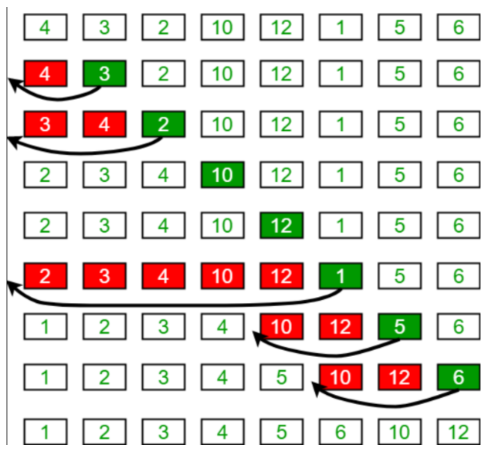
INSERTION SORT

DEFINITON AND APPLICATION OF INSERTION SORT

Insertion sort is an application of the decrease-by-one technique to build sorted array or list. In this sorting method we solve smaller problem to get solution of original problem.

In an insertion sort, the first element in the array is considered as sorted, even if it is an unsorted array. In an insertion sort, each element in the array is checked with the previous elements, resulting in a growing sorted output list. With each iteration, the sorting algorithm removes one element at a time and finds the appropriate location within the sorted array and inserts it there. The iteration continues until the whole list is sorted.



Execution example with unsorted array

ANALYSIS OF INSERTION SORT

THEORETICAL ANALYSIS

PsudoCode of Insertion Sort ->

```
InsertionSort($A[0..n-1]$)
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of $n$ orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ to $n-1$ do
$v \leftarrow A[i]$
$j \leftarrow i-1$
while $j \geq 0$ and $A[j] > v$ do
$A[j+1] \leftarrow A[j]$
$j \leftarrow j-1$
$A[j+1] \leftarrow v$
```

- The basic operation of the algorithm is the key comparison
- The number of key comparisons in this algorithm obviously depends on the nature of the input.

In the worst case,

$A[j] > v$ is executed the largest number of times especcially inputs like reverse order sorted list or array of decreasing values

Therefore,

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

In the best case,

the comparison *A*[*j*]> *v* is executed only once on every iteration of the outer loop. The best case

inputs for insertion sort are already sorted arrays or lists

Therefore,

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

In average case ,

randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays and it is asymptoticly yields *O(n*^2*).*

Therefore,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

Conclusion

In best case of insertion sort, it has time complexity O(n) with inputs which are sorted alredy, in average case with inputs which have random numbers and worst case with inputs that reverse order sorted it has a time complexity O(n^2) theoretically.

EMPIRICAL ANALYSIS

For empirical analysis of the insertion algorithm, operation time counter has chosen as the metric. By using this metric, count of the basic operation execution will be computed.

The basic operation of the algorithm is the key comparison.

DIFFERENT TEST INPUTS FOR EMPIRICAL ANALYSIS

4 different input types with 9 different sizes for each has been used as test inputs.

Input types: already sorted, reverse sorted, equal values, randomly duplicated values.

Input sizes: 1, 10, 50, 100, 500, 1000, 5000, 7500, 9900.

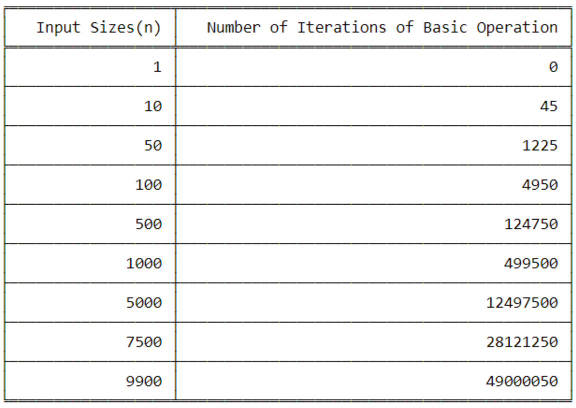
Testing with reverse sorted list

EX → reverse_sorted_1 =[1] ,

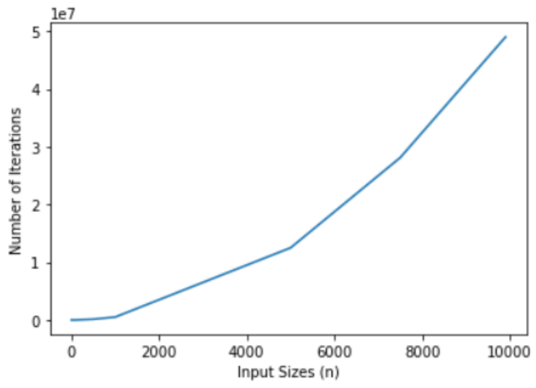
reverse_sorted_10 =[9,8,7,6,5,4,3,2,1,0]

reverse_sorted_50 =[49,48,47,46,45,44,.......0]

…



Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	45
50	1225
100	4950
500	124750
1000	499500
5000	12497500
7500	28121250
9900	49000050

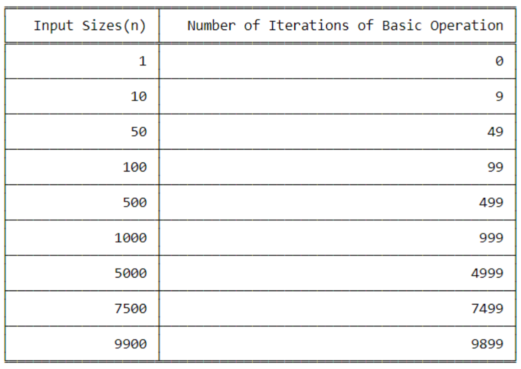


Since types of decreasing order sorted lists are apply worst case for insertion sort, results of testing with inputs which are reverse sorted lists, parallel to theoretical analysis. As seen in table , relation of number of iteration and input sizes is corresponding with the formula of “ (n*(n-1)) / 2 “. Moreover according to input size and number of iterations plot, quadratic increasing has observed , therefore we can say , in empricaly , when reverse order lists are input for insertion sort, time complexity is going to be O(n^2) and this is the worst case for insertion sort.

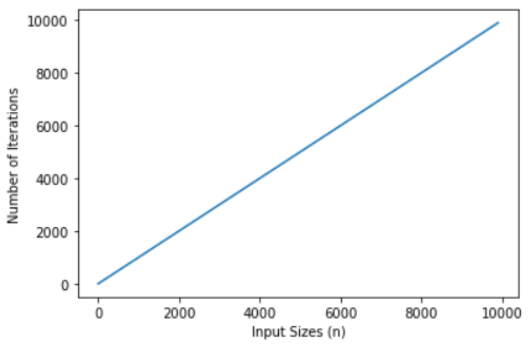
Testing with already sorted list

EX → reverse_sorted_10 = [9,8,7,6,5,4,3,2,1,0]

…



Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	9
50	49
100	99
500	499
1000	999
5000	4999
7500	7499
9900	9899



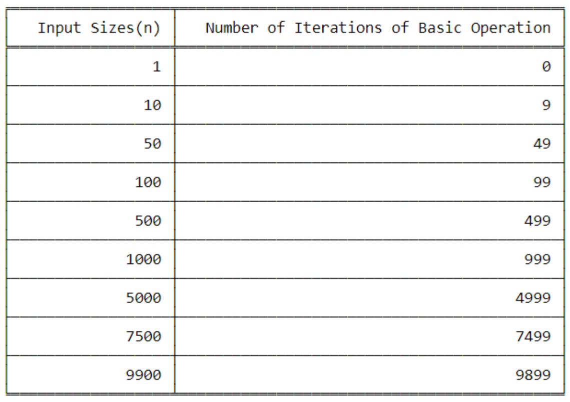
As known in therotical analyisis , types of inputs that already sorted lists are apply best case for Insertion sort because there is no need insertion operation. As seen in table , relation of number of iteration and input sizes is corresponding with the formula of " (n-1) ", this indicates the loop for elements. Moreover Input sizes and number of iterations are one-to-one corresponding to eachother and lineer increase observed according to plot above.

In conclusion , when increasing order lists are input in insertion sort , the time complexity of algorithm is going to be O(n) and this is the best case for this algorithm

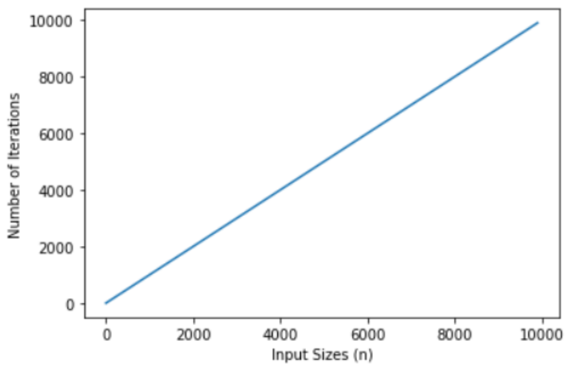
Testing with equal values in list

EX → equal_list_10 = [5,5,5,5,5,5,5,5,5,5]

…



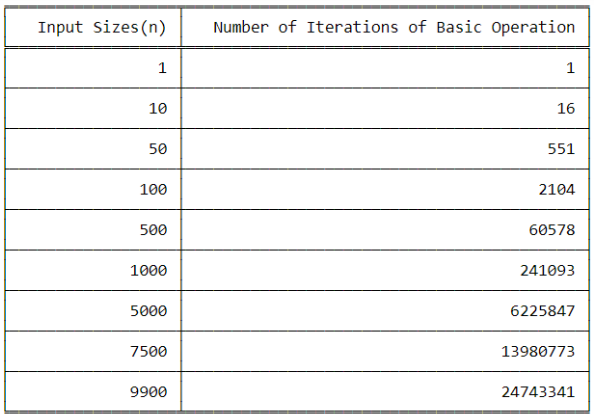
Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	9
50	49
100	99
500	499
1000	999
5000	4999
7500	7499
9900	9899



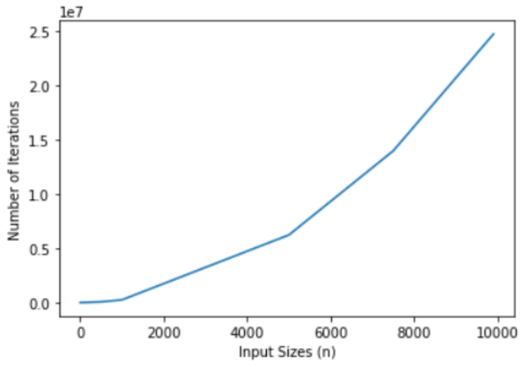
Since every elements in input array are equal to eachother , this case is also one of the best case for insertion sort . According to table and plot above, lineer increase has observed according to input sizes and number of iterations, therefore when inputs have equal value elements, time complexity of the algorithm is going to be O(n)

Testing with duplicated and random values in list

Ex → equal_list_10 = [2,2,6,8,5,8,6,4,3,3]



Input Sizes(n)	Number of Iterations of Basic Operation
1	1
10	16
50	551
100	2104
500	60578
1000	241093
5000	6225847
7500	13980773
9900	24743341



In this case, to approximately observe average case of insertion sort ,inputs have randomly chosen elements and there can be many duplicate element in it. As seen in table and plot, relation of number of iteration and input sizes is approximately corresponds with the formula of “((n^2) / 4)“ Therefore we can say there are quadratic increase observed for this case. Due to algorithm yields quadraticaly , time complexity for this case is O(n^2)

CONCLUSION

As seen above, algorithm has been analyzed and compared with theoretical and empirical results. Graphs and tables show us that results of empirical analysis are in harmony with the results of theoretical analysis.

