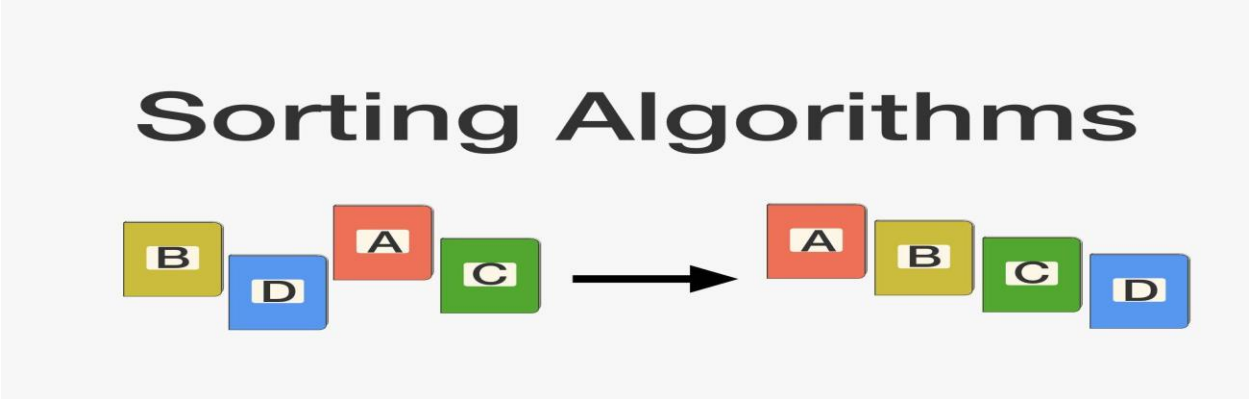
Comparison-based Sorting Algorithms



DOCUMENTATION REPORT

PROGRAMMING PROJECT 1

ITCS 6114 – Algorithms and Data Structures

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

DEVRAJ PATEL

801076509

UNC CHARLOTTE
College of Computing and Informatics

Table of Contents

Comparison-Based Sorting Algorithms

INTRODUCTION:

Here we have implemented 4 sorting algorithms

1) Insertion Sort
2) Merge Sort
3) In-Place Quick Sort (Any random number as pivot or the first or last item is taken as pivot)
4) Modified Quick Sort
    a. Use Median-of-three as pivot.
    b. For small subproblem size (≤ 10), you must use insertion sort.

Execution Instructions:

1) These algorithms are executed for different input sizes (e.g. n = 500, 1000, 2000, 4000, 5000, 10000, 20000, 30000, 40000 and 50,000). Numbers are randomly generated for the input array. The execution time has been recorded and average time has been taken of various test cases. These sorting algorithms are then compared, and graphs have been generated. The algorithms are compared for the same data set for more accurate comparison.

2) Performance for the case where input is already sorted, and reverse sorted also has been performed and recorded for observation.

NOTE: Along with the 4 files for each sorting algorithms we have created a 5^{th} file for taking in the input size as desired by the user and it generates random numbers in the list and same list is used by all the 4 algorithms for accurate time measurement results.

Insertion Sort:

About: Insertion sort is a sorting algorithm that makes the sorted list one item at a time. It is less efficient on large lists and better for small lists.

