CMSC 351: kth Order

Justin Wyss-Gallifent

August 3, 2022

1 Introduction: ......................................................... 2
2 Naive Brute-Force Method ........................................ 2
  2.1 Approach .......................................................... 2
  2.2 Time Complexity .................................................. 2
3 Thoughts on Other Possibilities ................................... 3
  3.1 Sorting First ...................................................... 3
  3.2 Partially Sorting .................................................. 3
  3.3 Using a Min Heap .................................................. 3
4 QuickSelect - Decrease and Conquer ............................. 4
  4.1 Approach .......................................................... 4
  4.2 Pseudocode ...................................................... 5
  4.3 Time Complexity - The Beginning ............................. 6
  4.4 Pivot Value Choice - Median of Medians ..................... 7
  4.5 Return to Time Complexity .................................... 12
5 QuickSort-Related Note ............................................ 15
6 Thoughts, Problems, Ideas: ....................................... 16
7 Python Test and Output - Naive .................................. 17
8 Python Test and Output - Median of Medians .................. 18
1 Introduction:

Given an unsorted list of $n$ distinct numbers suppose we wish to find the $k$th smallest of those. Here both $n$ and $k$ can vary. How might we go about this? This question arises when finding the minimum, the maximum, and, what is valuable to us, the median.

Definition 1.0.1. The $k$th order element in a list is the $k$th smallest entry. So the 1st order entry is the minimum, for example.

2 Naïve Brute-Force Method

2.1 Approach

One brute-force method to finding the $k$th order entry might be to first find the smallest, then the next smallest, and so on, until we reach the $k$th smallest. This is not hard to implement, the pseudocode follows.

The general idea is that we find the minimum and assign it to $k$th. We then iterate by repeatedly: Go through $A$ and find the smallest which is less than the maximum but larger than $k$th. This would be the next smallest. We do this $k-1$ times since $k$th was already the 1st order statistic.

```python
def kthorder(A, k):
    n = len(A)
    mini = minimum element in A
    maxi = maximum element in A
    kth = mini
    for i = 1 to k-1
        nextsmallest = maxi
        for i = 0 to n-1:
            if (A[i] < nextsmallest) and (A[i] > kth)
                nextsmallest = A[i]
        kth = nextsmallest
    return (kth)
```

2.2 Time Complexity

Finding the maximum and minimum is $\Theta(n)$. The inner loop is $\Theta(n)$ and iterates $k-1$ times, and so the entire algorithm is $\Theta(n + (k-1)n) = \Theta(kn)$. This isn’t bad and for a specific pre-determined $k$ it’s just $\Theta(n)$ but we’d like something with a better time complexity for all possible $n$ and $k$. For example when finding the median $k$ depends upon $n$. Could we possibly do better?
3  Thoughts on Other Possibilities

There are a number of other approaches including:

3.1  Sorting First

We could simply sort the entire list and then select the \((k - 1)st\) entry (to adjust for the index). We can use something like MergeSort or HeapSort which has \(O(n \log n)\) time complexity. This is nice in the sense that our time complexity does not depend on \(k\).

3.2  Partially Sorting

We can do okay even with something like partial reverse BubbleSort. If we sort from the right then each pass through sorts the start of the list one more step. Since each pass through is \(\Theta(n)\) then this will take \(\Theta(kn)\), not really better than a brute force approach.

3.3  Using a Min Heap

Analogous to building a max heap we could build a min heap. When we built a max heap we said it was worse-case \(O(n \log n)\) time complexity but this is not asymptotically tight and in fact it is worse-case \(O(n)\) (we have not shown this) and the same is true for a min heap. Since extracting an element and fixing the heap takes worst-case \(O(\log n)\) time and we would do this \(k\) times the time complexity would be worse-case \(O(n + k \log n)\).

These are all pretty good and interesting but can we do better overall? Perhaps something that’s \(O(n)\) no matter what \(k\) is?
4 QuickSelect - Decrease and Conquer

4.1 Approach

It’s not at all clear how decrease-and-conquer might help here but it’s worth recalling the initial step of QuickSort. What we did was we chose a pivot value and then performed a partition on the list which resulted in the pivot value being re-located such that all smaller values were to the left and all larger values to the right. It follows that if the pivot value ends up at index \( p \) then the pivot value, now \( A[p] \), is the \((p + 1)\)st order entry.