Iteration counts from empirical analysis are in the range of the time complexity values from theoretical analysis:

for worst case:

$$\theta(n^2) \in O(n^2)$$

for average case:

$$\theta(n^2) \in O(n^2)$$

for best case:

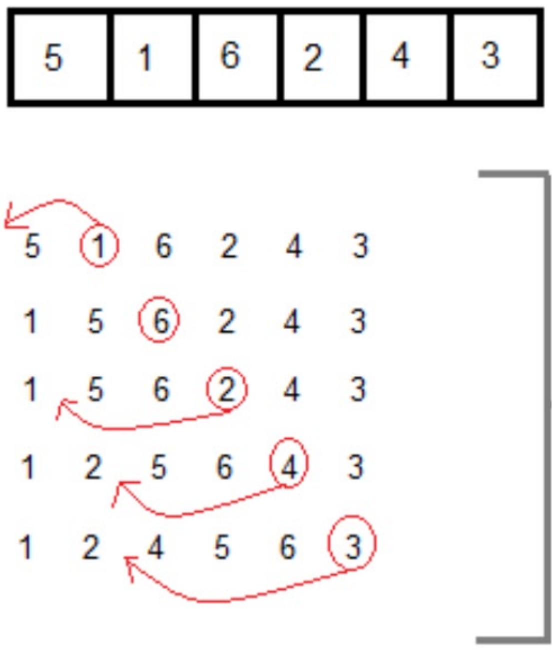
$$\theta(n) \in O(n)$$

BINARY INSERTION SORT

DEFINITON AND APPLICATION OF BINARY INSERTION SORT

Binary insertion sort is based on merging of two algorithms Binary Search and Insertion Sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. Using binary search helps to reduce the number of comparisons in normal Insertion Sort. After the proper location found by using binary search We can obtain a sorted list by shifting rest of the elements to put selected element to proper place.

Example execution of binary insertion sort



ANALYSIS OF BINARY INSERTION SORT

THEORETICAL ANALYSIS

Because of this method has 2 parts , it has to analysed separately. In the part of binary search , algorithm finds proper location for element that is going to be inserted, so for each element in the list, binary search is going to be executed. Due to binary search has O(log(n)) time complexity , for each element, the complexity of the algoirthm will be O(nlog(n)). However Binary Insertion sort has not O(nlog(n)) time complexity in every case because after finding proper location for every element in the list , they have to be placed in their proper location. For this proccess algorithm shifts the elements for each iteration, therefore the time complexity of binary insertion sort will be O(n^2).

- In the part of Binary Search , basic operation is comparison and it has *O(nlog(n)) time complexity.*
- In the part of Insertion, basic operation is shifting and it has *O(n^2) time complexity.*

In the best case,

If the input is already sorted list , in the insertion part, because of the selected element is greater than the part of the list which is sorted, there is no need to shift any element, therefore time complexity of the program will be O(nlog(n)).

In the worst case,

Assume that if the input is reversed sorted list , In the section of Insertion , Due to selected element is smaller than the part of the list which is sorted , all the elements in the list have to be shifted in each iteration therefore time complexity of the program will be O(n^2).

In the average case,

the time compelxity of the program is O(n^2)

EMPIRICAL ANALYSIS

For empirical analysis of the binary insertion algorithm, operation time counter has chosen as the metric. By using this metric, count of the basic operation execution will be computed.

For the empirical analysis of the algorithm, basic operation will be comparison for binary search part. For Insertion part, it will be shifting.

DIFFERENT TEST INPUTS FOR EMPIRICAL ANALYSIS

4 different input types with 9 different sizes for each has been used as test inputs.

Input types: already sorted, reverse sorted, equal values, randomly duplicated values.

Input sizes: 1, 10, 50, 100, 500, 1000, 5000, 7500, 9900.

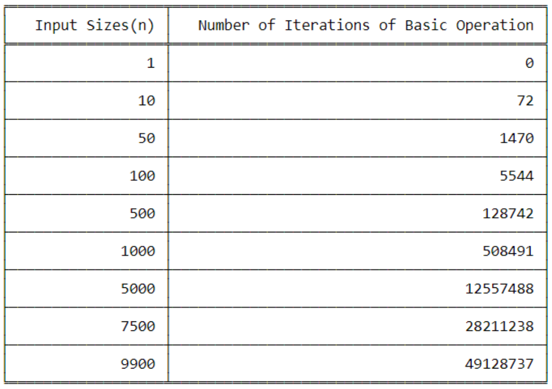
Testing with reverse sorted list

EX → reverse_sorted_1 =[1]

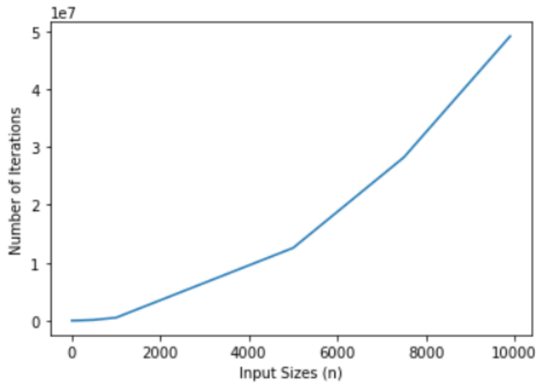
reverse_sorted_10 =[9,8,7,6,5,4,3,2,1,0]

reverse_sorted_50 =[49,48,47,46,45,44,.......0]

...



Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	72
50	1470
100	5544
500	128742
1000	508491
5000	12557488
7500	28211238
9900	49128737



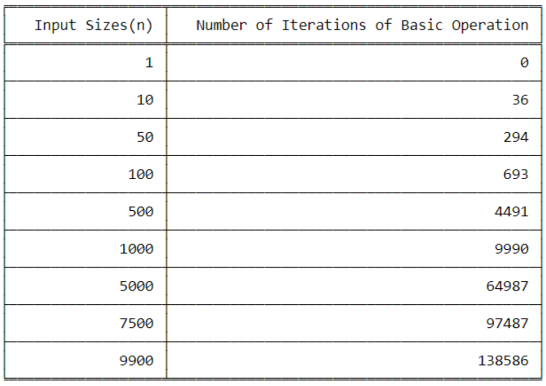
As seen in table , input sizes and number of iteration have a relation between with a formula that

"((n*n-1) / 2) + (n-1)*log2(n)". First part of the formula [(n*n-1) / 2)] indicates the number of shifting operation in every iteration , second part of the formula inditades the finding proper location with binary search for every element. This is equivalent to Θ (n^2) + Θ (nlogn) ∈ O(n^2) asymtotically. Moreover, quadratic increase observed in plot above. Therefore the time complexity for this case is O(n^2)

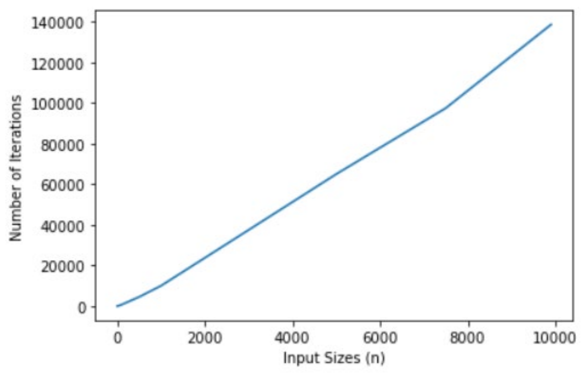
Testing with already sorted list

EX → sorted_10 = [0,1,2,3,4,5,6,7,8,9]

...



Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	36
50	294
100	693
500	4491
1000	9990
5000	64987
7500	97487
9900	138586



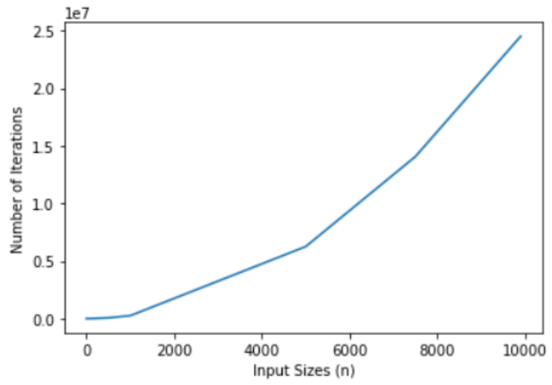
In this case , Due to input is already sorted list , there is no need shifting operation. As seen in the plot , the relation of number of iteration and input sizes is corresponding with the formula of "(n)*($\lceil \log_2(n+1) \rceil$)". This formula indicates the number of binary search operation for each iteration and this is equavelent to Θ (nlogn) ∈ O(nlogn) asymtotically.

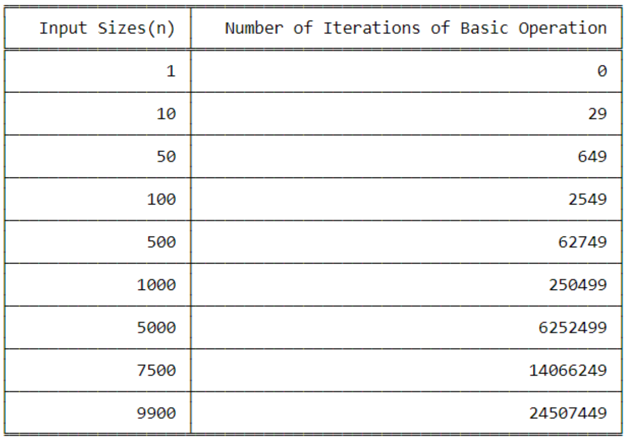
Moreover, in the plot above logarithmic increase observed. Therefore, in this case, time compelxity of the algorithm is O(nlog(n))

Testing with equal values in list

EX → equal_list_10 = [5,5,5,5,5,5,5,5,5,5]

…





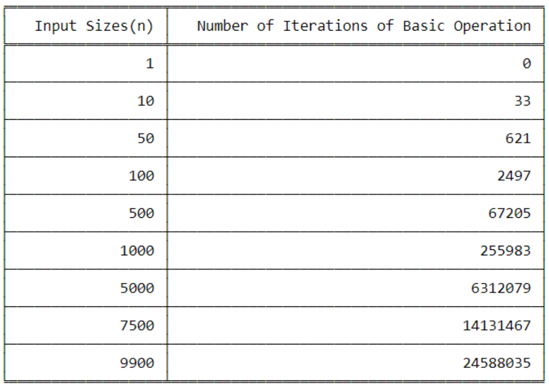
Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	29
50	649
100	2549
500	62749
1000	250499
5000	6252499
7500	14066249
9900	24507449

As seen in plot and table , There is a quadratic increase between input sizes and number of iteration . Because of shifting operation for each iteration algorithm yields quadratic increase.

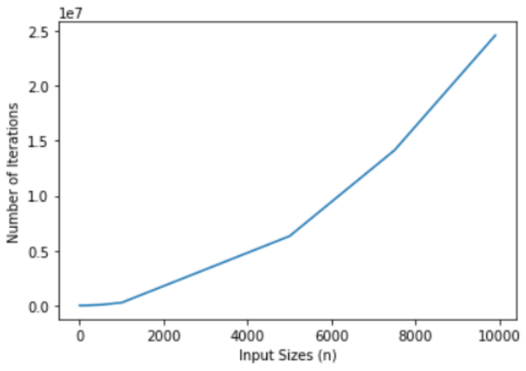
In conculision , time complexity of the algorithm is O(n^2).

