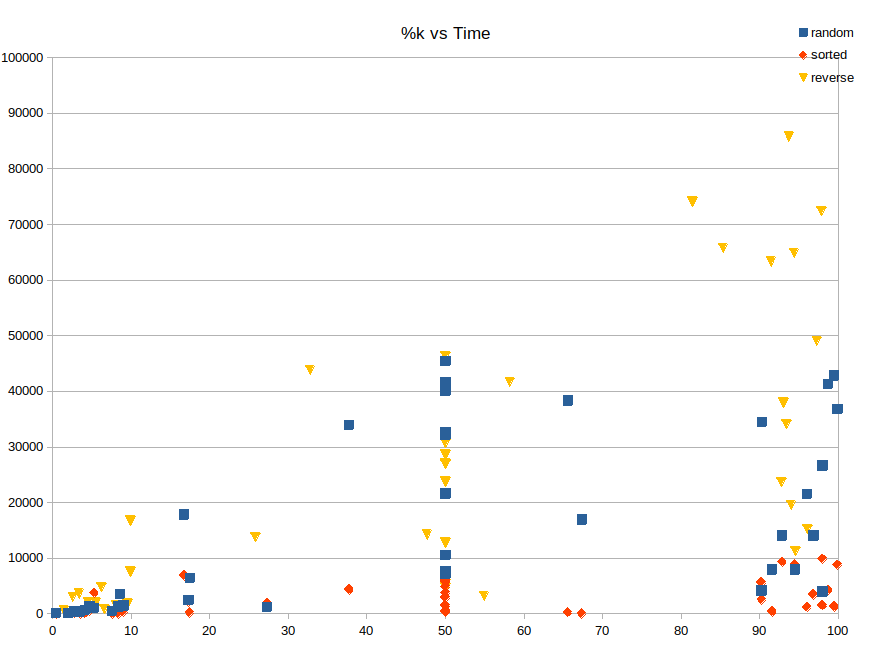
### Short Array



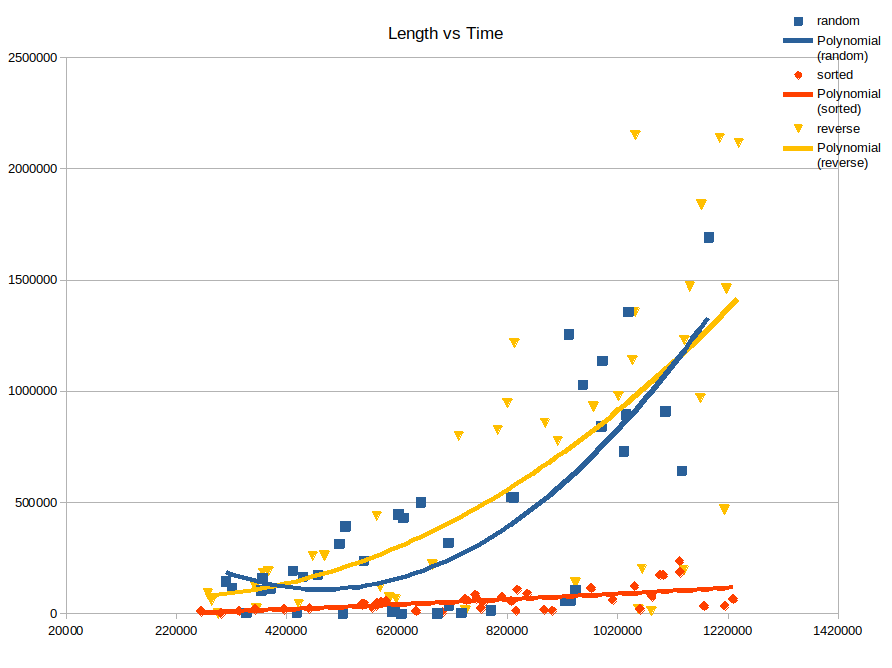


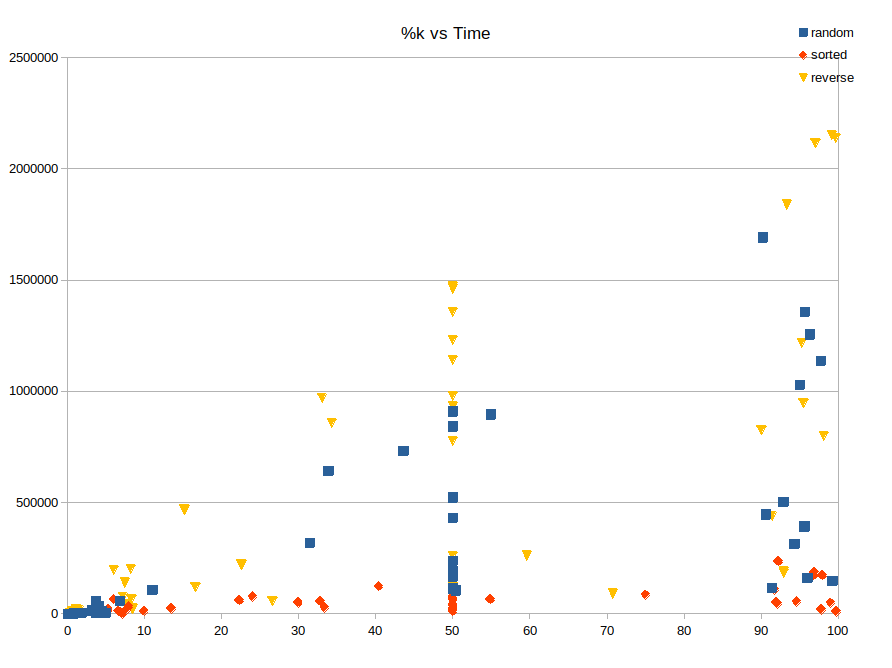
### Medium Array



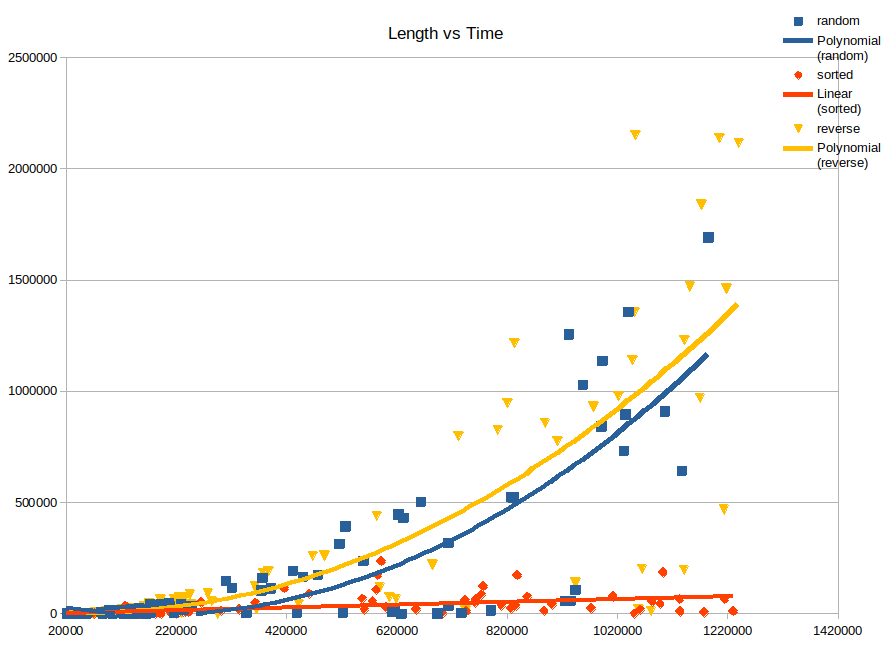


### Long Array



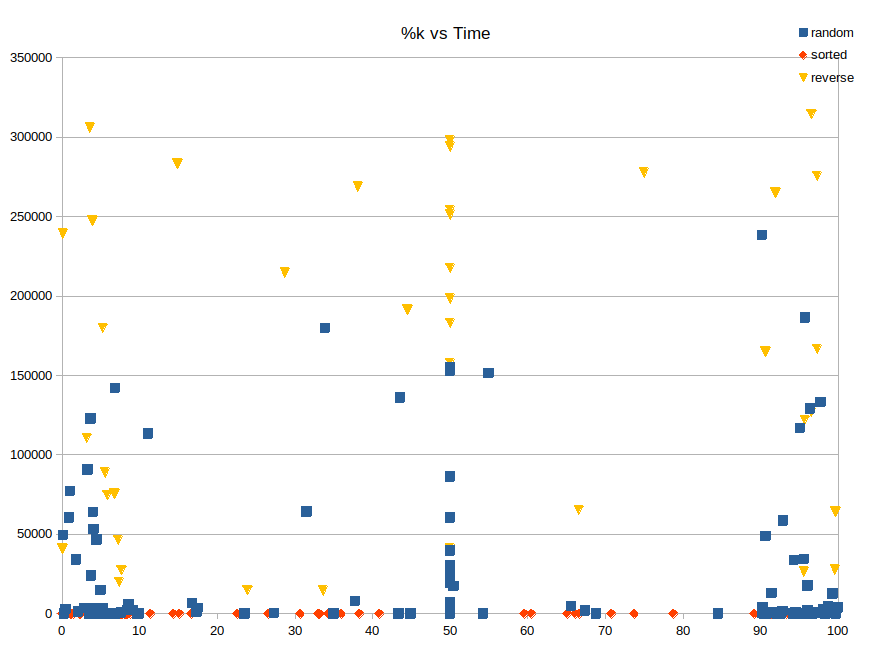


### Merging Charts



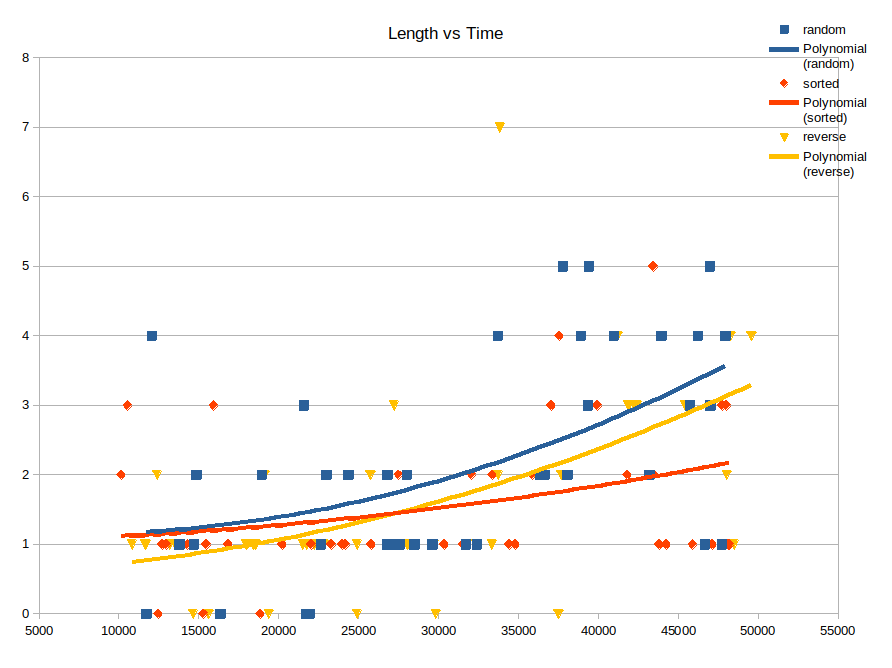
Selection sort algorithm is an algorithm that has the same alignment in worst and best cases and does not clearly have a