We need to implement **k-th order selection** in three different versions, which are:
(i) version with pivotitem set at A[low],
(ii) version with pivotitem using median of medians,
(iii) randomized or probabilistic selection algorithm

We will be using base concept of quick sort algorithm which comes under divide and conquer algorithms.We select a pivot element in the array and then we will make partition around the pivot which we are selected.We can select pivot in different ways:
1-regular procedure with last element as pivot
2-using A[low] which is first element as pivot
3-using a random element as pivot
4-using median as pivot

version with pivotitem set at A[low]:
Code:

```
def partition(array, start, end):
    pivot = array[start]
    low = start + 1
    high = end

    while True:
        while low <= high and array[high] >= pivot:
            high = high - 1

        # Opposite process of the one above
        while low <= high and array[low] <= pivot:
            low = low + 1

        if low <= high:
            array[low], array[high] = array[high], array[low]
            # The loop continues
        else:
            # We exit out of the loop
            break

    array[start], array[high] = array[high], array[start]

    return high

# function to perform quicksort
def quick_sort(array, start, end):
    if start >= end:
        return
```

```
    p = partition(array, start, end)
    quick_sort(array, start, p-1)
    quick_sort(array, p+1, end)

array = [ 10, 7, 8, 9, 1, 5]
quick_sort(array, 0, len(array) - 1)

print(f'Sorted array: {array}')
```

With the help of this code, we can extract the sorted array but we have to find kth largest/smallest element which increases time complexity by this way since we have to use a[k] to access of O(n^2) time complexity
We can avoid that by stopping at kth order using

```
if (k > 0 and k <= r - l + 1):
        index = partition(arr, l, r)
        if (index - l == k - 1):
            return arr[index]
        if (index - l > k - 1):
            return kthSmallest(arr, l, index - 1, k)
        return kthSmallest(arr, index + 1, r, k - index + l - 1)
```

With this part of code we can be able to access kth sorted element with time complexity less than n^2 and depending on ‘k’

version with pivotitem using median of medians:
Code:

```
def kthSmallest(arr, l, r, k):
      if (k > 0 and k <= r - l + 1):
        n = r - l + 1
        median = []
        i = 0
        while (i < n // 5):
            median.append(findMedian(arr, l + i * 5, 5))
            i += 1
        if (i * 5 < n):
            median.append(findMedian(arr, l + i * 5, n % 5))
            i += 1
         if i == 1:
            medOfMed = median[i - 1]
        else:
```

```
        medOfMed = kthSmallest(median, 0,
                               i - 1, i // 2)
      pos = partition(arr, l, r, medOfMed)
      if (pos - l == k - 1):
        return arr[pos]
      if (pos - l > k - 1):
        return kthSmallest(arr, l, pos - 1, k)
      return kthSmallest(arr, pos + 1, r,
                         k - pos + l - 1)
    return 999999999999

def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp
def partition(arr, l, r, x):
    for i in range(l, r):
        if arr[i] == x:
            swap(arr, r, i)
            break

    x = arr[r]
    i = l
    for j in range(l, r):
        if (arr[j] <= x):
            swap(arr, i, j)
            i += 1
    swap(arr, i, r)
    return i
def findMedian(arr, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(arr[i])
    lis.sort()

    return lis[n // 2]
if __name__ == '__main__':

    arr = [12, 3, 5, 7, 4, 19, 26]
    n = len(arr)
    k = 3
    print("K'th smallest element is", kthSmallest(arr, 0, n - 1, k))
```

Main objective is to divide the code in a balanced way which can be done using medians,for finding medians we need to have distinct elements else there would be swap based errors
With the help of this algorithm we can achieve a worst case time complexity of O(n)
1-for finding median of 5 elements-O(1)
2-since we are reducing to size 5,there will be n/5 arrays - O(n)
Rest are done using recursive steps and add-ons to the code executed

randomized selection algorithm:

Code:

```
import random
def kthSmallest(arr, l, r, k):
    if (k > 0 and k <= r - l + 1):
         pos = randomPartition(arr, l, r)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return kthSmallest(arr, l, pos - 1, k)
         return kthSmallest(arr, pos + 1, r,
                             k - pos + l - 1)
   # If k is more than the number of elements in the array
    return 999999999999

def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp
def partition(arr, l, r):
    x = arr[r]
    i = l
    for j in range(l, r):
        if (arr[j] <= x):
            swap(arr, i, j)
            i += 1
    swap(arr, i, r)
    return i
def randomPartition(arr, l, r):
    n = r - l + 1
    pivot = int(random.random() * n)
    swap(arr, l + pivot, r)
    return partition(arr, l, r)

if __name__ == '__main__':

    arr = [12, 3, 5, 7, 4, 19, 26]
```

```
    n = len(arr)
    k = 3
    print("K'th smallest element is",
        kthSmallest(arr, 0, n - 1, k))
```

This code also have worst case time complexity similar to the quick sort algorithm which is O(n^2) Since in a case of randomized way it may pick the corner element every time.But in normal cases it will have the time complexity of O(n) as an average.This can be seen in [reference].