Testing with duplicated and random values in list

Ex …



Input Sizes(n)	Number of Iterations of Basic Operation
1	0
10	33
50	621
100	2497
500	67205
1000	255983
5000	6312079
7500	14131467
9900	24588035



As seen in plot and table , There is a quadratic increase between input sizes and number of iteration . Because of shifting operation for each iteration algorithm yields quadratic increase.

In conculision , time complexity of the algorithm is O(n^2).

CONCLUSION

As seen above, algorithm has been analyzed and compared with theoretical and empirical results. Graphs and tables show us that results of empirical analysis are in harmony with the results of theoretical analysis.

Iteration counts from empirical analysis are in the range of the time complexity values from theoretical analysis:

for worst case:

$$\theta(n^2) \in O(n^2)$$

for average case:

$$\theta(n^2) \in O(n^2)$$

for best case:

$$\theta(nlogn) \in O(nlogn)$$

MERGE SORT

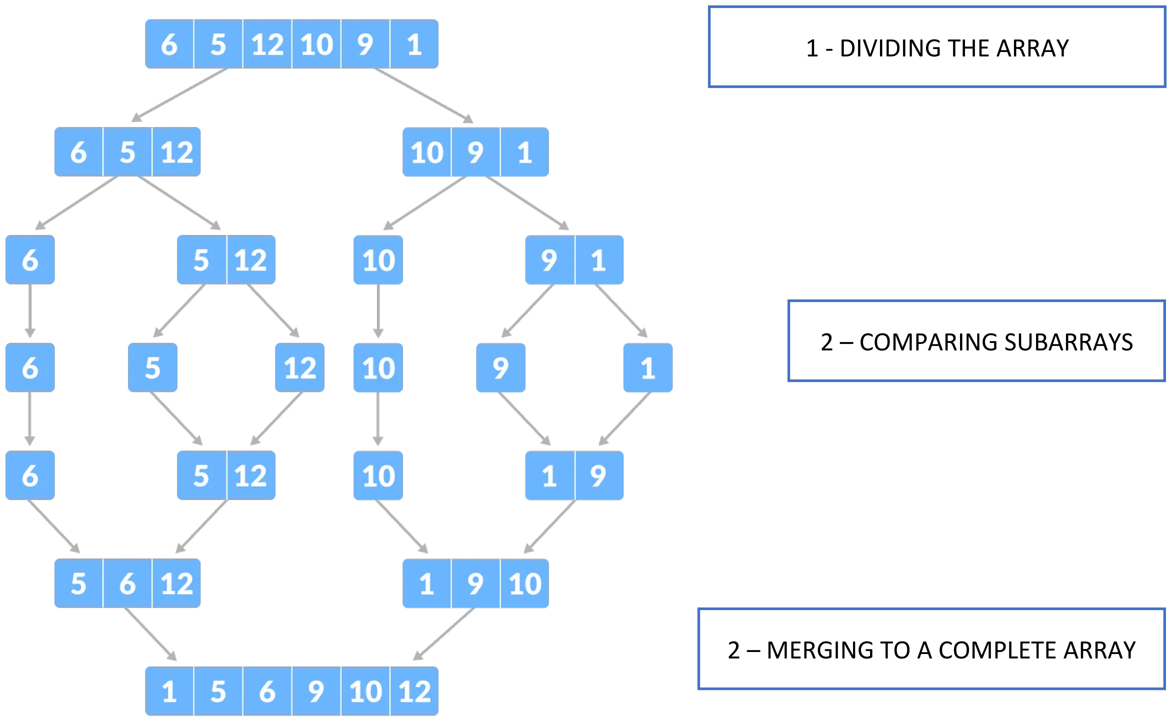
DEFINITON OF MERGE SORT

Merge sort is an algorithm which uses ‘Divide and Conquer’ strategy to operate. It divides the problem into subproblems, finds a solution for each subproblem and merges them.

Algorithm takes an unsorted array, recursively divides it to half until there are no more arrays to divide; then it merges them according to their values. After the last merge operation, acquired array will be sorted.

APPLICATION OF MERGE SORT

Application of Merge Sort can be represented as:



ANALYSIS OF MERGE SORT

THEORETICAL ANALYSIS

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

Following formula can be obtained by the pseudocode:

$$T(n) = 3T\left(\frac{n}{3}\right) + \frac{5n}{3} - 2$$

By using the Master's Theorem, time complexity for all cases can be found as follows:

$$T(n) \in \ \theta(nlogn)$$

EMPIRICAL ANALYSIS

For empirical analysis of the Merge Sort algorithm, operation time counter has chosen as the metric. By using this metric, count of the basic operations' executions will be computed.

For the empirical analysis of the algoritm, basic operations will be divide operation of array and comparison in merging.

Merge sort algorithm has O(nlogn) time efficiency for all cases, according to the theoretical analysis.

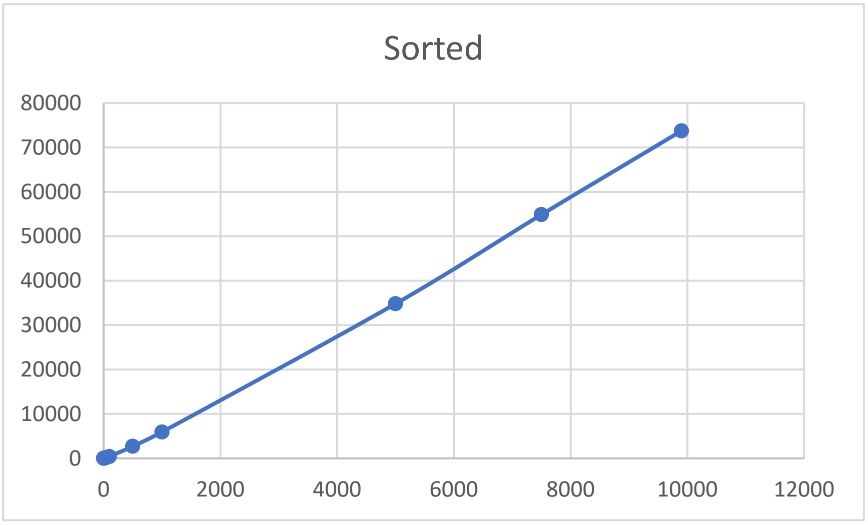
DIFFERENT TEST INPUTS FOR EMPIRICAL ANALYSIS

5 different input types with 9 different sizes for each has been used as test inputs.

Input types: already sorted, reverse sorted, equal values, randomly duplicated values, semi-sorted values.

Input sizes: 1, 10, 50, 100, 500, 1000, 5000, 7500, 9900.

INPUT SIZE	COUNT
1	0
10	24
50	182
100	415
500	2715
1000	5931
5000	34803
7500	54875
9900	73739

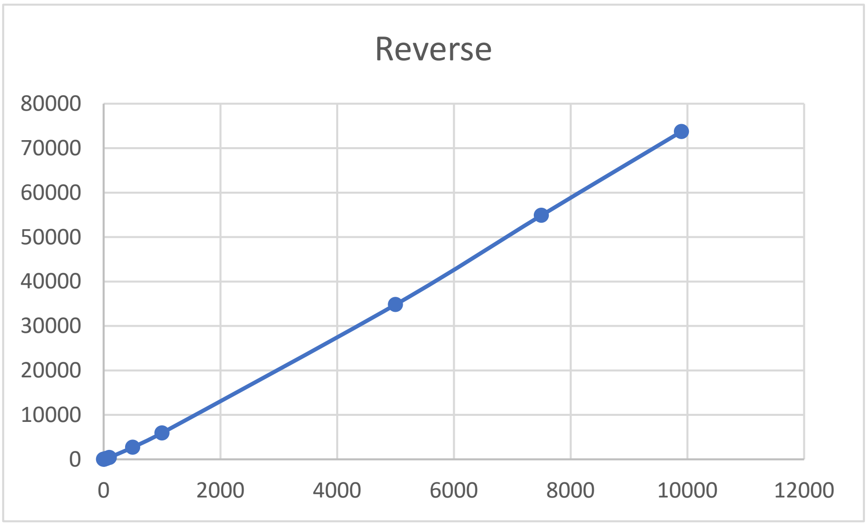


As seen in table, algorithm executed for O(k*nlogn) times, k as 0.72

$$for\ input\ size\ n = 100, \qquad O(0.72 * 100 * log100) = O(480)$$

$$\theta(nlogn)\ \in\ O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	24
50	182
100	415
500	2715
1000	5931
5000	34803
7500	54875
9900	73739

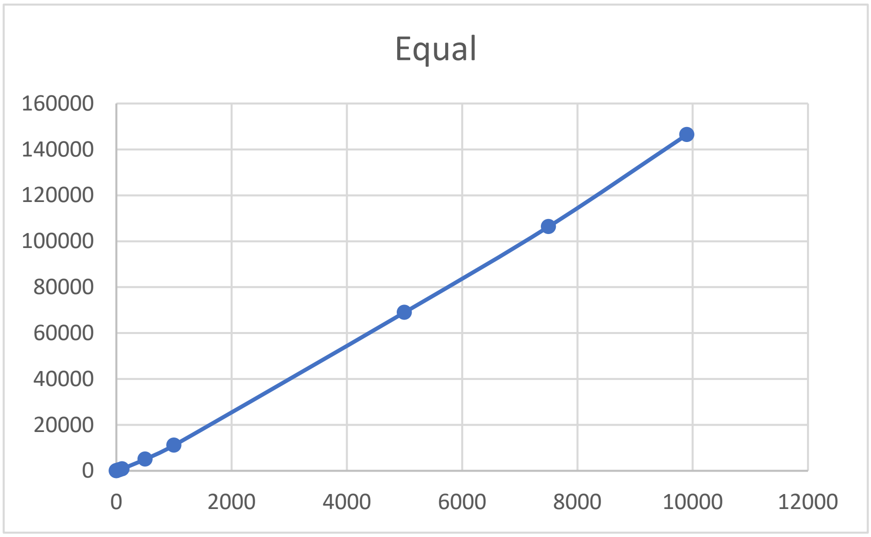


As seen in table, algorithm executed for O(k*nlogn) times, k as 0.72

$$for\ input\ size\ n = 100, \qquad O(0.72 * 100 * log100) = O(480)$$

$$\theta(nlogn) \in O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	47
50	355
100	811
500	5043
1000	11087
5000	69007
7500	106363
9900	146451

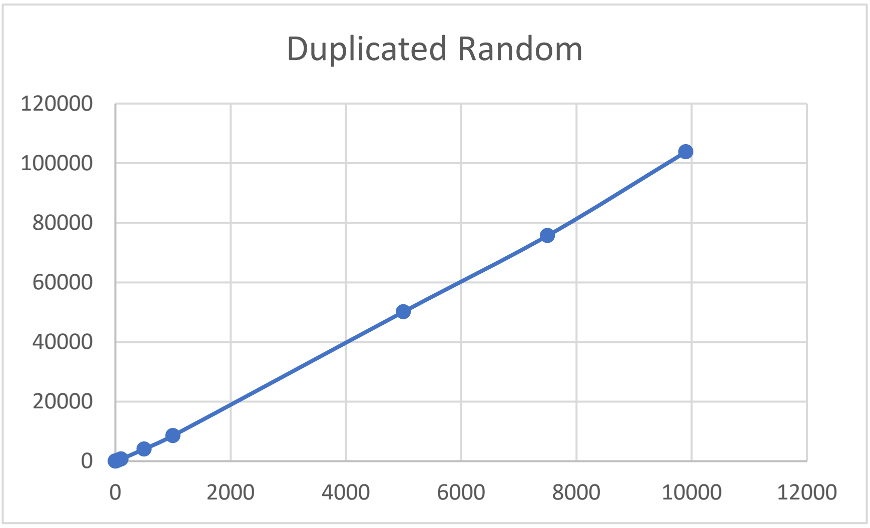


As seen in table, algorithm executed for O(k*nlogn) times, k as 1.41

$$for\ input\ size\ n = 100, \qquad O(1.41 * 100 * log100) = O(940)$$

$$\theta(nlogn) \in O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	36
50	323
100	694
500	4052
1000	8563
5000	50130
7500	75680
9900	103788