Code:

```
import time

import sys

sys.setrecursionlimit(2000)

try:

    from global_var import numbers

    from global_var import input_size

except:

    print "Please run the code 'randm.py' and enter the input size, then run this #again(Because we want the same dataset for all 4 sort techniques)"

    exit()

# print input_size

# print numbers

def insertion_Sort(the_list):   #Function for sorting numbers

   for index in range(1,len(the_list)):



     cv = the_list[index]

     cp = index
```

```
    while cp>0 and the_list[cp-1]>cv:   #while loop for comparing values
        the_list[cp]=the_list[cp-1]
        cp = cp-1

    the_list[cp]=cv

the_list = []   #Define an empty list
the_list = numbers
print ("The " + str(input_size) + " Randomly generated numbers are ")
print(numbers)
start=time.time()   #For Calculating the time taken by the Insertion Sort Algorithm
insertion_Sort(the_list)   #Feeds the values to the function
end=time.time()
print("The List of numbers after being sorted is as follows ")
print(the_list)

print ("The numbers were sorted using Insertion Sort in " + str((end-start) * 1000)  + " ms time.")
print("\nSpecial Case 'a': Already Sorted Array")
start=time.time()
insertion_Sort(the_list)   #Feeds the values to the function
```

```
end=time.time()

print(the_list)

print ("Already Sorted : The numbers were sorted using Insertion Sort in " + str((end-start) * 1000)  + " ms time.\n")

print("Special Case 'b': Reverse Sorted array")

the_list = the_list[::-1]   #This reverses the list

print the_list

start=time.time()

insertion_Sort(the_list)   #Enters the list in the loop

end= time.time()

print the_list

print ("Reverse Sorted : The numbers were sorted using Insertion Sort in " + str((end-start) * 1000)  + " ms time.")
```

Observations: Insertion Sort works pretty good for small list and it is very fast for sorted list as input and takes time for reverse sorted list. In short if the list is small Insertion is perfect.

Merge Sort:

About: Merge sort is an efficient and stable sort. It is basically divide and conquer algorithm. It divides the list and keeps on doing it until its completed divided and then merging is done along with sorting of the elements. It works pretty well for all sorting

Code:

```
import time
import sys
sys.setrecursionlimit(2000)
try:
    from global_var import numbers
    from global_var import input_size
except:
    print "Please run the code 'randm.py' and enter the input size, then run this #again(Because we want the same dataset for all 4 sort techniques)"
    exit()


def merge_Sort(the_list):   #Function for sorting numbers
    #print("Split ",the_list)
    if len(the_list)>1:
        mid = len(the_list)//2   #Floor Division
        leftside = the_list[:mid]   #All elements on the left of mid
        rightside = the_list[mid:]   #All elements on the right of mid
        merge_Sort(leftside)
```

```
    merge_Sort(rightside)


    i=0   #Index for the Left Array
    j=0   #Index for the Right Array
    k=0   #Index for the Merged Array
    while i < len(leftside) and j < len(rightside):   #Comparing the elements of both sides
        if leftside[i] < rightside[j]:
            the_list[k]=leftside[i]
            i=i+1
        else:
            the_list[k]=rightside[j]
            j=j+1
        k=k+1


    while i < len(leftside):   #Copy the remaining elements of left Side
        the_list[k]=leftside[i]
        i=i+1
        k=k+1
```

```
        while j < len(rightside):   #Copy the remaining elements of Right Side
            the_list[k]=rightside[j]
            j=j+1
            k=k+1
    #print("Merge ",the_list)

the_list = numbers
print ("The " + str(input_size) + " Randomly generated numbers are ")
print(numbers)
start=time.time()   #For Calculating the time taken by the Insertion Sort Algorithm

merge_Sort(the_list)   #Feeds the values to the function
print("The List of numbers after being sorted is as follows ")
end=time.time()
print(the_list)
print ("The numbers were sorted using Merge Sort in " + str((end-start) * 1000)  +
" ms time.")
```

```
print("\nSpecial Case 'a': Already Sorted Array")

start=time.time()

merge_Sort(the_list)   #Feeds the values to the function


end=time.time()

print(the_list)

print ("Already Sorted : The numbers were sorted using Merge Sort in " + str((end-start) * 1000)  + " ms time.\n")

print("Special Case 'b': Reverse Sorted array")

the_list = the_list[::-1]   #Reverses the list

print the_list

start_1=time.time()

merge_Sort(the_list)

end_1= time.time()

print the_list

print ("Reverse Sorted : The numbers were sorted using Merge Sort in " + str((end_1-start_1) * 1000)  + " ms time.")
```

Observations: It works with the run time of O(nlogn). Merge Sort works well for large data sets

In Place Quick Sort:

About: In place Quick Sort is a sorting algorithm which sorts the list among itself without creating any additional array.