best or worst-case scenario. But as we can see from the graphs, when it is done as a partial selection sort, a recognizable best case emerges in the sorted array.

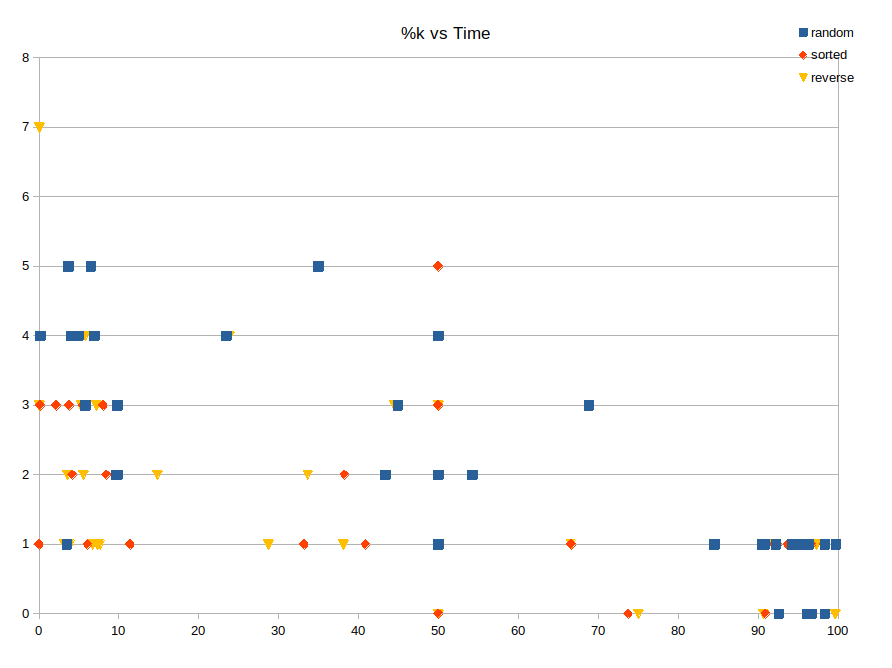
As we can see from the %k vs time graph, the compile time increases as the percentage of k increases. Because of the the number of elements to be compared in the array increases.



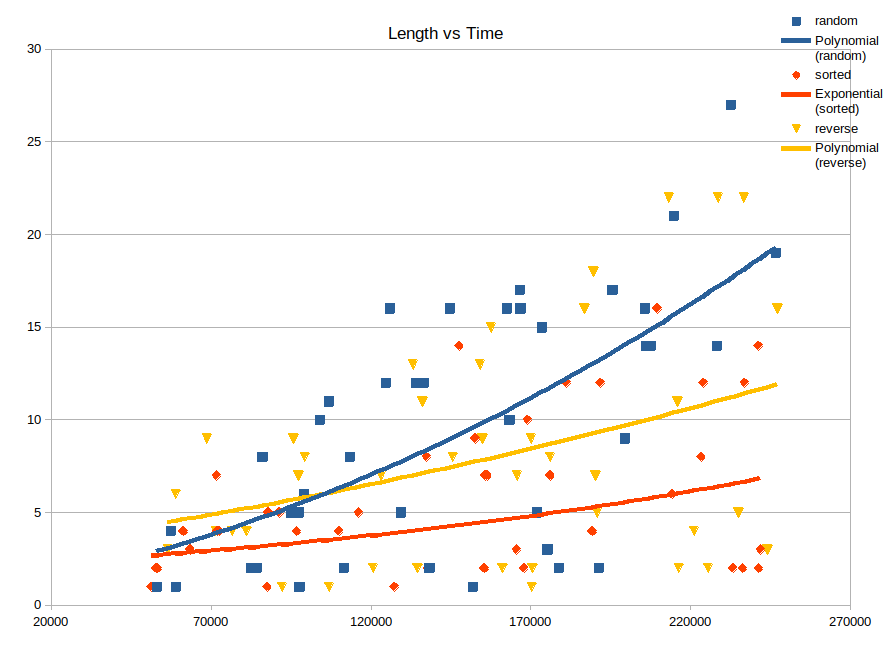
## Partial Heap Sort:

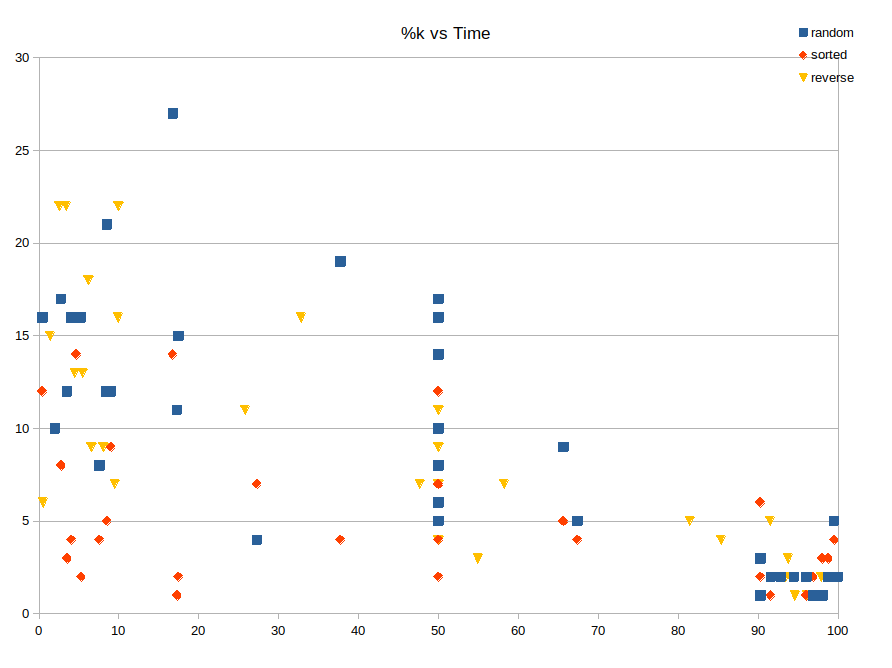
### Short Array



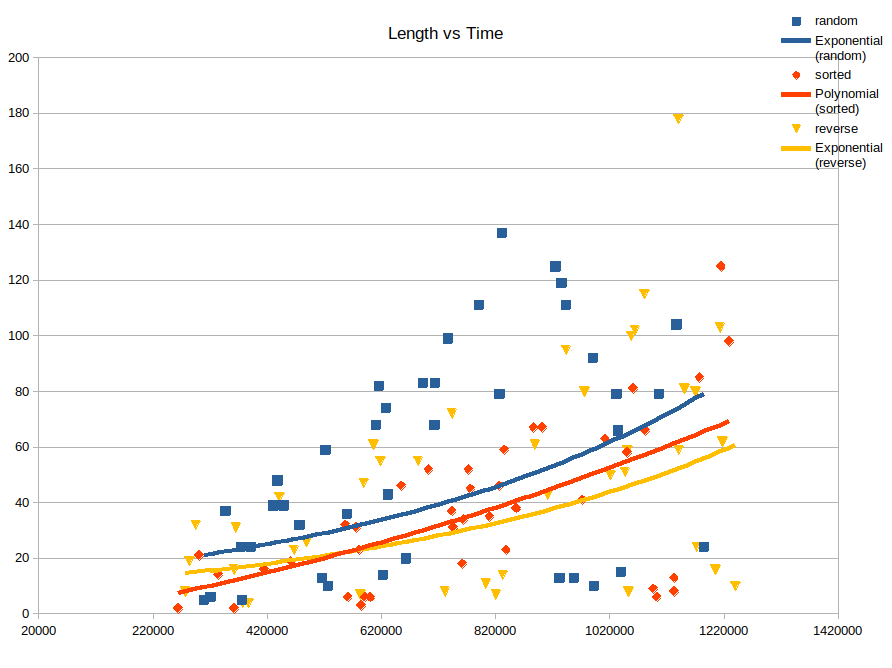


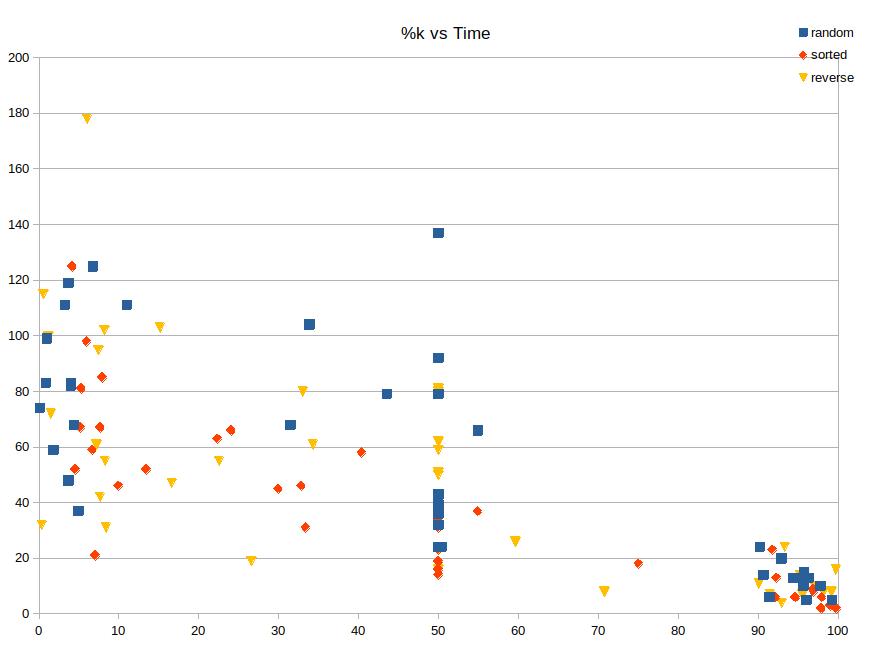
### Medium Array





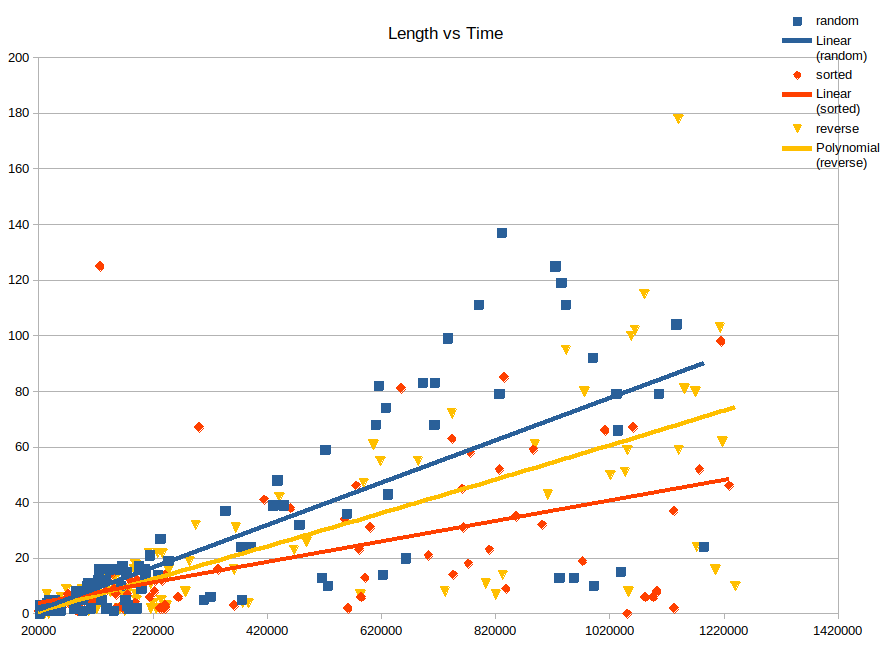
### Long Array

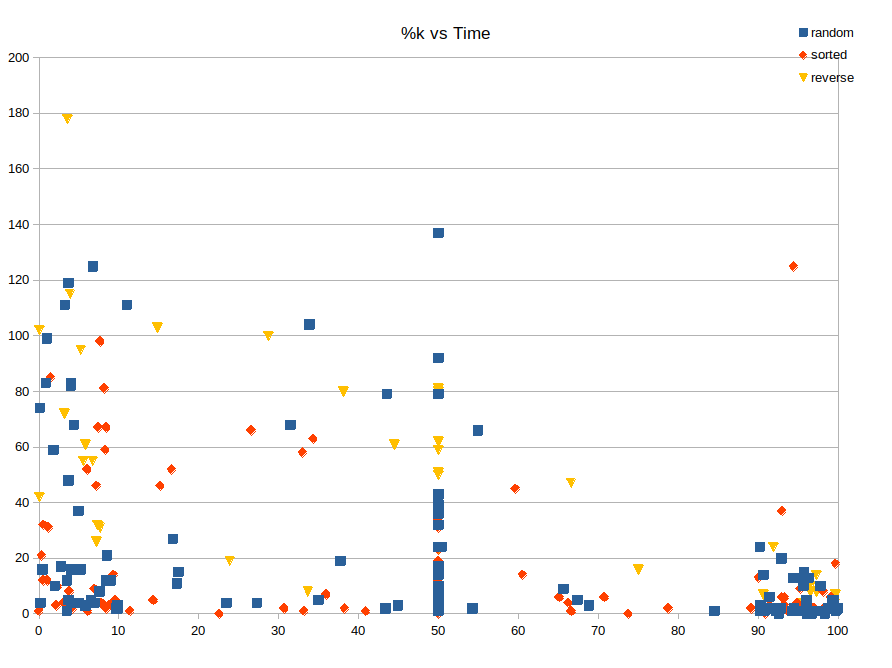




The Heapsort algorithm is an algorithm that has the same alignment in worst and best cases and does not clearly have a best or worst-case scenario. We can clearly see this from the graphics. Sometimes sorted, sometimes random, sometimes reverse sorted worked faster. It's all minor speed differences related to the positions of the values in the array. Therefore, the type of the array does not affect the speed clearly.