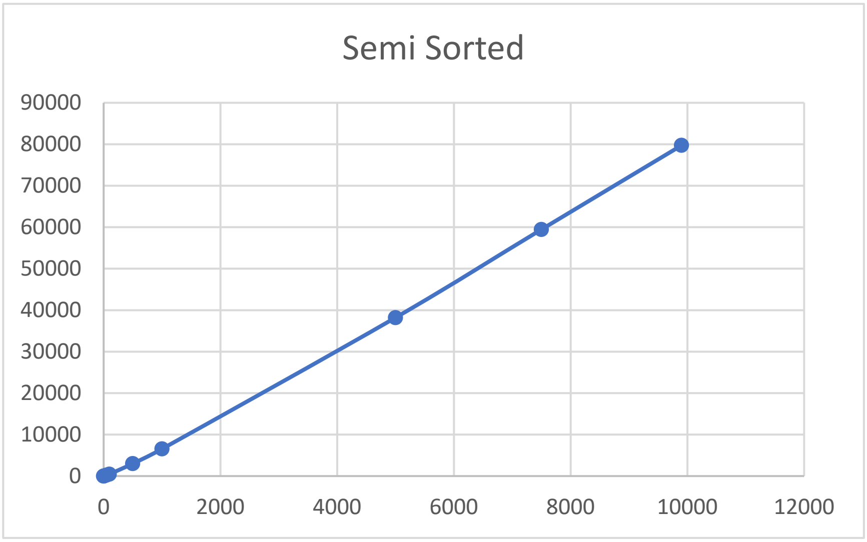


As seen in table, algorithm executed for O(k*nlogn) times, k as 1.08.

$$for\ input\ size\ n = 100, \qquad O(1.08 * 100 * log100) = O(720)$$

$$\theta(nlogn)\ \in\ O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	29
50	215
100	471
500	3005
1000	6548
5000	38202
7500	59462
9900	79726



As seen in table, algorithm executed for O(k*nlogn) times, k as 0.87

$$for\ input\ size\ n = 100, \qquad O(0.87 * 100 * log100) = O(580)$$

$$\theta(nlogn)\ \in\ O(k * nlogn)$$

CONCLUSION

As seen above, algorithm has been analyzed and compared with theoretical and empirical results. Graphs and tables show us that results of empirical analysis are in harmony with the results of theoretical analysis.

Iteration counts from empirical analysis are in the range of the time complexity values from theoretical analysis:

for worst case:

$$\theta(nlogn) \in O(nlogn)$$

for average case:

$$\theta(nlogn) \in O(nlogn)$$

for best case:

$$\theta(nlogn) \in O(nlogn)$$

QUICK SORT

APPLICATION OF QUICK SORT (PIVOT AS THE FIRST ELEMENT)

Note: Quicksort algorithm is being implemented in java rather than python due to lack of tail-recursive functions in python that doesn't allow us to use a larger stack. We have implemented the quicksort algorithm in java which allows us to use the stack of the application in a more efficient way.

Quicksort application has 4 steps,

1) Choose the pivot value
2) Apply partition to the array
    a) Increase start and decrease end indexes by 1 if the current index's value is lower or greater than the pivot.
    b) Swap the start and end values in the array and continue the step a until the start index is equal or greater than the end index.
    c) Array's end index value will be swapped by the pivot.
3) Apply Quick Sort recursively by decreasing the end index by 1.
4) Apply Quick Sort recursively by increasing the start index by 1.

ANALYSIS OF QUICK SORT (PIVOT AS THE FIRST ELEMENT)

THEORETICAL ANALYSIS

Number of comparisons between array elements and the pivot value will give the time complexity of the quick sort application.

There are two stages that time complexity is being handled:

1) Partitioning the array

At the initial situation, the pivot will be selected as the first element of the array. Start and the end indexes will imply the limits of the given array which will be changed before quicksort called recursively by setting the parameters as pivot that is start or end values for each two functions. In this step, there will be n comparisons between pivot and the array's elements. (Start part will be considered as n/2, the end part will be considered as n/2. This could be changed, but they will be equal to n+1 which will be n for time complexity.)

$C(n) = n$

2) Dividing the array to two and calling it recursively two times

After the array is divided to two by the pivot, the partitioning will be done recursively until there are two element arrays which will be achieved in k steps.

C(n) = 2 C(n/2) + n

In each step the array is being divided to two and that will be done in k steps,

k = n^2 => k = log2n

In k steps, the partitioning will do comparisons n times,

n * log2n + n

This statement can be simplified as, n * log2n and will have a big o of O(n * logn) (n is dominated by the n * logn)

EMPIRICAL ANALYSIS

Empirical Analysis will be calculated by a counter that is being increased when the array's start and end indexes values compared with pivot.

The worst case: When the array is sorted if the first or last element will be picked as the pivot the worst case will occur since there will be no other elements greater or lower than the pivot in this case.

Cw(n) = Cw(n-1) + n

Cw(n) = n + n + n + …. + n = n * n = n^2

Cw(n) = ***O(n^2)***

The average case: The array's average case will be ***n * logn***

Best case: The array's best case will occur when the middle element picked in a sorted array, ***n * logn.***

DIFFERENT TEST INPUTS FOR EMPIRICAL ANALYSIS

5 different input types with 9 different sizes for each has been used as test inputs.

Input types: already sorted, reverse sorted, equal values, randomly duplicated values, semi-sorted values.

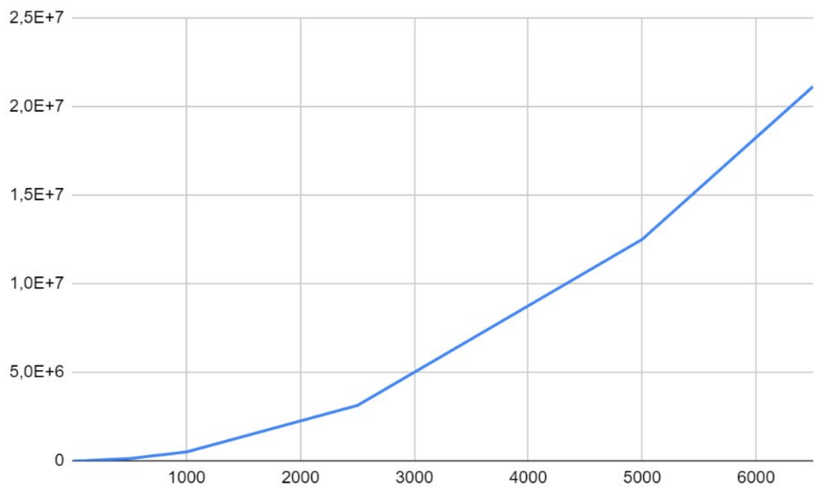
Input sizes: 1, 10, 50, 100, 500, 1000, 2500, 5000, 6500.

SORTED

Explanation: Sorted arrays are the worst case for quicksort. Since they are already sorted, quicksort becomes a brute force approach in sorted arrays. That makes time complexity O(n^2).

WORST CASE

INPUT SIZE	COUNT
1	0
10	54
50	1274
100	5049
500	125249
1000	500499
5000	69007
7500	106363
9900	146451



CALCULATIONS FOR THE SORTED

Assume k = 6, Theoretical time complexity = O(n ^ 2)

For 1000, 1000 * 1000 = 10^6 * 6 = 6 * 10 ^6

6 * 10^6 > 500499

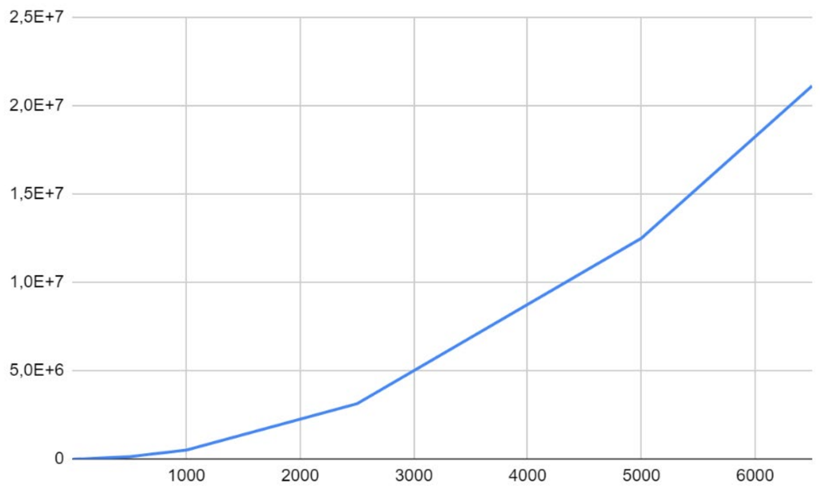
k = 6

REVERSE SORTED

Explanation: Reverse Sorted arrays are also the worst case for quicksort. Since they are already sorted, quicksort becomes a brute force approach in sorted arrays. That makes time complexity O(n^2).

WORST CASE

INPUT SIZE	COUNT
10	50
50	1250
100	5000
500	125000
1000	500000
2500	3125000
5000	12500000
6500	21125000
10	50



CALCULATIONS FOR REVERSE SORTED

Assume k = 5, Theoretical time complexity = O(n ^ 2)

For 1000, 1000 * 1000 = 10^6 * 6 = 5 * 10 ^ 6

5 * 10^6 >= 5 * 10 ^ 5

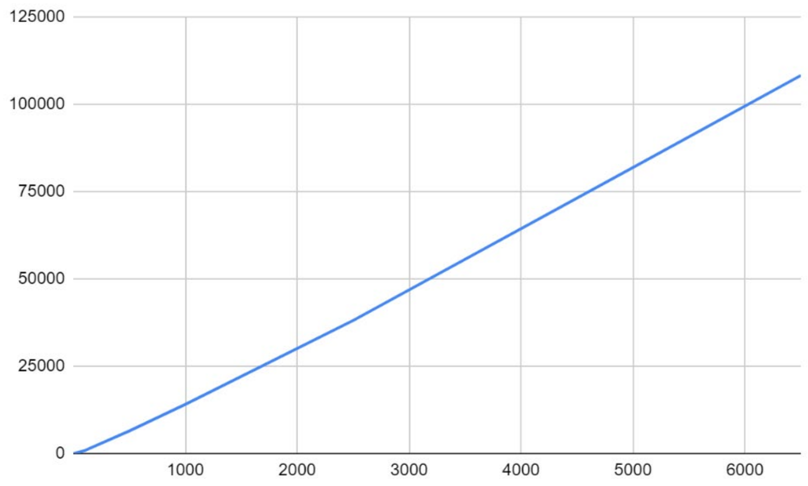
k = 5

RANDOM

Explanation: Random can be described as the best or average case for the quicksort algorithm, that has a time complexity of O(nlogn).