Code:

```
import time
import sys
sys.setrecursionlimit(2000)
try:
    from global_var import numbers
    from global_var import input_size
except:
    print "Please run the code 'randm.py' and enter the input size, then run this #again(Because we want the same dataset for all 4 sort techniques)"
    exit()
def quick_Sort(the_list):   #Function for sorting numbers
    if len(the_list) <= 1:   #If list has 1 number or less it will return it
        return the_list
    return divide(the_list,0,len(the_list)-1)   #passing the call to divide function
```

```
def divide(the_list,start_of_list,end_of_list):
    pivot = the_list[end_of_list]   #last element is considered as the pivot
    partition = start_of_list
    if start_of_list < end_of_list:
        for i in range(start_of_list,end_of_list+1):   #loops through
            if the_list[i] <= pivot:
                the_list[partition], the_list[i] = the_list[i], the_list[partition]
                if i != end_of_list:
                    partition += 1
        divide(the_list,start_of_list,partition-1)   #recursion until list is sorted
        divide(the_list,partition+1,end_of_list)
    return the_list


def main():
    the_list = numbers
    print ("The " + str(input_size) + " Randomly generated numbers are ")
    print(numbers)
    start=time.time()   #For Calculating the time taken by the Inplace Quick Sort
Algorithm
```

```
    quick_Sort(the_list)   #Feeds the values to the function
    end = time.time()
    print("The List of numbers after being sorted is as follows ")
    print(the_list)   #sorted list
    print ("The numbers were sorted using Inplace Quick Sort in " + str((end-start) *
1000)  + " ms time.")
    reverse_list = the_list[::-1]   #reverses the list
    print(reverse_list)


    print("\nSpecial Case 'a': Already Sorted Array")
    start = time.time()
    quick_Sort(the_list)  # Feeds the values to the function
    end = time.time()
    print(the_list)
    print ("Already Sorted : The numbers were sorted using Inplace Quick Sort in "
+ str(
        (end - start) * 1000) + " ms time.\n")
    print("Special Case 'b': Reverse Sorted array")
    the_list = the_list[::-1]   #Reverses the list
    print the_list
```

```
    start = time.time()
    quick_Sort(the_list)
    end = time.time()
    print the_list
    print ("Reverse Sorted : The numbers were sorted using Inplace Quick Sort in "
+ str(
        (end - start) * 1000) + " ms time.")
main()
```

Observations: In place quick sort sorts the array in its own place. It's not stable. In general Quick sort worked pretty well for large input sizes. Although in scenarios were list is already sorted or reverse sorted the time consumption and recursions exceeded too much.

Modified Quick Sort:

About: In modified Quick Sort here we are using median of three as pivot for better results and if the problem size is less than or equal to 10 insertion sort will be used.