### Merging Charts



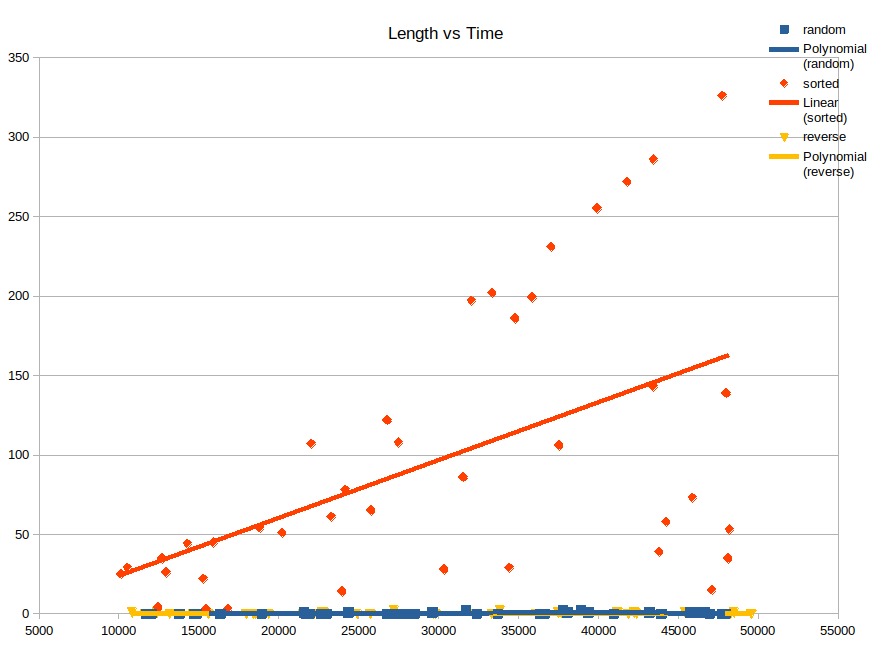


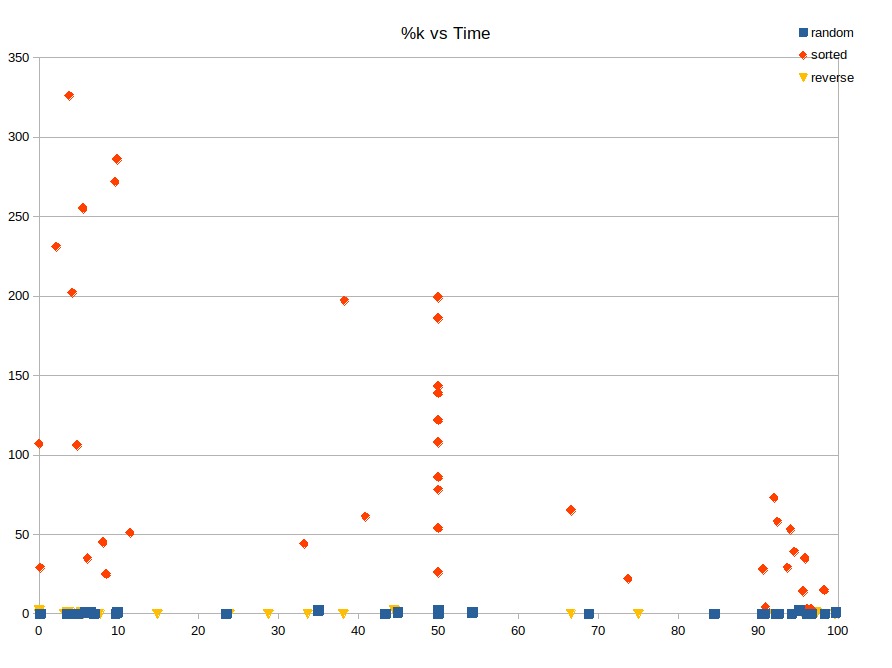
Since the Heapsort algorithm has the same character as the merge sort, it is not possible to see a clear difference according to the type of the array in the Length vs Time table, there are ups and downs.

Since we are using the partial heap sort algorithm, not the normal heapsort, we can clearly see that the time decreases enormously as k increases in the %k vs Time table. This is reversed because when we want to find the kth smallest element in the max-heap, we do (n-k) root removal. Therefore, the more k increases, the less we have to do.

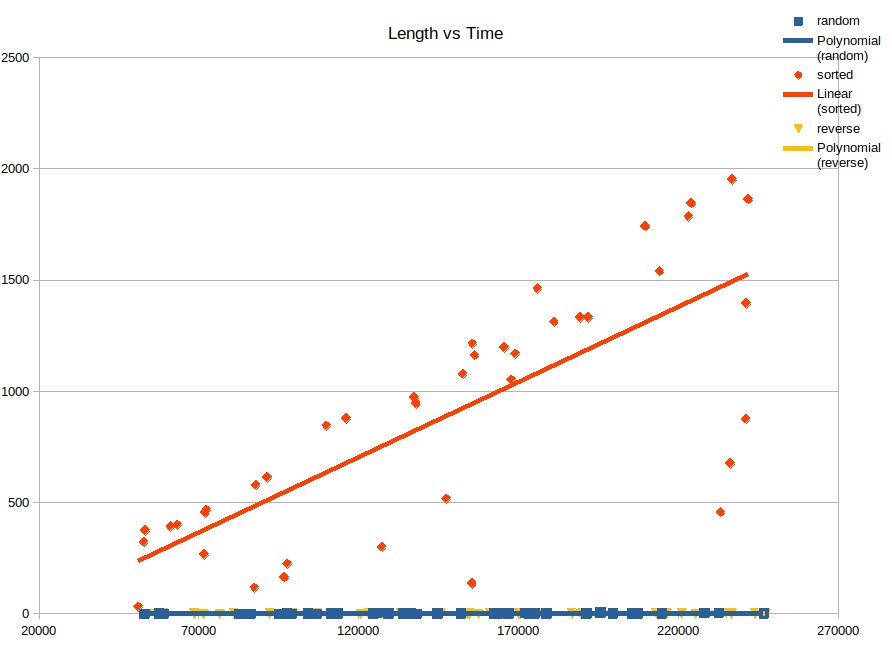
## Quick Select:

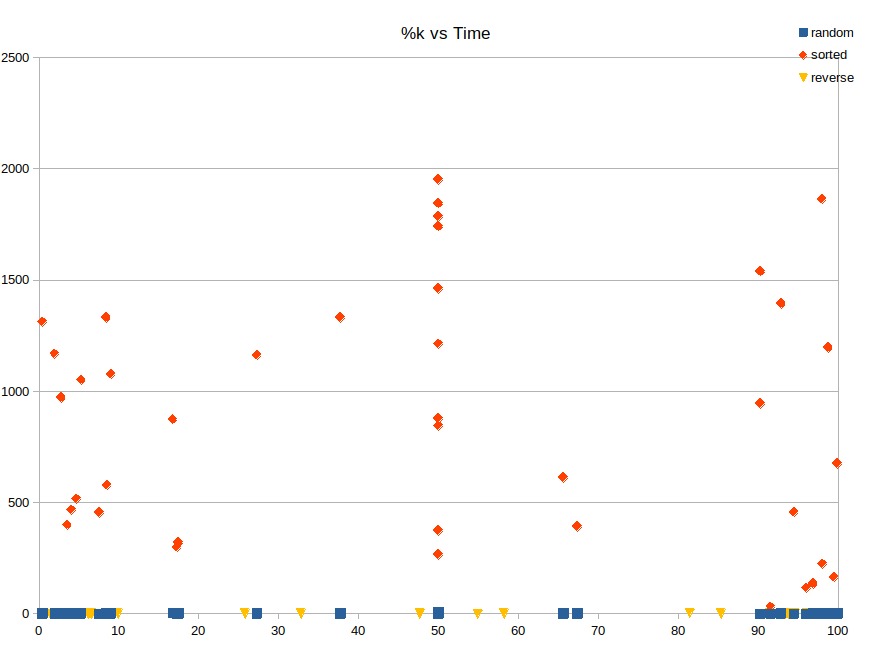
### Short Array



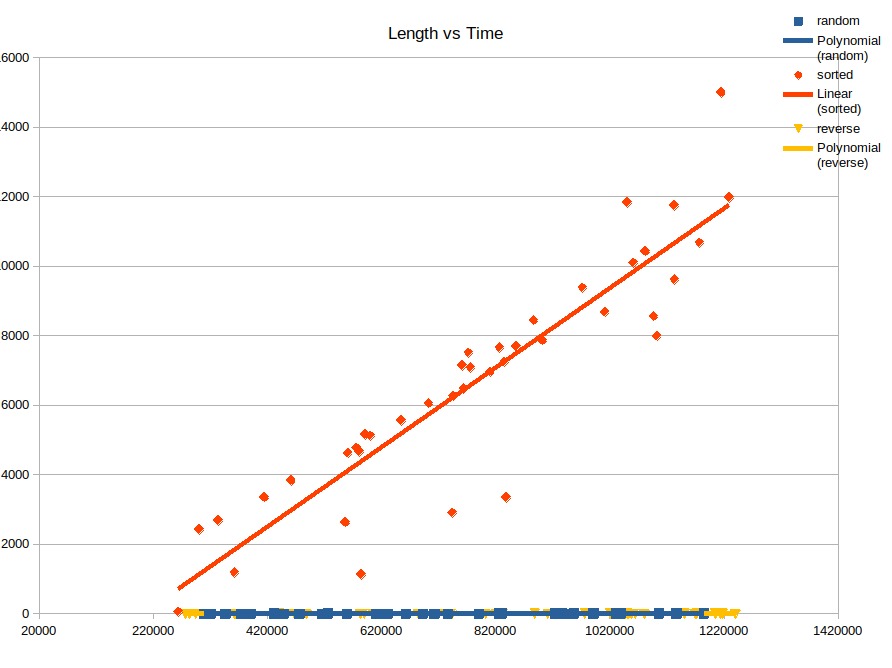


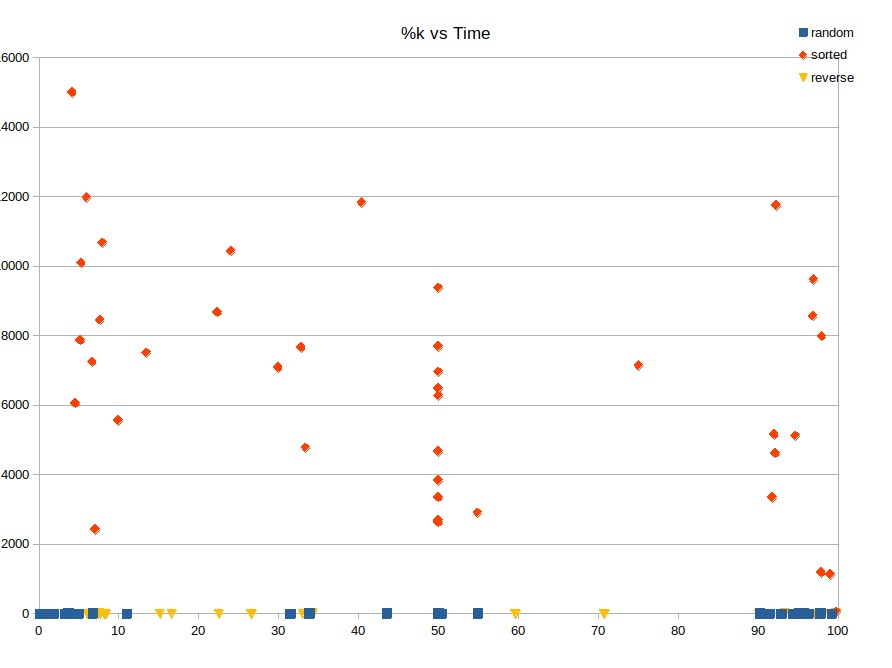
### Medium Array



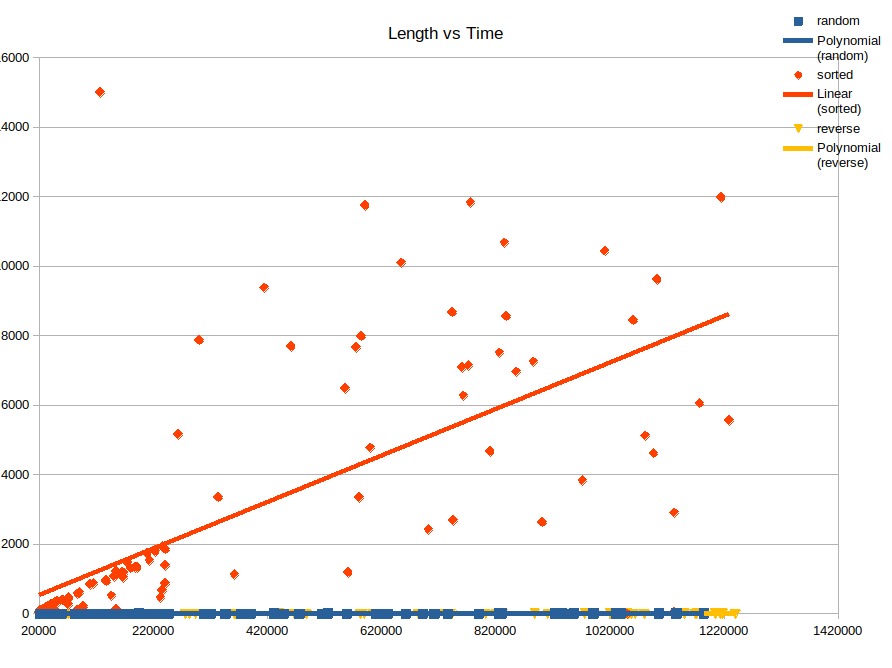


### Long Array



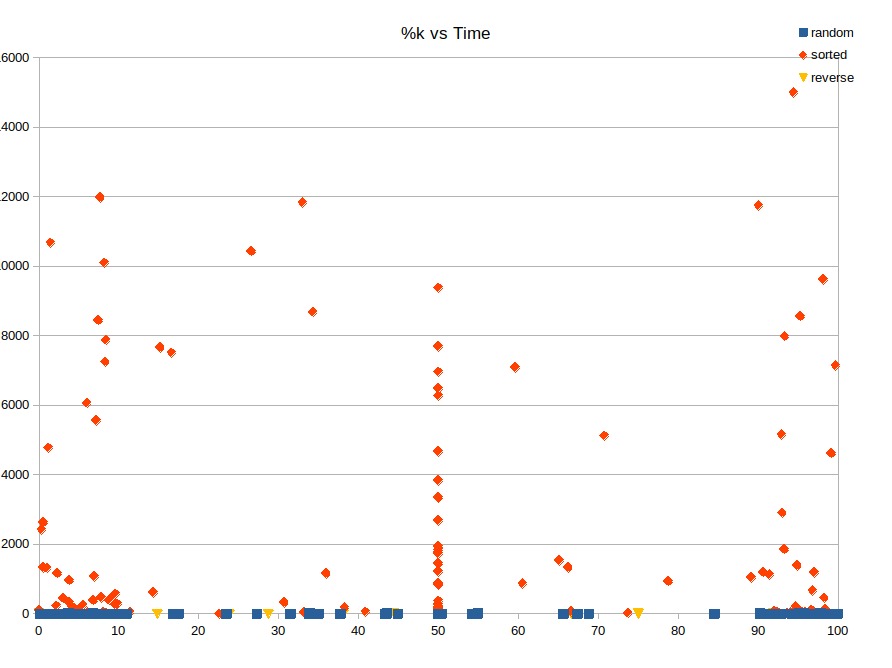


### Merging Charts

****

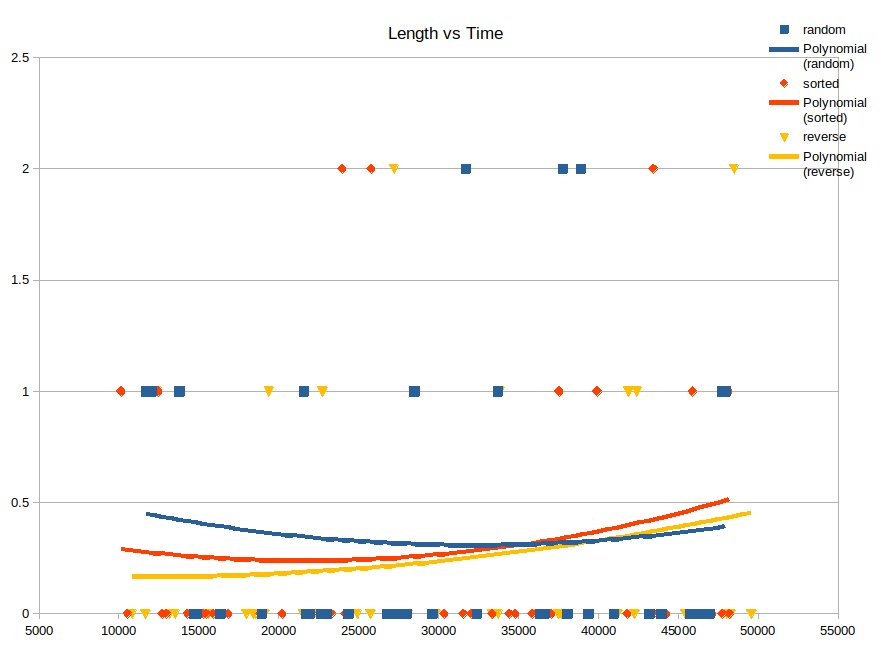
When we look at the analyzes and tests we have done from the graphics in the quickselect algorithm, there is an increase in time when