AVERAGE - BEST CASE

INPUT SIZE	COUNT
1	0
10	34
50	391
100	800
500	6447
1000	14074
2500	38097
5000	81908
6500	108282



Assume k = 4, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 4 = 4 * 10 ^5

4 * 10^5 >= 14074

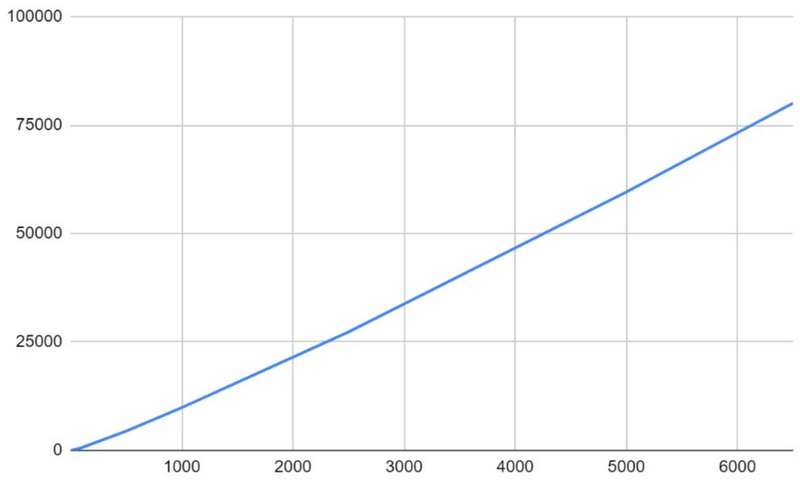
k = 4

EQUAL

Explanation: Equal has the worst case in our implementation of quicksort. This is due to, comparing while looking to greater or lower values.

AVERAGE CASE

INPUT SIZE	COUNT
1	0
10	30
50	266
100	632
500	4432
1000	9864
2500	27304
5000	59608
6500	80080



Assume k = 4, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 4 = 4 * 10 ^5

4 * 10^5 >= 9864

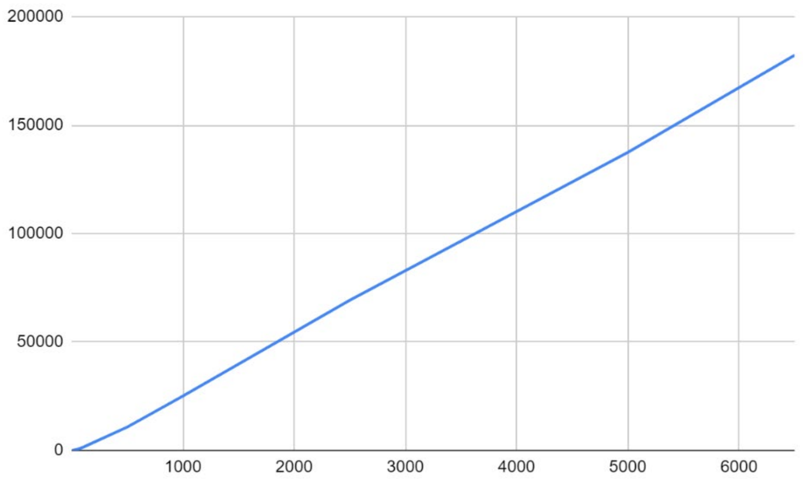
k = 4

SEMI - SORTED

Explanation: Semi - Sorted arrays are worse than random arrays and better than fully sorted arrays. That's why the time complexity is between n log n and n square.

AVERAGE CASE

INPUT SIZE	COUNT
1	0
10	43
50	442
100	1365
500	10753
1000	25042
2500	69269
5000	137454
6500	182258



Assume k = 5, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 5 = 5 * 10 ^5

5 * 10^5 >= 25042

k = 5

CONCLUSION FOR QUICK SORT (PIVOT AS THE FIRST ELEMENT)

The Theoretical Analysis and the Empirical Analysis are supporting each other and there is a harmony between them.

Graphs of the empirical analysis shows that there are little differences than the theoretical analysis but these graphs can be accepted if we assume that there is a tolerance for the empirical analysis.

This tolerance can be accepted because random variables are generated in each array and they might be different from each other.

To conclude, the quicksort algorithm has a time complexity of O(nlogn) for average and the best cases and O(n ^2) for the worst case.

APPLICATION OF QUICK SORT (MEDIAN OF THREE)

This approach is different from the original quicksort in only one stage, choosing the pivot. Since choosing the first or the last element as the pivot will cause the worst case in sorted arrays and also it will increase the time taken for random and semi-sorted arrays, the median of three allows the algorithm to find a better optimized pivot.

Pivot selection can be done by choosing the first, middle and last elements in the array. Median of these three elements will be the pivot that will improve the time complexity of the algorithm.

THEORETICAL ANALYSIS

Every step and theoretical explanation is the same except the pivot selection stage. This approach improves the time complexity in a good way but time complexity will not change, only the coefficient k will be decreased due to the median approach.

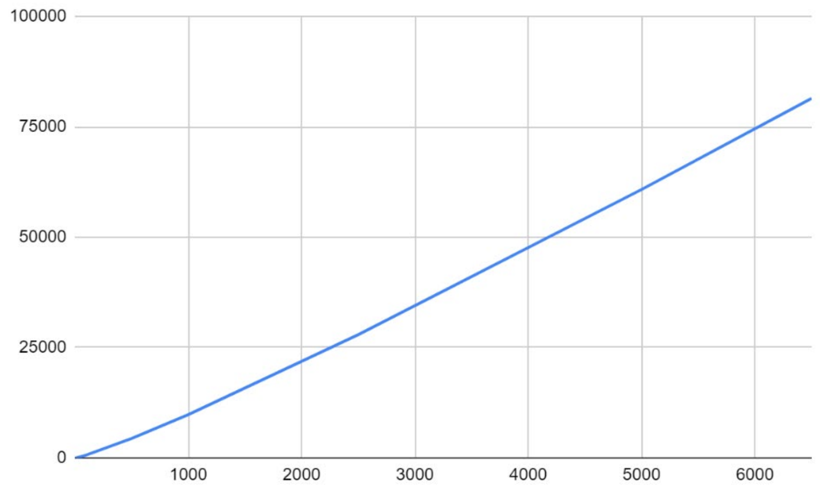
EMPIRICAL ANALYSIS

SORTED

Explanation: When the median of three is applied to the quicksort algorithm, the median which is the most suitable pivot for the array is at the center of the array. That's why the array will be sorted in a better time.

AVERAGE CASE

INPUT SIZE	COUNT
1	0
10	34
50	276
100	652
500	4403
1000	9807
2500	27914
5000	60824
6500	81443



Assume k = 4, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 4 = 4 * 10 ^5

4 * 10^5 >= 9807

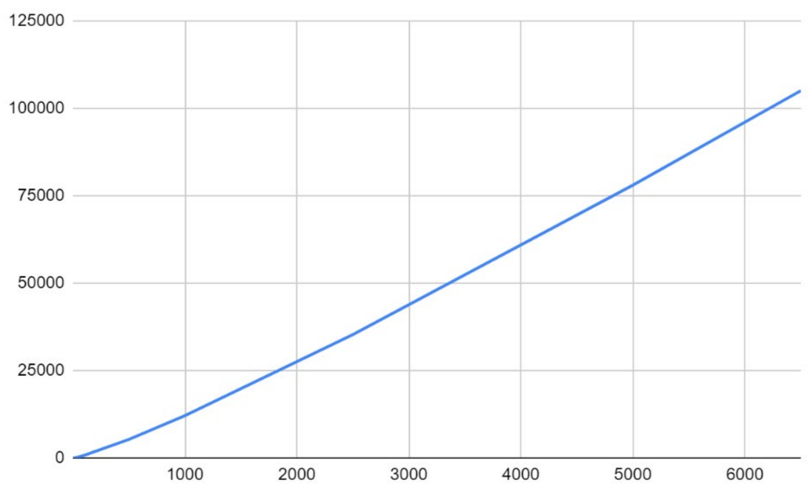
k = 4

REVERSE SORTED

Explanation: When the median of three is applied to the quicksort algorithm, the median which is the most suitable pivot for the array is at the center of the array. That's why the array will be sorted in a better time.

AVERAGE CASE

INPUT SIZE	COUNT
1	0
10	31
50	301
100	736
500	5318
1000	12110
2500	35296
5000	78053
6500	105083



Assume k = 4, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 4 = 4 * 10 ^5

4 * 10^5 >= 12110

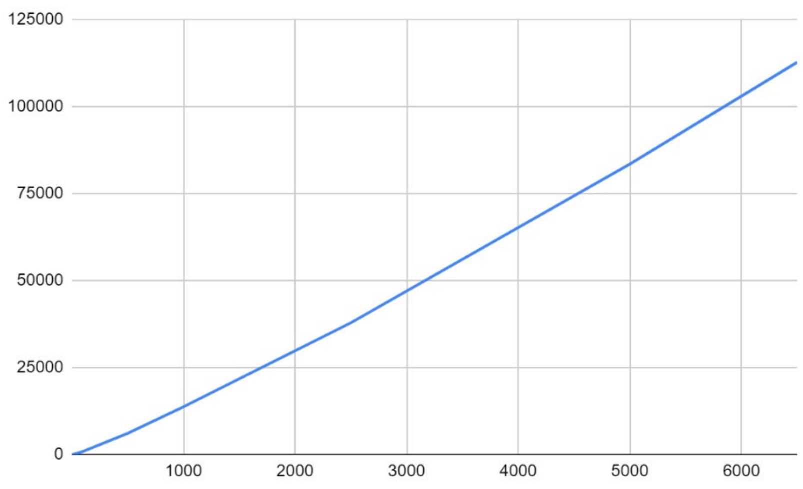
k = 4

RANDOM

Explanation: The time taken will be reduced but the time complexity will not change.

AVERAGE CASE

INPUT SIZE	COUNT
1	0
10	36
50	363
100	844
500	6069
1000	13695
2500	37853
5000	83493
6500	112784



Assume k = 4, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 4 = 4 * 10 ^5

4 * 10^5 >= 13695

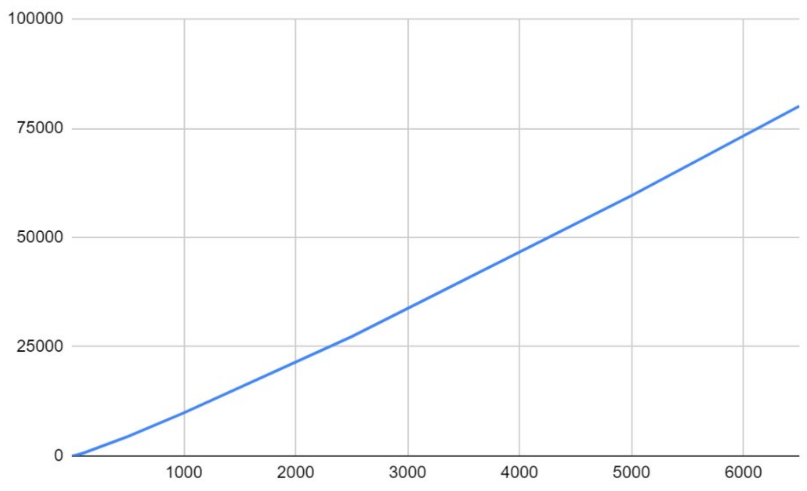
k = 4

EQUAL

Explanation: There will be no change, but if it was an original quicksort algorithm it was going to be n log n instead of n square. (Our algorithm is better optimized than the original one.)