Analysis:

I have generated random data set and calculated the value of execution time using this code for variation(1):

```
t=0
while t<50:
 n = random.randint(4,50)
 arr = []
 for i in range(0,n):
   x = random.randint(1,100)
   arr.append(x)

 n1 = len(arr)
 k = 4
 print("K-th smallest element is ", end = "")
 print(kthSmallest(arr, 0, n1 - 1, k))
 print("size",n)
 et = time.time()
 elapsed_time = (et - st)*1000000
 print('Execution time:', elapsed_time, 'micro seconds')
 t = t + 1
```

The results for completely randomized data set of size and array numbers are:
45,55.3131103515625
7,127.31552124023438
30,161.64779663085938
21,194.549560546875
44,257.4920654296875
50,332.3554992675781
46,413.1793975830078
14,449.4190216064453
8,476.1219024658203
49,557.6610565185547

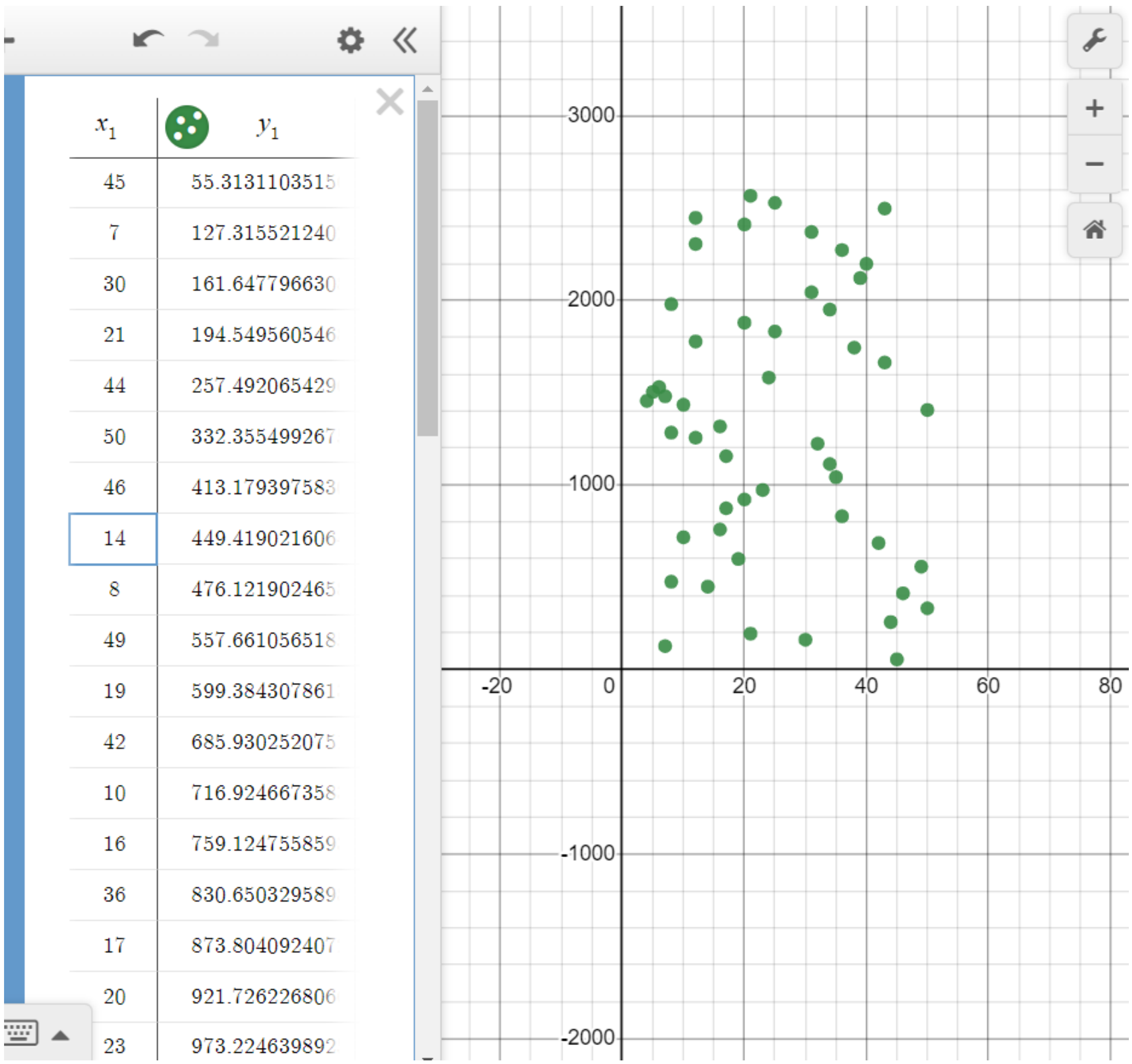
19,599.3843078613281
42,685.9302520751953
10,716.9246673583984
16,759.124755859375
36,830.6503295898438
17,873.8040924072266
20,921.7262268066406
23,973.2246398925781
35,1042.6044464111328
34,1113.6531829833984
17,1156.5685272216797
32,1224.0409851074219
12,1256.4659118652344
8,1283.884048461914
16,1317.739486694336
50,1406.6696166992188
10,1435.0414276123047
4,1456.0222625732422
7,1481.0562133789062
5,1504.8980712890625
6,1530.1704406738281
24,1581.9072723388672
43,1663.9232635498047
38,1744.2703247070312
12,1777.64892578125
25,1831.7699432373047
20,1879.6920776367188
34,1950.7408142089844
8,1980.0662994384766
31,2044.4393157958984
39,2121.2100982666016
40,2198.457717895508
36,2273.3211517333984
12,2305.9844970703125
31,2371.072769165039
20,2411.8423461914062
12,2447.6051330566406
43,2497.9114532470703
25,2529.144287109375
21,2567.0528411865234