Code:

```
import time
import sys
sys.setrecursionlimit(2000)
```

```
try:
    from global_var import numbers
    from global_var import input_size
except:
    print "Please run the code 'randm.py' and enter the input size, then run this #again(Because we want the same dataset for all 4 sort techniques)"
    exit()


def quick_Sort(the_list):   #the list is passed in the function from main
   recursive_quick_Sort(the_list,0,len(the_list)-1)


def recursive_quick_Sort(the_list,ele_first,ele_last):   #Recursively calls the funstion to sort
   if ele_first<ele_last:


       divide_here = divided(the_list,ele_first,ele_last)


       recursive_quick_Sort(the_list,ele_first,divide_here-1)   #to sort the left side from partition
```

```
    recursive_quick_Sort(the_list,divide_here+1,ele_last)   #to sort the right side from partition


def divided(the_list,ele_first,ele_last):   #For dividing
  index_of_pivot = median_decider(the_list, ele_first, ele_last, (ele_first + ele_last) // 2)   #used to decide the median
  the_list[ele_first], the_list[index_of_pivot] = the_list[index_of_pivot], the_list[ele_first]
  value_of_pivot = the_list[ele_first]

  l_s = ele_first
  r_s = ele_last

  done = False
  while not done:

    while l_s <= r_s and the_list[l_s] <= value_of_pivot:   #Check if it's less than the pivot
        l_s = l_s + 1
```

```
        while the_list[r_s] >= value_of_pivot and r_s >= l_s:   #check if it's greater
than the pivot
            r_s = r_s -1


        if r_s < l_s:
            done = True
        else:
            var_t = the_list[l_s]
            the_list[l_s] = the_list[r_s]
            the_list[r_s] = var_t


    var_t = value_of_pivot
    the_list[the_list.index(value_of_pivot)] = the_list[r_s]
    the_list[r_s] = var_t


    return r_s


def median_decider(a, i, j, b):   #Function for median
  if a[i] < a[j]:
```

```
    return j if a[j] < a[b] else b
  else:
    return i if a[i] < a[b] else b


def insertion_Sort(the_list):   #If the input size is less than or equal to 10 insertion sort will be called/used
   for index in range(1,len(the_list)):

     cv = the_list[index]
     index_pos = index

     while index_pos>0 and the_list[index_pos-1]>cv:   #comparing values
        the_list[index_pos]=the_list[index_pos-1]
        index_pos = index_pos-1

     the_list[index_pos]=cv

the_list = []   #Define an empty list
```

```
the_list = numbers
print ("The " + str(input_size) + " Randomly generated numbers are ")
print(numbers)
if(int(input_size)<=10):   #If input size is less than 10 it will use insertion sort
   print("Insertion Sort is used to sort numbers of input size less than or equal to 10")
   insertion_Sort(the_list)
else:
   print("Quick Sort is used to sort numbers of input size greater than or equal to 10")
start=time.time()   #Calculates the time
quick_Sort(the_list)

end=time.time()
print(the_list)
print ("The numbers were sorted in " + str((end-start) * 1000)  + " ms time.")

print("\nSpecial Case 'a': Already Sorted Array")
start = time.time()
quick_Sort(the_list)  # Feeds the values to the function
```

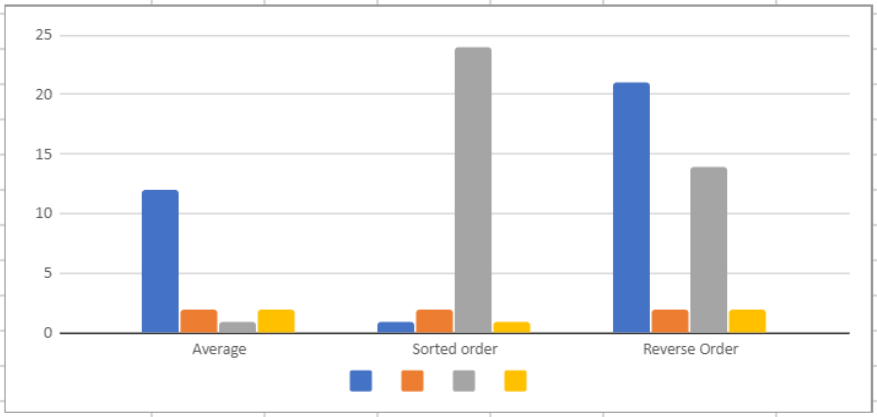
```
end = time.time()
print(the_list)
print ("Already Sorted : The numbers were sorted using Insertion Sort in " + str(
    (end - start) * 1000) + " ms time.\n")
print("Special Case 'b': Reverse Sorted array")
the_list = the_list[::-1]   #It reverses the list
print the_list
start = time.time()
quick_Sort(the_list)
end = time.time()
print the_list
print ("Reverse Sorted : The numbers were sorted using Insertion Sort in " + str(
    (end - start) * 1000) + " ms time.")
```

Observations: Here for a better value of pivot median was taken of the first, last and the middle element and that was chosen as pivot. This makes the execution a bit better compared to normal quick sort. Also as insertion sort was also added for input size less than or equal to 10 those values were sorted pretty fast. Quick sort works well for large random data sets.