BEST CASE (WORST CASE IN ORIGINAL QUICKSORT ALGORITHM)

INPUT SIZE	COUNT
1	0
10	30
50	266
100	632
500	4432
1000	9864
2500	27304
5000	59608
6500	80080



Assume k = 4, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 4 = 4 * 10 ^5

4 * 10^5 >= 9864

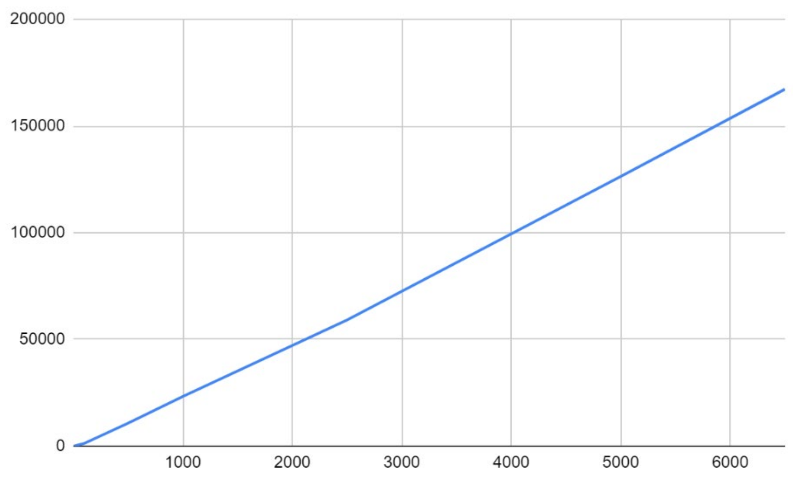
k = 4

SEMI - SORTED

Explanation: Improved by the median of three approach pivot selection and became the best possible case against random except equal.

AVERAGE (BETTER THAN RANDOM, BEST POSSIBLE TIME COMPLEXITY EXCEPT EQUAL)

INPUT SIZE	COUNT
1	0
10	49
50	670
100	1309
500	10782
1000	23283
2500	59085
5000	126404
6500	167237



Assume k = 5, Theoretical time complexity = O(nlogn)

For 1000, 1000 * log (1000) = 10^5 * 5 = 5 * 10 ^5

5 * 10^5 >= 23283

k = 5

Special situations for quicksort:

Sorted: In a sorted array of n elements, quicksort will use brute force approach if the pivot is being selected as the leftmost element of the array. This makes the time complexity to become O(n^2) which is the worst time complexity for quicksort.

Reverse Sorted: In a sorted array of n elements, the same situation occurs in reverse sorted, like sorted that makes the time complexity O(n^2) which is the worst situation.

Special situation for our implementation:

Equal: Normal time complexity is O(n^2) for equal but our implementation and partition function is working more optimized than the original quicksort algorithm.

Special situations for median of three approach quicksort for our implementation:

Equal: Normal time complexity is O(n^2) for equal but our implementation and partition function is working more optimized than the original quicksort algorithm.

Best Case:

Median of three Sorted and Reverse Sorted (Median of three): Since the array is sorted, when the middle element of the array is chosen, the array will become the perfect time complexity for quicksort since all the elements are in the correct location.

Random (Leftmost pivot selection): This could be the best or average case for the quicksort algorithm but the results are more commonly close to the median of three sorted and reverse sorted applications.

Equal: Normally equal would be the same as sorted and reverse sorted. But the partition algorithm that we have used in our quicksort algorithm is improving the time complexity and making it O(nlogn).

CONCLUSION FOR QUICK SORT (MEDIAN OF THREE)

The Theoretical Analysis and the Empirical Analysis are supporting each other and there is a harmony between them.

Graphs of the empirical analysis shows that there are little differences than the theoretical analysis but these graphs can be accepted if we assume that there is a tolerance for the empirical analysis.

This tolerance should be accepted because random variables are generated in each array and they might be different from each other.

Median of three Approach changes the pivot as the median of left, right and middle elements of the array.

Since sorted arrays are worst case in the leftmost pivot selection, median of three approach improves the time complexity to O(nlogn) which makes the quicksort more applicable and makes it O(nlogn) time complexity function.

In many ways, the median of three approach should be used in quicksort application.

To conclude, Median of three approach has O(nlogn) time complexity.

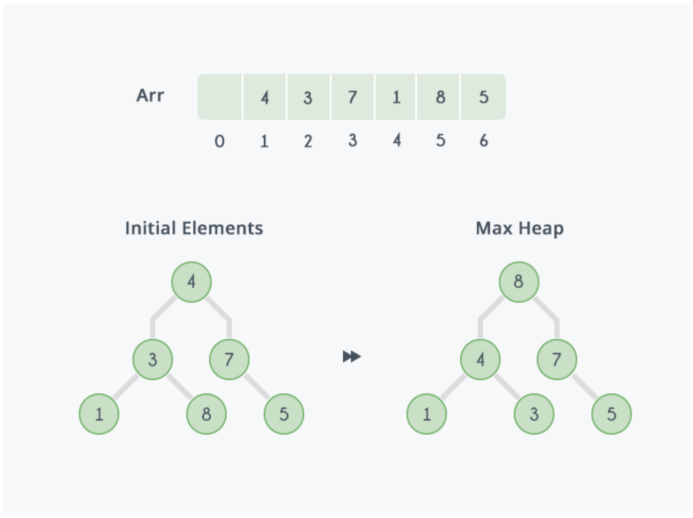
HEAP SORT

DEFINITION OF HEAP SORT

Heap sort is a comparison based sorting algorithm based on heap structure. It takes an unsorted array and converts it into a max/min heap. By replacing the root node and placing it to the last available index of array, this algorithm sorts the array.This process can be summarized as: building a heap, removing the node on top, placing its value into the last available index and heapifying it again; untill there are no more nodes in the heap. After this process, array will be sorted.

APPLICATION OF HEAP SORT

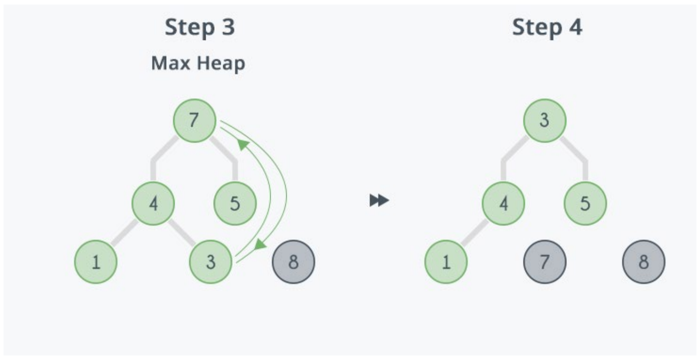
Application of Heap Sort can be listed as:

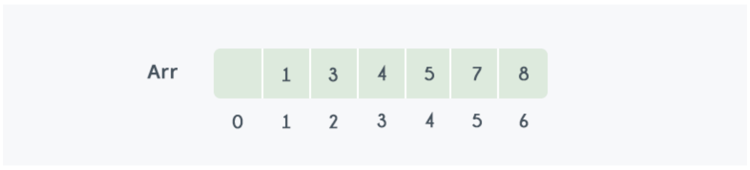


1 - BUILDING A HEAP

2 - REPLACING THE ROOT WITH THE LAST NODE OF HEAP AND REDUCING THE HEAP SIZE BY 1, HEAPIFYING THE ROOT OF THE TREE AGAIN

3 - REPEATING STEP 2 WHILE THERE IS AT LEAST 1 NODE IN THE HEAP.





After these steps, array will be sorted.

ANALYSIS OF HEAP SORT

THEORETICAL ANALYSIS

Theoretical analysis of heap sort in worst case can be handled in two stages:

1. **Building a heap for a given list of n keys.**
   Initially, an unranked heap will be created. In order to rank the heap, last node with at least 1 child will be found and compared to it's children values. This comparison will be counted as the basic operation to compute runtime. After this node's comparison has done, previous nodes with children will be compared to their children, etc. Thus we will obtain a formula:

$$C(n) = \sum_{i=0}^{h-1} 2(h-i) * 2^i = 2\big(n - log_2(n+1)\big) \in \theta(n)$$

$$h{:}\, height\ of\ the\ tree,\ 2^i{:}\, number\ of\ nodes\ at\ level\ i.$$

Result of this formula will yield to θ(n).

2. **Root node removal and heapifying until there are no nore nodes to remove.**
   After ranking process of heap, root node of the heap will be removed. Then, heap will be heapified again with a basic operation as comparison of children. For this part, the following formula can be obtained:

$$C(n) = \sum_{i=1}^{n-1} 2log_2 i \in \theta(nlogn)$$

$$log_2 i{:}\, height\ of\ the\ tree\ for\ given\ node\ count\ n.$$

Result of this formula will yield to θ(nlogn).

<u>**Special Case for Heap Sort**</u>: If the given array has equal values, building a heap will take θ(n) time and node removal will take θ(1). This situation yields to a time complexity of O(n) but this is **only for equal lists**.

<u>If the list is made of distinct keys</u>: θ(nlogn) + θ(n), and time efficiency for heap sort can be found as O(nlogn) for all cases.

EMPIRICAL ANALYSIS

For empirical analysis of the heapsort algorithm, operation time counter has chosen as the metric. By using this metric, count of the basic operation execution will be computed.

For both stages of the analysis (building a heap and root node removal), basic operation will be the comparison of the node with it's children.

Heap sort algorithm has O(nlogn) time efficiency in all cases, according to the theoretical analysis.

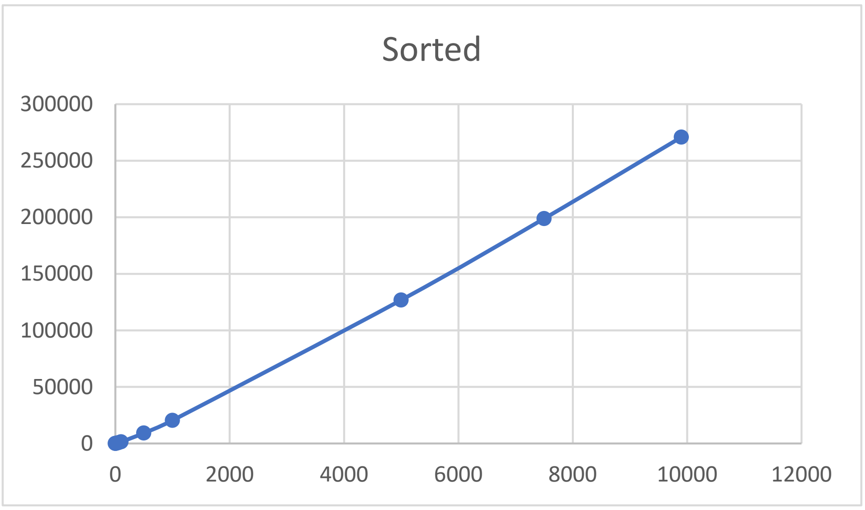
DIFFERENT TEST INPUTS FOR EMPIRICAL ANALYSIS

5 different input types with 9 different sizes for each has been used as test inputs.

Input types: already sorted, reverse sorted, equal values, randomly duplicated values, semi-sorted values.

Input sizes: 1, 10, 50, 100, 500, 1000, 5000, 7500, 9900.