Results:

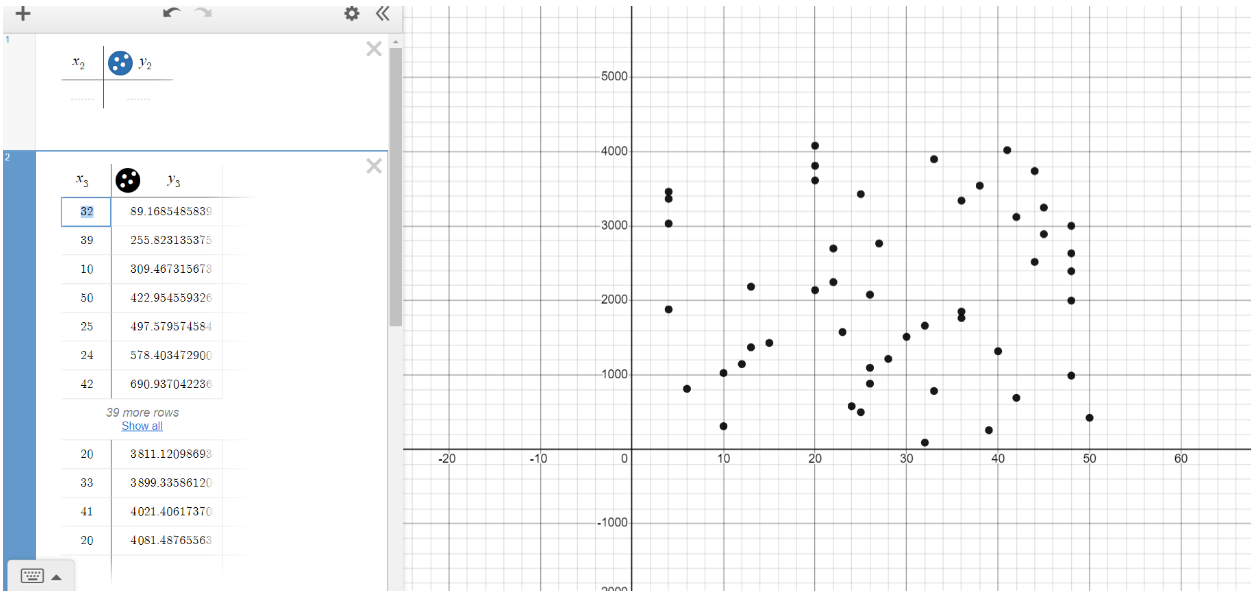
The graph got is completely randomized since we are changing both size and array elements.Along with that,the varying in graph to ideal estimated graph is due to two major reasons:

1.The kth element value which we are about to find is given as 4,so for the values of size of array near 4 or very far from 4 have similar time complexity since both cases are complex cases to each other.The time complexity value of those cases of n which are near/far from k are greater than medium range valued value of ‘n’

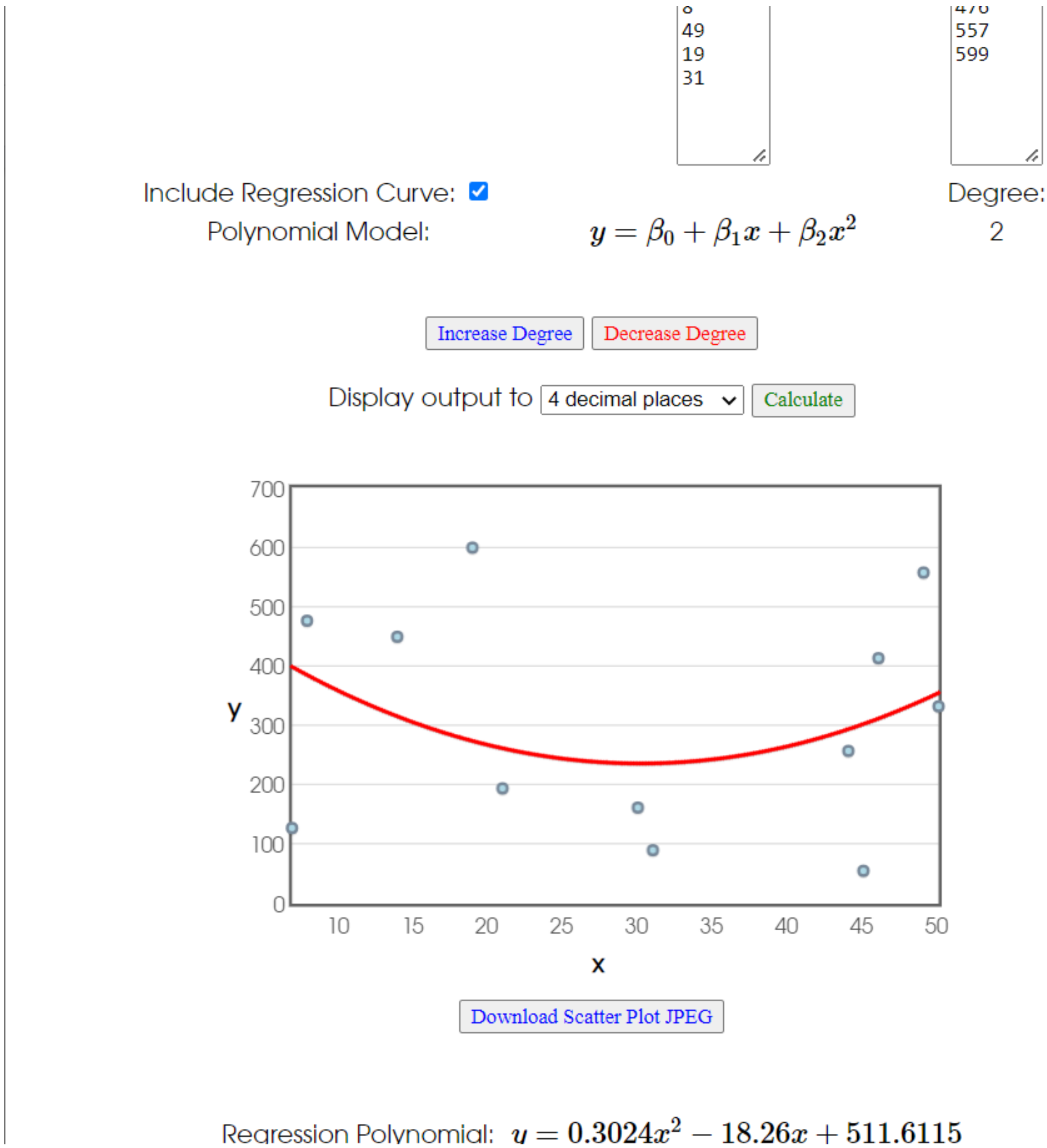
2.When the value of set are randomized that may result in same values which may cause distortion from normal cases.