If we want the \((p + 1)\)st order entry then we simply return \( A[p] \) and we’re done.

If we want order less than \( p + 1 \) we know that the value we are seeking is smaller and hence to the left of index \( p \) so we repeat the process (choose pivot value and partition) on \( A[0, .., p - 1] \) whereas if we want order greater than \( p + 1 \) we know that the value we are seeking is larger and hence to the right of index \( p \) so we repeat the process (choose pivot value and partition) on \( A[p + 1, .., n - 1] \).

What this then leads to is exactly a decrease-and-conquer approach which reduces the size of the list each time and results in the desired value being constrained in a smaller and smaller list.

Since the element of desired order certainly exists in the list if the target rank is not found before the list reaches length 1 then it must be found at that point.

The partitioning process used in quicksort is \( \Theta(n) \) which is great so the critical issue here is how much we can shorten our sublist at each stage. If we can shorten it quickly then the decrease-and-conquer approach may result in a time complexity better than \( \Theta(kn) \).

Shortening the sublist means making good pivot value choices at each stage. It’s not clear yet how we will do this but for now we can at least write down the pseudocode and fill in the pivot value choice details later.
4.2 Pseudocode

Here is the pseudocode. The `partition` function is stolen from the QuickSort pseudocode so we’ve omitted the details. Note that there is a critical line `pivotvalue = choosepivotvalue(l,r)` which needs details. At least for right now, don’t worry about how this is done, the critical thing is that after `partition` is called all the elements to the left of the pivot value are less than the pivot value and all the elements to the right of the pivot value are more than the pivot value.

```python
def selectkth(L,R,k):
    pivotvalue, index = choosepivotvalue(L,R)
    pivotindex = partition(L,R)
    if k - 1 < pivotindex
        return (selectkth(L, pivotindex - 1, k))
    else if k - 1 > pivotindex
        return (selectkth(pivotindex + 1, R, k))
    else
        return (A[pivotindex])
end

def partition(L,R):
    same as quicksort
end

def choosepivotvalue(L,R):
    somehow pick a value in A[L,...,R]
    return it and its index
end
```
4.3 Time Complexity - The Beginning

Suppose a call to selectkth takes time $T(n)$. This will then involve a call to choosepivotvalue, a call to partition, and a further call to selectkth. Observe:

- Let’s say the call to choosepivotvalue takes time $CPV(n)$.
- The call to partition takes time $\Theta(n)$.
- The new call to selectkth will be on a sublist of length $f(n)$ for some function of $n$ hence will take time $T(f(n))$.

Thus we have:

$$T(n) = CPV(n) + \Theta(n) + T(f(n))$$

If the pivot value selection is not engineered well in choosepivotvalue then the pivot value could end up at one end or the other during each recursive call to selectkth and the list would decrease in length by 1 each time. In this case we would have $f(n) = n - 1$ and then:

$$T(n) = CPV(n) + \Theta(n) + T(n - 1)$$

Even if $CPV(n) = 0$ we still end up with quadratic time complexity (you can prove this!) so this is no better than brute force.

To fix this we need to engineer choosepivotvalue well enough such that it reduces the sublist by something significant (not just 1) each time and such that it is fast enough that the first sum is small.
4.4 Pivot Value Choice - Median of Medians

So now we need to figure out how to write the \texttt{choosepivotvalue} function so that it shrinks the list length by a good amount. The method we’ll use is the Median of Medians.

The Median of Medians is a fast recursive method for finding a value close to the median. There are a few ways this can be implemented but we’ll take the standard approach of creating a method which results in mutual recursion.

Once we’ve established the median of medians function we will integrate it into the pseudocode loosely as follows:

```python
def selectkth(L,R,k):
    mom, index = mom(A[L:R])
    pivotindex = partition(L,R)
    if k-1 < pivotindex
        return selectkth(L, pivotindex -1, k))
    else if k-1 > pivotindex
        return selectkth(pivotindex+1, R,k))
    else
        return A[pivotindex])
end

def partition(L,R):
    same as quicksort
end

def mom(A):
    find mom
    return mom, index
end
```
Now on to how the Median of Medians method actually will work.