Graphs for Various Input Sizes

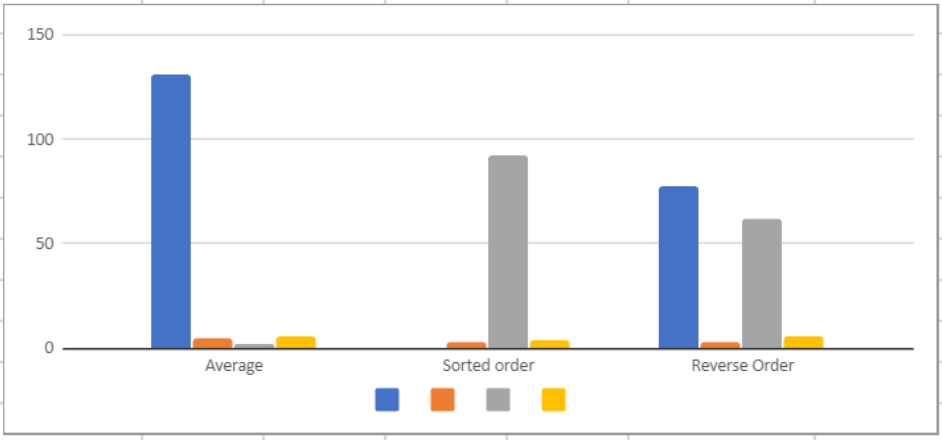
1) For Input Size 500

1	size=500 ms					
2	Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
3	Insertion Sort	13	11	10	12	1	21
4	Merge Sort	2	3	2	2	2	2
5	In-place Quick	1	1	2	1	24	14
6	Modified Quick	2	2	2	2	1	2



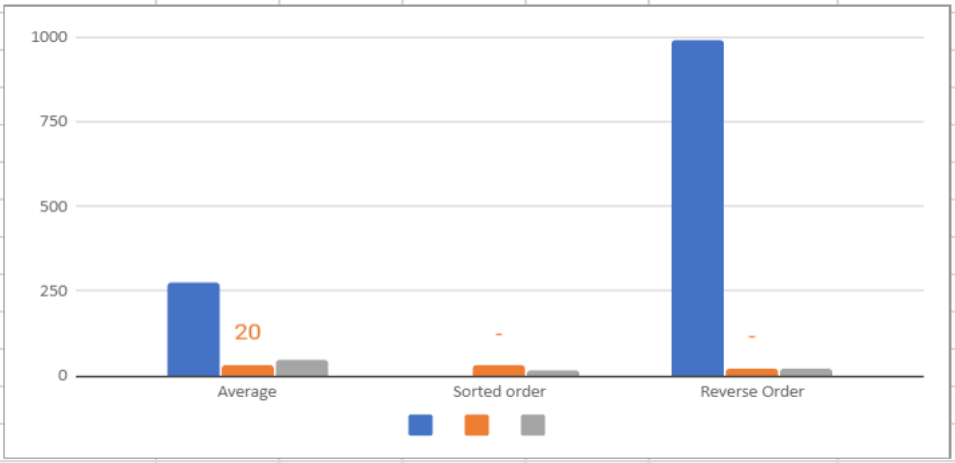
2) For Input Size 1000

22	size=1000 ms					
23	Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
24	Insertion Sort	141	123	130	131	0	78
25	Merge Sort	5	5	4	5	3	3
26	In-place Quick	2	2	2	2	92	62
27	Modified Quick	6	5	7	6	4	6