sorted, and this is due to the partition algorithm which we use like in quicksort. This part has the worst time complexity of O(n2). Works very fast to find the kth element except the sorted array.

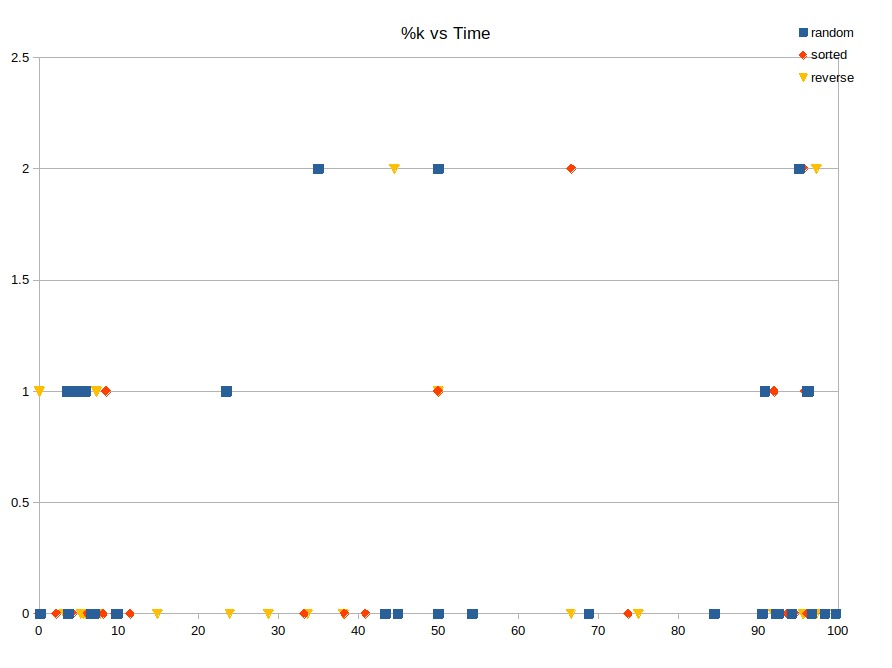
As we can see in the %k vs Time graphs, there is no noticeable change in the speeds according to the position of k, since it is not a partial algorithm.

****

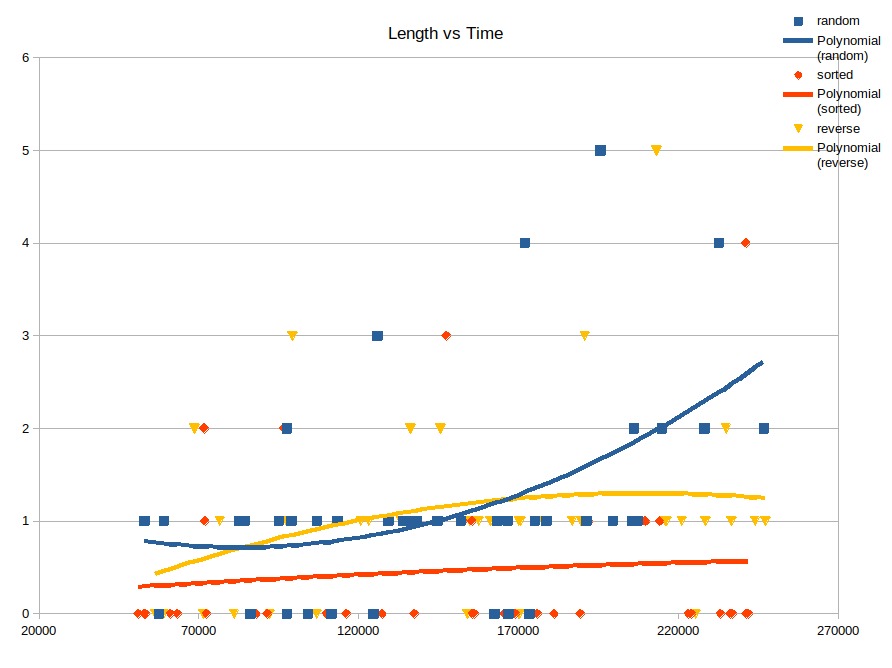
## Quick Select (Median-of-Three):