Similarly for different data set,I have plotted graph values which came less random



I have used regression analysis for finding the estimated curve from the observations recorded
Which gave results as follows,



Hence we can deduce time complexity to O(n^2) for this case
For the randomized data set,the results are randomized to O(n^2)

Comparing all three algorithms:

I have compared values of time complexity between algorithm 2, 1 which are as follows:
7.62939453125,23.365020751953125
6.67572021484375,25.74920654296875
10.013580322265625,24.557113647460938
9.775161743164062,24.557113647460938

6.4373016357421875,22.411346435546875
2.6226043701171875,23.603439331054688
5.9604644775390625,21.696090698242188
5.0067901611328125,23.365020751953125
5.0067901611328125,22.172927856445312
7.152557373046875,27.179718017578125
7.152557373046875,22.649765014648438
7.3909759521484375,23.603439331054688
5.7220458984375,23.126602172851562
6.9141387939453125,23.126602172851562
12.39776611328125,24.080276489257812
5.4836273193359375,23.603439331054688
10.728836059570312,20.742416381835938
6.67572021484375,23.603439331054688
7.3909759521484375,23.84185791015625
4.5299530029296875,23.84185791015625
7.8678131103515625,23.603439331054688
5.245208740234375,22.88818359375
7.152557373046875,23.126602172851562
4.0531158447265625,21.696090698242188
2.6226043701171875,22.88818359375
5.7220458984375,12.39776611328125
9.5367431640625,22.88818359375
4.5299530029296875,21.93450927734375
9.059906005859375,23.603439331054688
5.245208740234375,21.93450927734375
8.58306884765625,34.09385681152344
5.7220458984375,22.411346435546875
9.059906005859375,23.126602172851562
11.444091796875,24.557113647460938
8.58306884765625,23.365020751953125
3.814697265625,23.126602172851562
6.9141387939453125,22.411346435546875
9.5367431640625,23.603439331054688
9.059906005859375,21.219253540039062
5.4836273193359375,23.365020751953125
4.291534423828125,21.696090698242188
2.6226043701171875,12.874603271484375
8.106231689453125,22.172927856445312
5.0067901611328125,23.365020751953125
5.0067901611328125,22.649765014648438
2.384185791015625,23.126602172851562
7.8678131103515625,22.172927856445312
7.152557373046875,22.88818359375

7.8678131103515625,21.696090698242188
2.384185791015625,22.88818359375

For a fixed values of n=8,we can clearly 0bserve that time complexity is 2 is lesser compared to 1

Comparison between algorithms 2,3:

39.81590270996094,6.9141387939453125
32.66334533691406,6.198883056640625
50.067901611328125,15.497207641601562
31.948089599609375,15.020370483398438
32.66334533691406,16.450881958007812
72.71766662597656,13.113021850585938
37.90855407714844,14.066696166992188
31.709671020507812,11.205673217773438
29.802322387695312,24.557113647460938
29.56390380859375,18.358230590820312
31.232833862304688,16.689300537109375
15.735626220703125,8.106231689453125
30.755996704101562,14.781951904296875
32.18650817871094,9.059906005859375
31.948089599609375,5.0067901611328125
30.755996704101562,9.298324584960938
30.040740966796875,14.30511474609375
30.517578125,9.059906005859375
33.37860107421875,19.550323486328125
30.994415283203125,15.497207641601562
32.4249267578125,7.8678131103515625
41.72325134277344,18.11981201171875
32.4249267578125,10.967254638671875
31.47125244140625,15.2587890625
50.067901611328125,17.1661376953125
32.18650817871094,10.967254638671875
30.994415283203125,7.8678131103515625
30.517578125,18.358230590820312
31.709671020507812,8.106231689453125
30.994415283203125,19.311904907226562
30.994415283203125,15.020370483398438
32.18650817871094,17.1661376953125
33.37860107421875,14.066696166992188
31.232833862304688,17.404556274414062
29.802322387695312,8.344650268554688
31.709671020507812,306.84471130371094

36.716461181640625,11.920928955078125
31.948089599609375,17.881393432617188
32.66334533691406,21.93450927734375
18.596649169921875,5.0067901611328125
31.948089599609375,11.205673217773438
30.994415283203125,16.927719116210938
34.09385681152344,5.245208740234375
32.66334533691406,12.39776611328125
32.66334533691406,14.066696166992188
39.10064697265625,16.927719116210938
34.332275390625,23.365020751953125
31.948089599609375,33.61701965332031
29.802322387695312,15.2587890625
27.418136596679688,9.5367431640625

The performances are similar and sometimes better in algorithm 3 when compared to 2 but there are cases where time complexity of 3 are very high than 2 which is similar to the corner cases in the randomized function hence the results are similar to expected