INPUT SIZE	COUNT
1	0
10	70
50	590
100	1380
500	9208
1000	20416
5000	126864
7500	198880
9900	270876

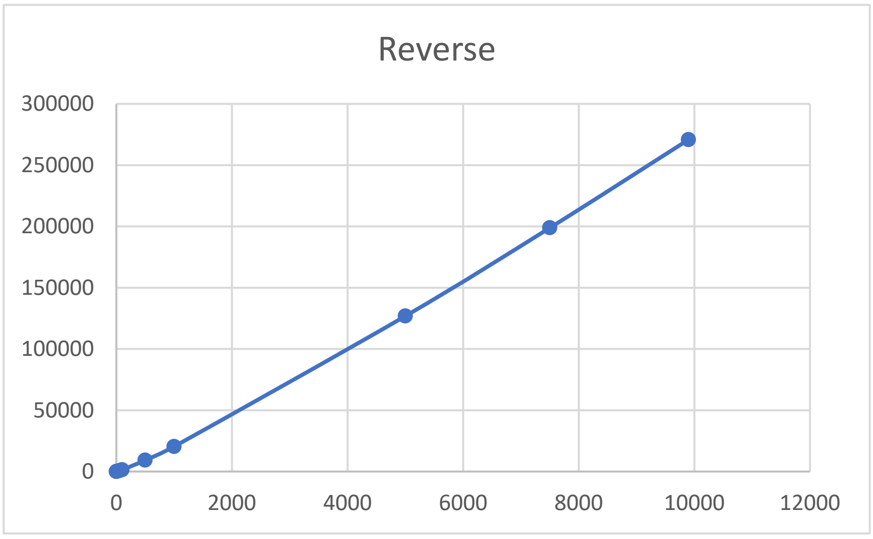


As seen in table, algorithm executed for O(k*nlogn) times, k = 2.107

$$for\ input\ size\ n = 100, \qquad O(2.107 * 100 * log100) = O(1399)$$

$$\theta(nlogn) \ \in \ O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	70
50	590
100	1380
500	9208
1000	20416
5000	126864
7500	198880
9900	270876

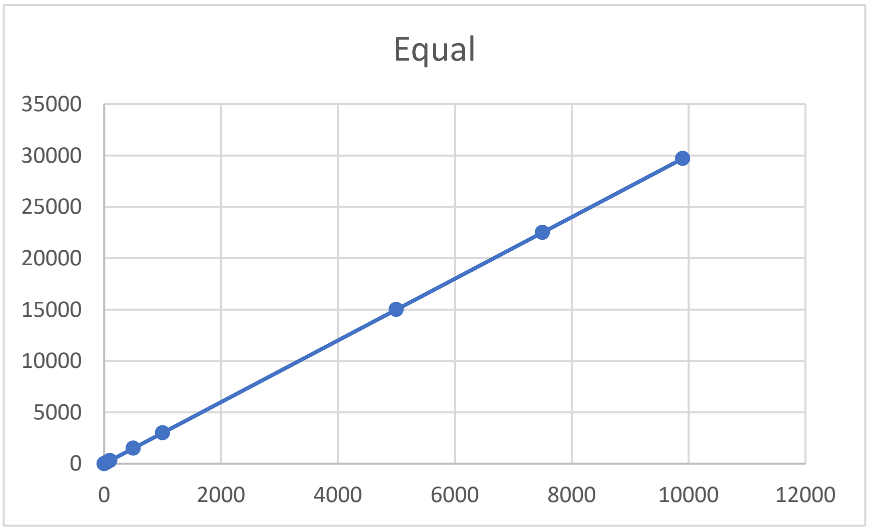


As seen in table, algorithm executed for O(k*nlogn) times, k as 2.107

$$for\ input\ size\ n = 100, \qquad O(2.107 * 100 * log100) = O(1399)$$

$$\theta(nlogn)\ \in\ O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	28
50	148
100	298
500	1498
1000	2998
5000	14998
7500	22498
9900	29698

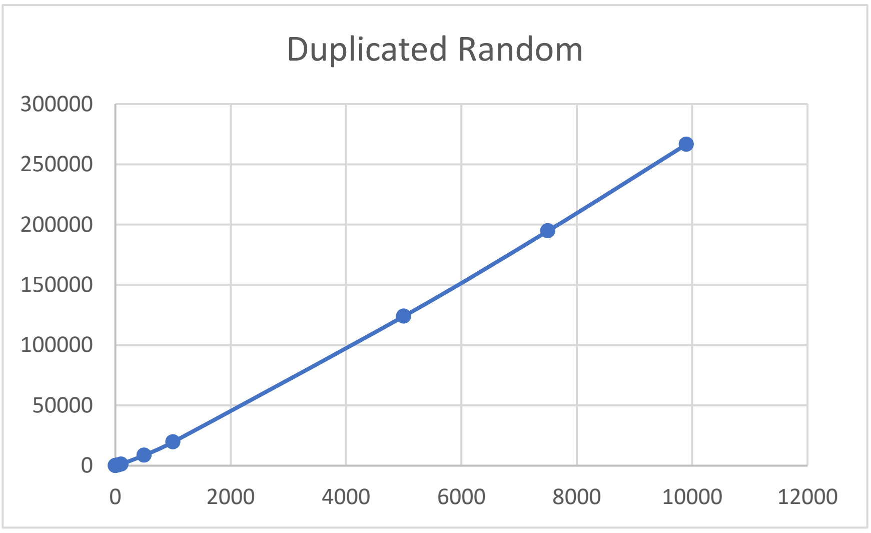


As seen in table, algorithm executed for O(k*n) times, k as approximate to 3.

$$for\ input\ size\ n = 100, \qquad O(3 * 100) = O(300)$$

$$Special\ case\ for\ Heap\ Sort, \qquad \theta(n)\ \in\ O(k * n)$$

INPUT SIZE	COUNT
1	0
10	60
50	496
100	1220
500	8634
1000	19606
5000	123916
7500	194726
9900	266550

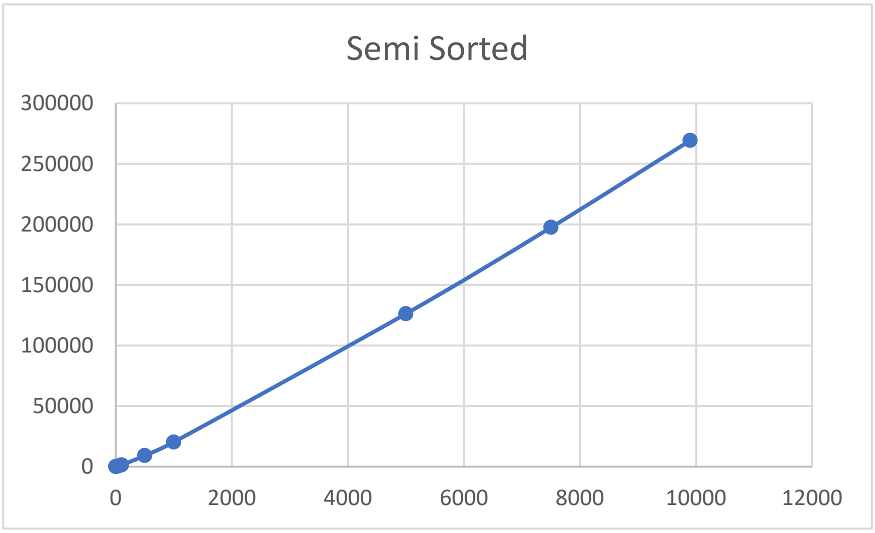


As seen in table, algorithm executed for O(k*nlogn) times, k as 2

$$for\ input\ size\ n = 100, \qquad O(2 * 100 * log100) = O(1328.77)$$

$$\theta(nlogn) \ \in \ O(k * nlogn)$$

INPUT SIZE	COUNT
1	0
10	70
50	582
100	1356
500	9144
1000	20316
5000	126244
7500	197490
9900	269272



As seen in table, algorithm executed for O(k*nlogn) times, k as 2.107

$$for\ input\ size\ n = 100, \qquad O(2.107 * 100 * log100) = O(1399)$$

$$\theta(nlogn) \ \in \ O(k * nlogn)$$

CONCLUSION

As seen above, algorithm has been analyzed and compared with theoretical and empirical results. Graphs and tables show us that results of empirical analysis are in harmony with the results of theoretical analysis.

Iteration counts from empirical analysis are in the range of the time complexity values from theoretical analysis:

for worst case:

$$\theta(nlogn) \in O(nlogn)$$

for average case:

$$\theta(nlogn) \in O(nlogn)$$

for best case:

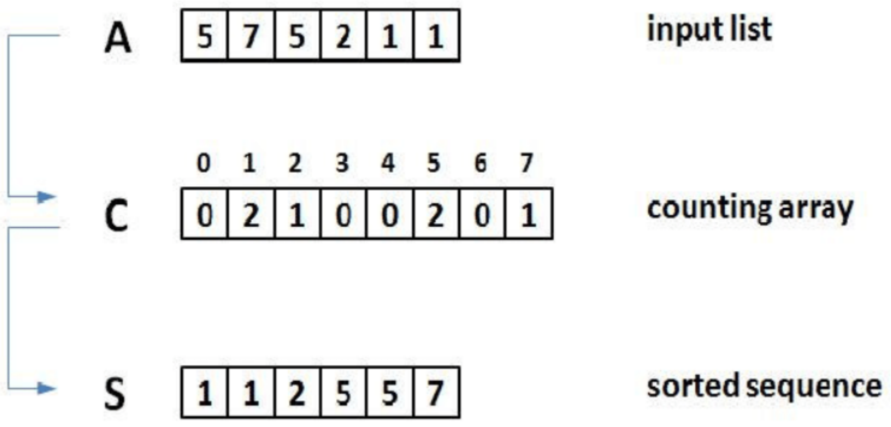
$$if\ list\ has\ distinct\ keys, \theta(nlogn) \in O(nlogn)$$

$$if\ list\ is\ equal\ keys, \theta(n) \in O(n)$$

COUNTING SORT

DEFINITON AND APPLICATION OF COUNTING SORT

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each distinct element in the array. The count is stored in an temprary array and the sorting is done by mapping the count as an index of the temprary array.



Example execution of counting sorting

ANALYSIS OF COUNTING SORT

THEORETICAL ANALYSIS

The algoithm as pseudocode,

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

TIME EFFICIENCY OF SORTING BY COUNTING

The time efficiency of counting method is same for best , average and worst cases.

In this method , algorithm uses simple loops for creating and sorting arrays therefore it can be analysed straightforwardly.

The loop which finds the total count of each unique element and stores into count array takes O(n).

The loop which finds the cumulative sum and stores it in count array itself takes O(k).

(“k” refers to value of maximum element in the unsorted array)

The loop which restores the elements to array and decreases the count of each element restored by 1 takes O(n)

The loop which sortes all elements into output array takes O(n)

$O(n) + O(n) + O(k) + O(n) \in O(n + k)$

- Therefore time complexity of this algorithm is the sum of these proccesses, O(n + k).

- This algorithm works effective with low sparsed inputs, it means, if the difference between minimum and maximum element in the list is great, count array is going to be created with a lot of unnecessary and unused zeros and the time complexity will increase.

- This sorting technique has O(n+k) time complexity for all cases and it uses linear amount of space relative to input size.

EMPIRICAL ANALYSIS

For empirical analysis of the counting sort algorithm, operation time counter has chosen as the metric. By using this metric, count of the basic operation execution will be computed.

For the empirical analysis of the algorithm, basic operation will be placing.

DIFFERENT TEST INPUTS FOR EMPIRICAL ANALYSIS

4 different input types with 9 different sizes for each has been used as test inputs.

Input types: already sorted, reverse sorted, equal values, randomly duplicated values.

Input sizes: 1, 10, 50, 100, 500, 1000, 5000, 7500, 9900.