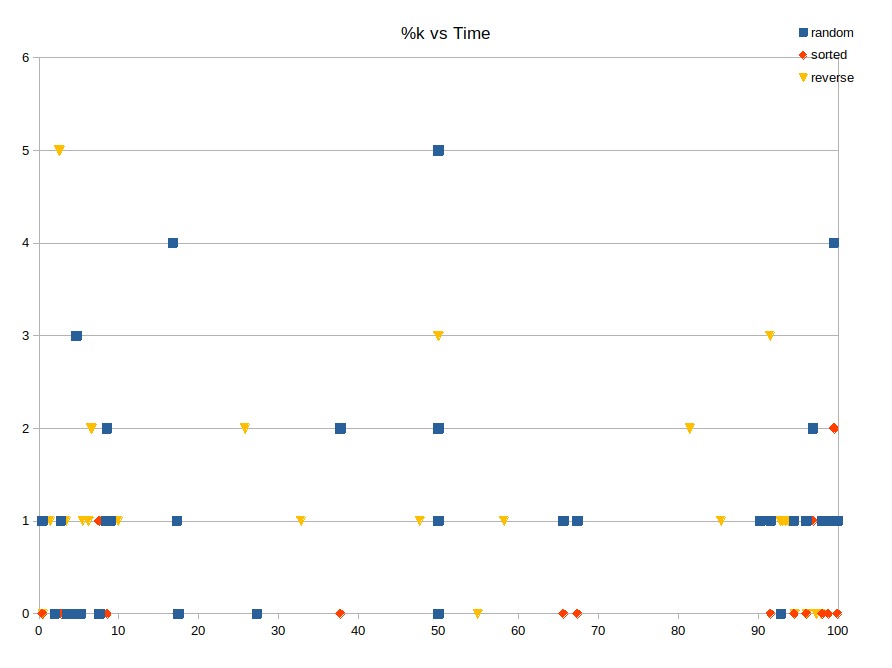
### Short Array



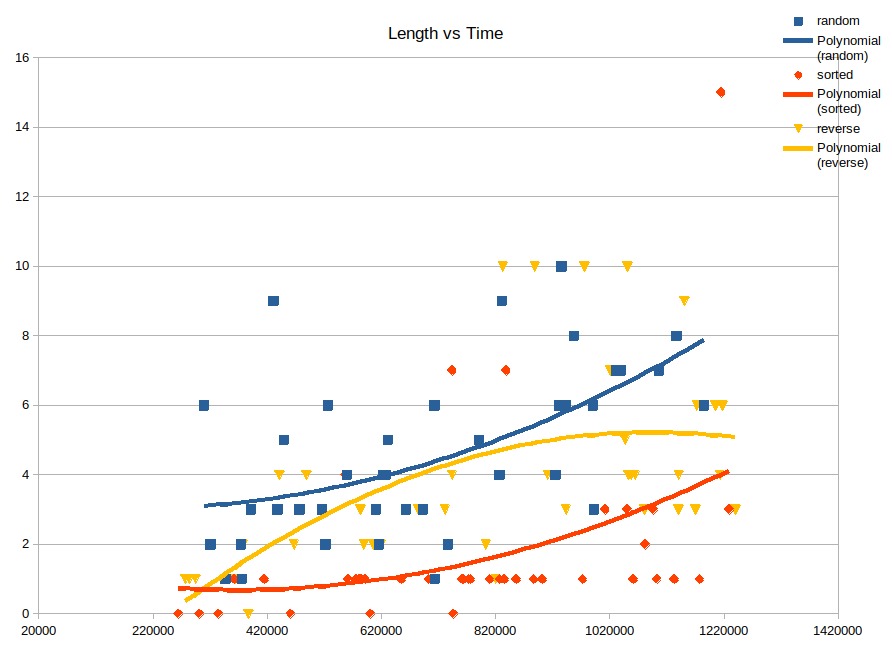


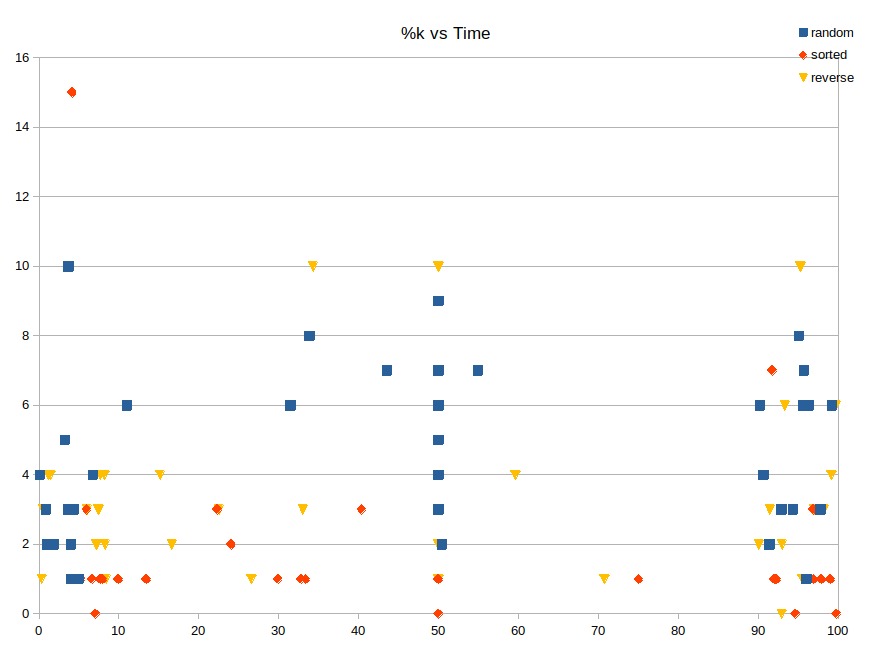
### Medium Array



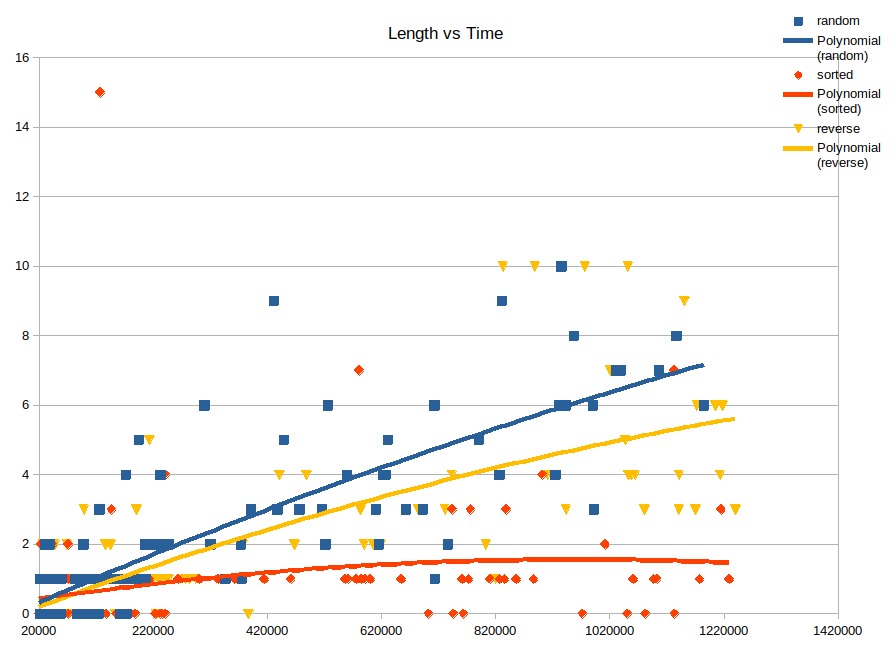


### Long Array



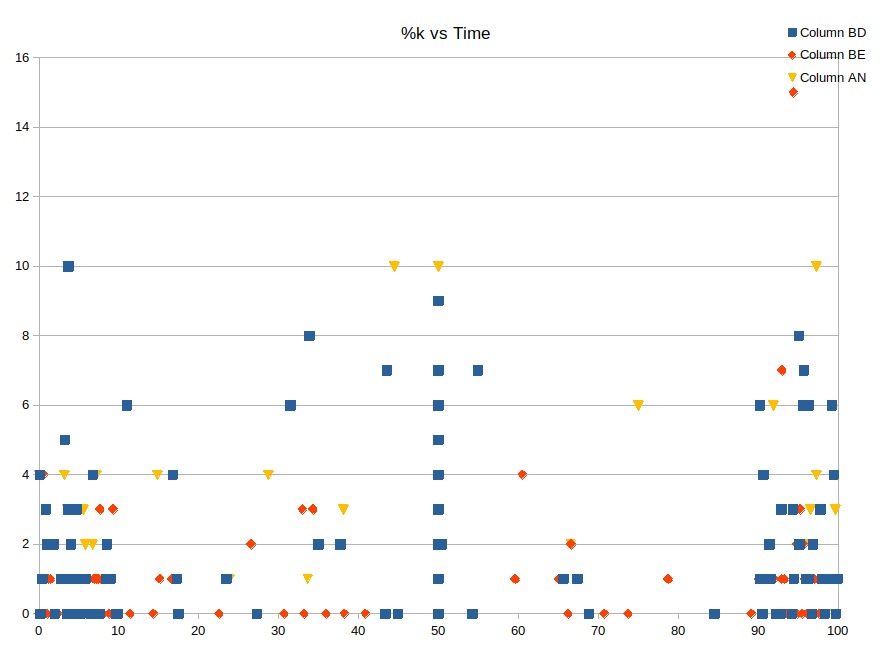


### Merging Charts



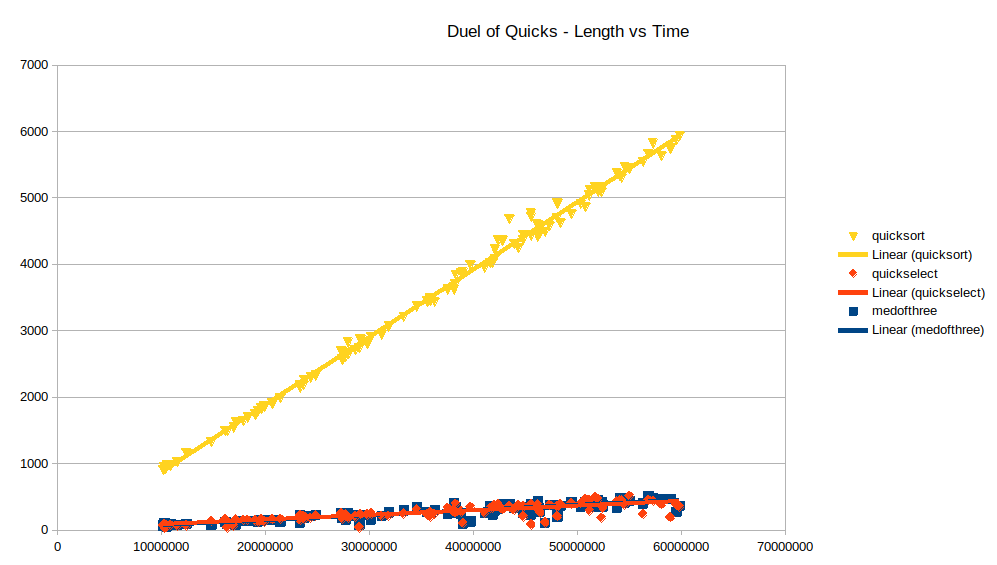
Thanks to the median of three partition algorithm(goat), the defect in the normal quick select is covered, providing speed in all situations.

As we can see in the %k vs Time graphs, there is no noticeable change in the speeds according to the position of k, since it is not a partial algorithm.



## Duel of Quicks:

In this part of the report, because the quickselect, quicksort, and quickselect (median-of-three) algorithms are extremely fast compared to the others, and often produce results close to 0, we gave them 100 gigantic random arrays and compared them with each other. The size of the arrays is between 10 million and 50 million (our computer didn't have enough memory for more).



As we can see from the graph, the select algorithms work faster than the sort algorithms and the 2 select algorithms can operate at almost the same speed as each other. However, it should not be forgotten that select algorithms only select. So it doesn't sort the array completely. Therefore, it is quite natural that it can complete its work in a shorter time than other algorithms.

# Conclusion

As a result, when we examine all algorithms, we can see that each algorithm has some special aspects of its own. To consider the algorithms one by one:

1. Insertion Sort: In small arrays, it consumes less memory and can be used in simple tasks since its algorithm can be written very easily.

1. Merge Sort: Shows high performance even in larger arrays, can be used to combine 2 different arrays with a few changes

1. Quick Sort: One of the fastest algorithms, but the biggest weakness of sorted or very close to sorted arrays. Since the program will give an error and close when such arrays arrive, it should be ensured that such an array will not appear while using it.