Comparing between 1,3:

22.88818359375,24.557113647460938
9.298324584960938,17.404556274414062
12.159347534179688,19.073486328125
10.251998901367188,4.5299530029296875
8.106231689453125,14.066696166992188
6.9141387939453125,16.21246337890625
6.67572021484375,4.5299530029296875
11.682510375976562,6.9141387939453125
16.450881958007812,16.927719116210938
11.444091796875,19.550323486328125
12.874603271484375,7.3909759521484375
9.5367431640625,5.245208740234375
3.814697265625,4.5299530029296875
7.152557373046875,9.5367431640625
11.444091796875,11.444091796875
6.9141387939453125,9.775161743164062
6.4373016357421875,11.682510375976562
9.5367431640625,16.689300537109375
7.3909759521484375,14.781951904296875
7.3909759521484375,7.62939453125
14.781951904296875,10.728836059570312
7.62939453125,9.5367431640625
9.5367431640625,17.404556274414062

8.106231689453125,19.073486328125
5.4836273193359375,12.636184692382812
7.3909759521484375,11.444091796875
5.9604644775390625,11.682510375976562
9.298324584960938,17.881393432617188
3.5762786865234375,16.927719116210938
3.5762786865234375,5.245208740234375
7.62939453125,10.013580322265625
10.49041748046875,13.589859008789062
8.344650268554688,10.728836059570312
14.543533325195312,12.874603271484375
6.198883056640625,8.821487426757812
13.828277587890625,18.358230590820312
15.020370483398438,10.728836059570312
3.5762786865234375,13.113021850585938
10.013580322265625,9.775161743164062
15.497207641601562,6.9141387939453125
13.113021850585938,15.497207641601562
9.059906005859375,18.358230590820312
11.444091796875,8.58306884765625
10.251998901367188,10.728836059570312
9.059906005859375,10.967254638671875
4.291534423828125,10.49041748046875
6.67572021484375,12.636184692382812
4.291534423828125,23.126602172851562
13.589859008789062,8.821487426757812
11.205673217773438,13.3514404296875

These are the results obtained when programmes are compared at randomized data set of 50 cases

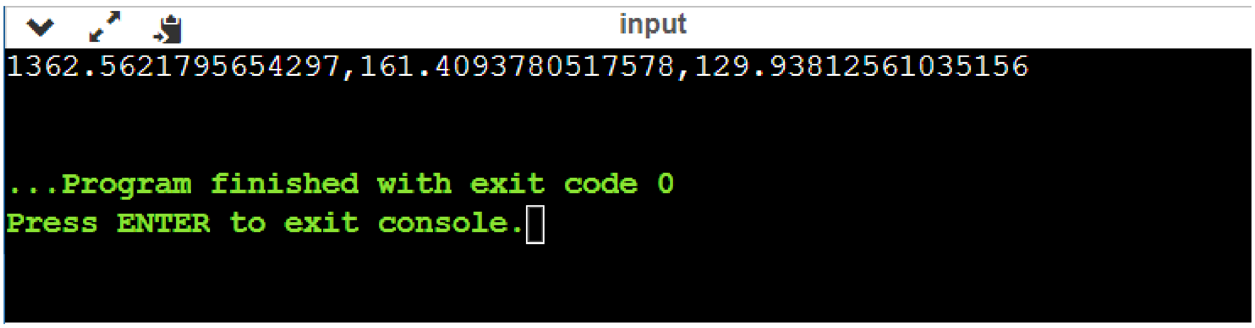
For same test case:

code:

```
n1 = 50
arr = numpy.arange(50)
k = 8
st1 = time.time()
kthSmallest1(arr, 0, n1 - 1, k)
et1 = time.time()
elapsed_time1 = (et1 - st1)*1000000
print(elapsed_time1, end=",")
st2 = time.time()
kthSmallest2(arr, 0, n1 - 1, k)
et2 = time.time()
elapsed_time2 = (et2 - st2)*1000000
```

```
print(elapsed_time2, end=",")
st3 = time.time()
kthSmallest3(arr, 0, n1 - 1, k)
et3 = time.time()
elapsed_time3 = (et3 - st3)*1000000
print(elapsed_time3)
```

The results are:



Conclusion:

This finally proves that
For algorithm 1,time complexity is of-O(n^2)
For algorithm 2,time complexity if of -O(n)
For algorithm 3,time complexity if of -O(n) but it may increase at corner cases