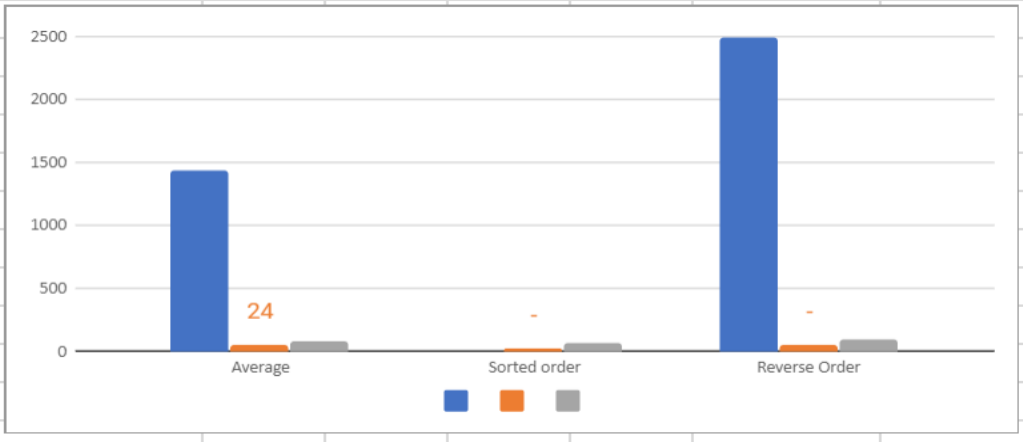
3) For Input Size 2000

size=2000 ms						
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	222	254	355	277	1	993
Merge Sort	24	38	39	34	31	23
In-place Quick	20	21	16	20	-	-
Modified Quick	40	61	33	45	14	19



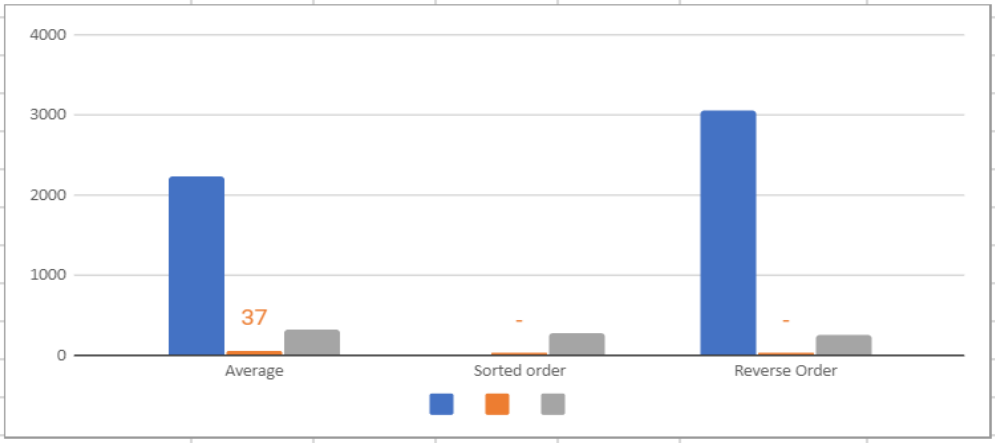
4) For Input Size 4000

size=4000 ms						
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	1414	1366	1524	1434	1	2485
Merge Sort	55	55	65	58	29	59
In-place Quick	25	36	12	24	-	-
Modified Quick	66	82	84	77	66	95



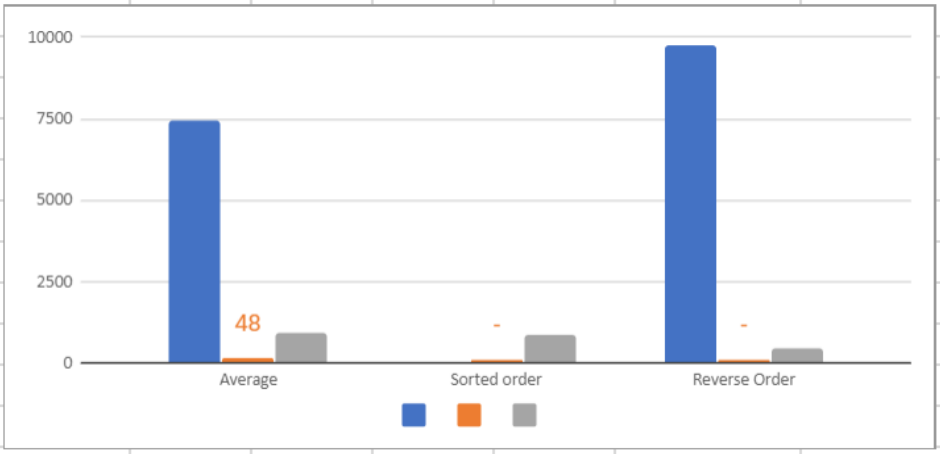
5) For Input Size 5000

size=5000 ms						
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	2383	2351	1934	2222	1	3047
Merge Sort	39	55	59	51	35	34
In-place Quick	45	31	36	37	-	-
Modified Quick	285	354	299	312	281	246