1. Partial Selection Sort:It has a simple algorithm like Insertion Sort, but it should not be used as the k value increases. If you want to find the smallest first few values ​​of a data set, it's definitely fast but incredibly slow as k increases.

1. Partial Heap Sort: An algorithm that can show high performance in large arrays such as merge sort, but because it is partial and is used on max-heap, it starts to give faster results as the k value increases. That is, it is more suitable to be used in large arrays and if the k value is large.

1. Quick Select: Quick Select algorithm has exactly the same character as quicksort. If the array is sorted or very close to being sorted, the program will throw an error. It should be kept in mind that select algorithms are not sorting algorithms, if there is only one kth smallest element to be found in that array, it would be more appropriate to find it with it. If it is to be found more than once, it will take longer to run this algorithm many times in the long run than to run the others all at once, as this algorithm does not sort the array.

1. Quick Select (Median-of-Three): It is the best algorithm according to all other (in the report) algorithms. However, as explained above, if it will be found only once, it is more logical to use it. Unlike normal quick select, it is unaffected by an array that is sorted or very close to sorted. All types work at the same speed.

# References & Resources

* <https://www.geeksforgeeks.org/insertion-sort/#:~:text=Insertion%20sort%20is%20a%20simple,Algorithm>
* <https://en.wikipedia.org/wiki/Insertion_sort>
* <https://en.wikipedia.org/wiki/Merge_sort#:~:text=In%20computer%20science%2C%20merge%20sort,in%20the%20input%20and%20output>.
* <https://www.geeksforgeeks.org/merge-sort/>
* <https://en.wikipedia.org/wiki/Quicksort>
* <https://www.geeksforgeeks.org/quick-sort/>
* <https://www.techiedelight.com/quicksort/>
* <https://www.geeksforgeeks.org/selection-sort/>
* <https://en.wikipedia.org/wiki/Selection_sort>
* <https://www.geeksforgeeks.org/heap-sort/>
* <https://en.wikipedia.org/wiki/Heapsort>
* <https://en.wikipedia.org/wiki/Quickselect>
* <https://www.geeksforgeeks.org/quickselect-algorithm/>
* <https://stackoverflow.com/questions/7559608/median-of-three-values-strategy>
* Our lecture slides