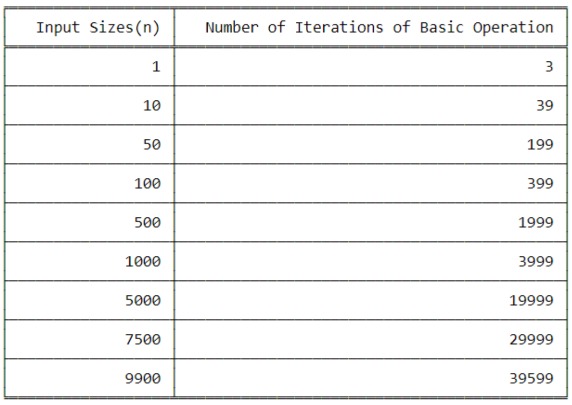
Testing with reverse sorted list

EX → reverse_sorted_1 =[1]

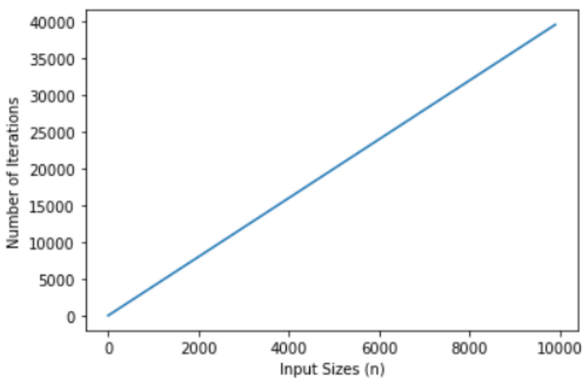
reverse_sorted_10 =[9,8,7,6,5,4,3,2,1,0]

reverse_sorted_50 =[49,48,47,46,45,44,.......0]

...



Input Sizes(n)	Number of Iterations of Basic Operation
1	3
10	39
50	199
100	399
500	1999
1000	3999
5000	19999
7500	29999
9900	39599

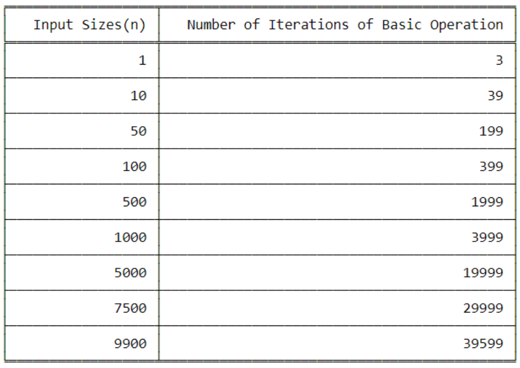


As seen in table and plot above, relation of number of iterations and input sizes is corresponding to each other linearly. Becaouse of the loops in program , number of iterations based on “n” and “k”. Asymotically O(n) + O(n) + O(k) + O(n) ∈ O(n + k) and in this case, due to maximum element is very close to input size , time complexity of the algorithm is O(n).

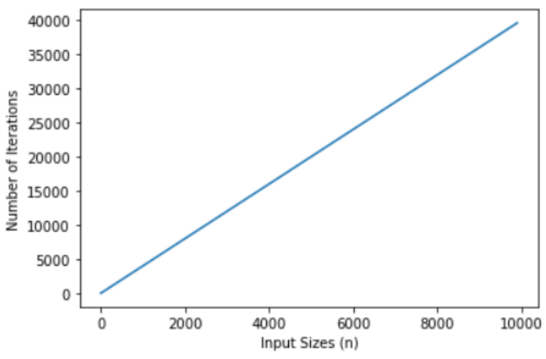
Testing with already sorted list

EX → reverse_sorted_10 = [9,8,7,6,5,4,3,2,1,0]

...



Input Sizes(n)	Number of Iterations of Basic Operation
1	3
10	39
50	199
100	399
500	1999
1000	3999
5000	19999
7500	29999
9900	39599



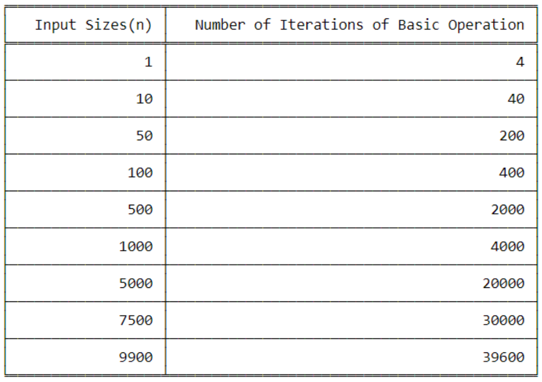
As seen in table and plot above, relation of number of iterations and input sizes is corresponding to each other linearly. Becaouse of the loops in program , number of iterations based on “n” and “k”. Asymotically O(n) + O(n) + O(k) + O(n) ∈ O(n + k) and in this case, due to maximum element is very close to input size , time complexity of the algorithm is O(n).

Testing with equal values in list

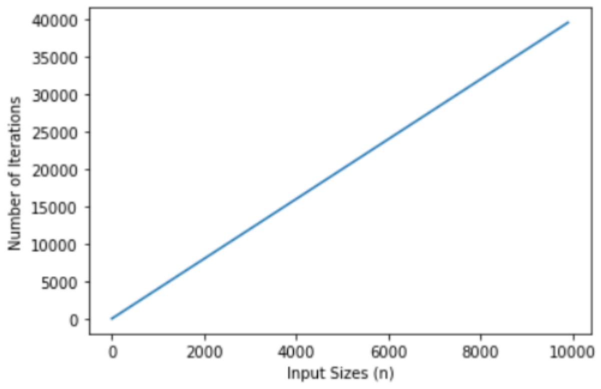
EX → equal_list_5 = [5,5,5,5,5]

equal_list_10 = [10,10,10,10,10,10,10,10,10,10]

…



Input Sizes(n)	Number of Iterations of Basic Operation
1	4
10	40
50	200
100	400
500	2000
1000	4000
5000	20000
7500	30000
9900	39600

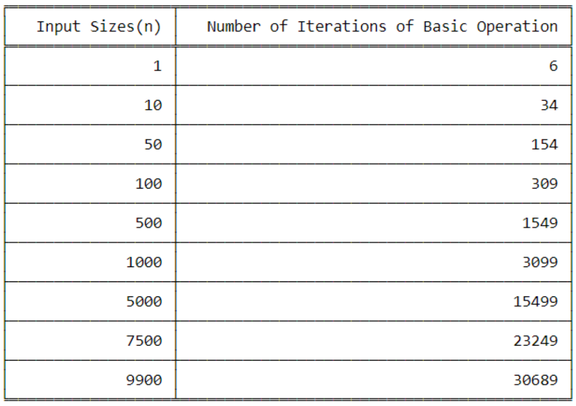


As seen in table and plot above, relation of number of iterations and input sizes is corresponding to each other linearly. Becaouse of the loops in program , number of iterations based on “n” and “k”. Asymotically O(n) + O(n) + O(k) + O(n) ∈ O(n + k) and in this case, due to maximum element is very close to input size , time complexity of the algorithm is O(n).

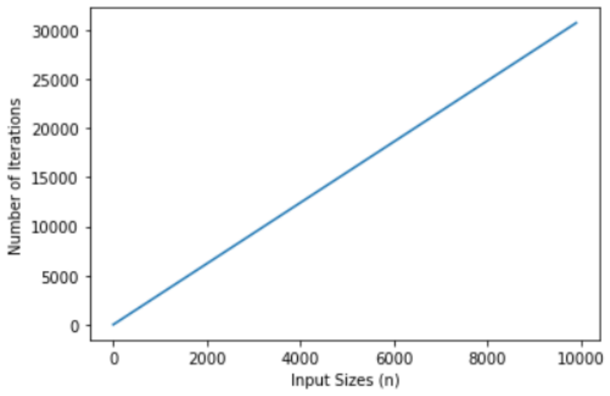
Testing with duplicated and random values in list

Ex →

…



Input Sizes(n)	Number of Iterations of Basic Operation
1	6
10	34
50	154
100	309
500	1549
1000	3099
5000	15499
7500	23249
9900	30689



As seen in table and plot above, relation of number of iterations and input sizes is corresponding to each other linearly. Becaouse of the loops in program , number of iterations based on “n” and “k”. Asymotically O(n) + O(n) + O(k) + O(n) ∈ O(n + k) and in this case, due to numbers are random , we can say time complexity of the algorithm is O(n + k).

Special Case for sorting by counting

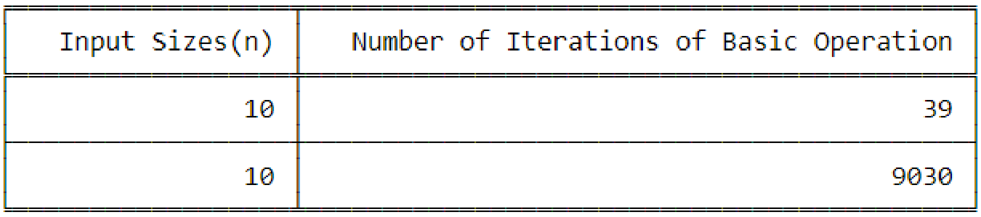
In this case ,

first input arr_1 = [0,6,4,5,7,8,9,3,1,2]

Second input arr_2 = [0,1,2,3,4,5,6,7,8,9000]

As known , sorting by counting has O(n+k) time complexity. In first input , value of maximum element is same with size of first array therefore O(n+k) = O(n + n) and O(n + n) ∈ O(n)

However , in second input , due to value of difference between maximum and minimum element is great (k>>n), although input sizes are same for two arrays, second case has more time complexity than first case O(n+k).



Input Sizes(n)	Number of Iterations of Basic Operation
10	39
10	9030

CONCLUSION

As seen above, algorithm has been analyzed and compared with theoretical and empirical results. Graphs and tables show us that results of empirical analysis are in harmony with the results of theoretical analysis.

Iteration counts from empirical analysis are in the range of the time complexity values from theoretical analysis:

for worst case:

$$\theta(n) \in O(n+k)$$

for average case:

$$\theta(n) \in O(n+k)$$

for best case:

$$\theta(n) \in O(n+k)$$

FINAL CONCLUSION

We wanted to compute the iteration count of the basic operations in loops and recursions of an algorithm.

ex: `for x = 0 to n ;` **>** For the given algorithm, total number of iteration will be: **n+z**

```
for x = 0 to n ;
        …
for x = 0 to z ;
        …
```

For empirical analysis of algorithms, operation time counter has chosen as the metric. By using this metric, count of basic operation executions were computed. Since computers are not running at the same speed, we thought that using time as a metric would be unstable to use in empirical analysis of sorting algorithms.

By using the results observed from empirical analysis, we obtained the tabled below:

Sorting Algorithm	Worst Case	Average Case	Best Case	Stable	In-place
Insertion	$O(n^2)$	$O(n^2)$	$O(n)$	**YES**	**YES**
Binary Insertion	$O(n^2)$	$O(n^2)$	$O(nlogn)$	**YES**	**YES**
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	**YES**	**NO**
Quick Sort (pivot 1st)	$O(n^2)$	$O(nlogn)$	$O(nlogn)$	**NO**	**YES**
Quick Sort (med. of three)	$O(n^2)$	$O(nlogn)$	$O(nlogn)$	**NO**	**YES**
Heap Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$, $O(n)$ $for\ equal\ keys$	**NO**	**YES**
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	**YES**	**NO**

Depending on the table above,

According to their time complexity values, best four sorting algorithms can be listed as:

- ✓ Counting Sort (with O(n + k) in all cases)
- ✓ Merge Sort (with O(nlogn) in all cases)
- ✓ Heap Sort (with O(nlogn) in all cases)
- ✓ Quick Sort (with O(nlogn) in best case and average cases)

Since they are not in-place, merge sort and counting sort use extra space. If we worry about memory and worst cases, we can say that heap sort is an optimum algorithm for sorting. On the other hand, we may use merge sort, quick sort and counting sort.