6) For Input Size 10,0007

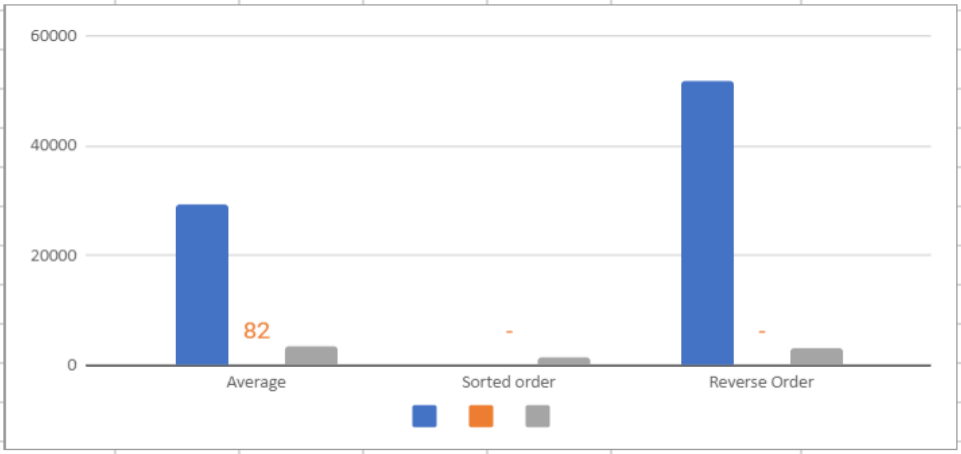
size=10000 ms						
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	8636	6866	6911	7471	3	9759
Merge Sort	99	131	150	126	80	87
In-place Quick	33	60	50	48	-	-
Modified Quick	821	1003	899	907	866	448



7) For Input Size 20,000

size=20000 ms

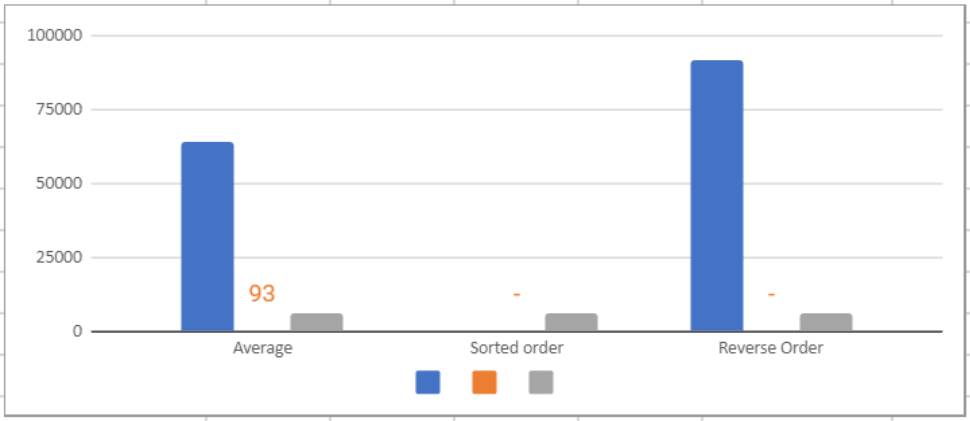
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	30212	31002	26456	29223	4	51814
Merge Sort	151	171	169	163	171	110
In-place Quick	78	85	82	82	-	-
Modified Quick	3752	4169	2966	3636	1641	3004



8) For Input Size 30,000

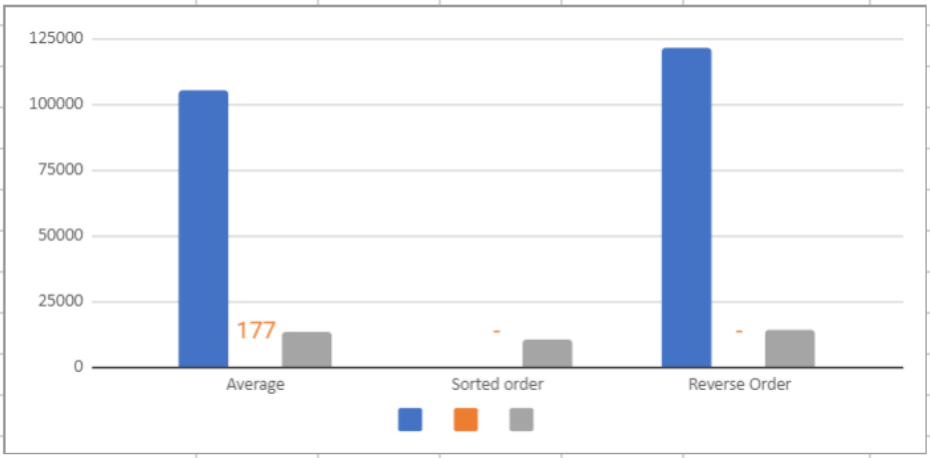
size=30000 ms

Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	60546	56648	74212	63802	9	91450
Merge Sort	260	240	182	227	182	178
In-place Quick	96	90	92	93	-	-
Modified Quick	4788	6464	7207	6153	5877	5699



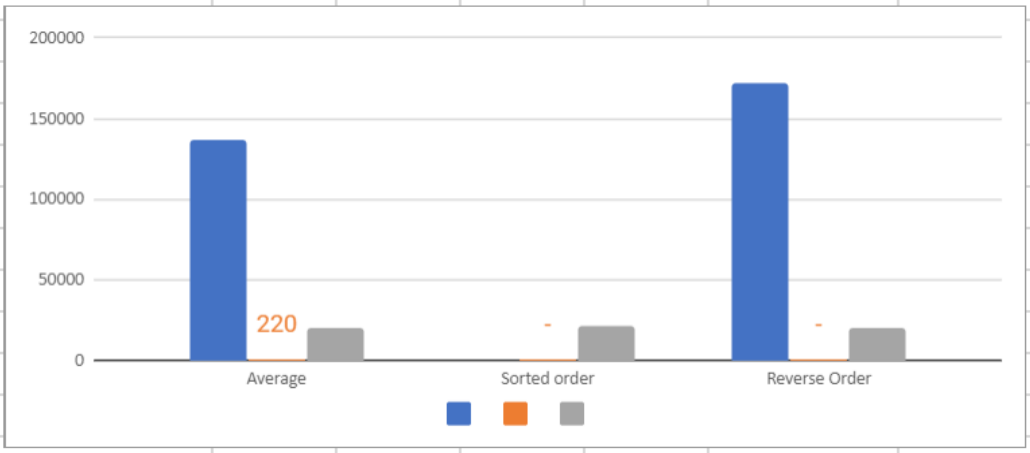
9) For Input Size 40,000

size=40000 ms						
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	104221	100822	110264	105102	15	121646
Merge Sort	305	300	305	304	275	260
In-place Quick	177	190	165	177	-	-
Modified Quick	13521	11562	15361	13472	10264	14115



10) For Input Size 50,000

size=50000 ms						
Sorting Tech	Time 1	Time 2	Time 3	Average	Sorted order	Reverse Order
Insertion Sort	130121	141467	139789	137117	15	171568
Merge Sort	335	366	377	359	335	342
In-place Quick	226	220	215	220	-	-
Modified Quick	18778	20253	21789	20273	21746	20255



Random File Generator:

Code:

```
#this will ask the user for the input size and will create array of randomly
#generated numbers and will be used as a global variable by all other 4
#algorithms so that the data set is same and can be compared with each
#other.
import random,os
# #def return_random_list():
print("Enter your desired input size")
input_size=input()
numbers = random.sample(range(1, 100000), int(input_size))
print numbers
with open('global_var.py', 'w') as f:
    f.write("input_size=%s\n" % input_size)
    f.write("numbers=%s" % numbers)
```