Broadly speaking the method works recursively as follows for a list containing \( n \) elements:

1. Divide the list into \( \lfloor n/5 \rfloor \) sublists of length 5 and perhaps one group with the remaining elements and sort those sublists.

2. Select the median of each sublist. For the final sublist if it has two or four elements select the lower median.

3. Apply \texttt{selectkth} recursively on the smaller list of those values in order to find the median (or lower median) of that new list.

Note that the call to \texttt{selectkth} inside this method itself will make calls to \texttt{partition} and on to \texttt{choosepivotvalue}, hence the mutual recursion. These successive calls will be on smaller lists, of course.

**Example 4.1.** Let’s demonstrate the median of median method on the following list:

10, 8, 5, 12, 11, 15, 21, 99, 7, 6, 70, 17, 3, 35, 71, 1, 2, 30, 36, 31, 32, 33, 60, 29, 28, 34, 40, 41, 80

First we divide this list into columns and sort each column:

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>3</th>
<th>1</th>
<th>28</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>17</td>
<td>2</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>35</td>
<td>30</td>
<td>32</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>21</td>
<td>70</td>
<td>31</td>
<td>33</td>
<td>80</td>
</tr>
<tr>
<td>12</td>
<td>99</td>
<td>71</td>
<td>36</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

The list formed by the medians (and lower median) of the columns is 10, 15, 35, 30, 32, 40 and calling \texttt{selectkth} specifically to get the (lower) median will return 30.

Of course this call to \texttt{selectkth} itself involves partitions and deeper calls to the median of median function.
Now that we see how the Median of Medians will work, here is the pseudocode again with the mutual recursion visible:

```python
def selectkth(L, R, k):
    mom, index = mom(A[L:R])
    pivotindex = partition(L, R)
    if k - 1 < pivotindex
        return (selectkth(L, pivotindex - 1, k))
    else if k - 1 > pivotindex
        return (selectkth(pivotindex + 1, R, k))
    else
        return (A[pivotindex])
end
end

def partition(L, R):
    same as quicksort
end

def mom(A):
    divide A into sublists and sort those
    mlist = new list of (lower?) medians of sublists
    s = apply selectkth to get the (lower?) median of mlist
    return s, index
end
```
**General Argument, Nice Case:**

We claim that the value obtained by the Median of Medians does a good job of subdividing the list via partition. To see this let’s figure out exactly what property it has.

To keep the calculation simple assume we have a list of $n$ elements where $n$ is an odd multiple of 5 in order to make the argument more straightforward. If this is not the case we need to tweak the approach slightly but the same basic outcome is guaranteed.

Consider this median of medians. Since it’s the median of a set of $n/5$ elements it’s greater than or equal to $\lceil (n/5)/2 \rceil = \lceil n/10 \rceil$ of them (because the median of $N$ elements is greater than or equal to $\lceil N/2 \rceil$ of them by definition). However those themselves are greater than or equal to three others each, since they’re each medians of 5. Thus our median of medians is greater than or equal to $3 \lceil n/10 \rceil \geq 3 \frac{3}{10} n$ elements in the original list.

In a symmetric fashion we can see that our median of medians is less than or equal to $\frac{3}{10} n$ elements in the original list.

Thus if we choose either of the sublists consisting of elements less than the MOM or the elements greater than the MOM and denote its length by $f(n)$ then we know:

$$\frac{3}{10} n \leq f(n) \leq \frac{7}{10} n$$
Example 4.2. Consider this list of 35 elements. Each column has been ordered and the median of medians has been shaded:

\[
\begin{array}{cccccc}
5 & 6 & 3 & 9 & 28 & 150 \\
8 & 7 & 17 & 13 & 29 & 251 \\
10 & 15 & 35 & 16 & 32 & 3057 \\
11 & 21 & 70 & 22 & 33 & 3158 \\
12 & 99 & 71 & 37 & 60 & 3673 \\
\end{array}
\]

Now let’s shade the medians of the other columns which are less than or equal to 30 and within those columns let’s shade the values which are less than or equal to their respective medians:

\[
\begin{array}{cccccc}
5 & 6 & 3 & 9 & 28 & 150 \\
8 & 7 & 17 & 13 & 29 & 251 \\
10 & 15 & 35 & 16 & 32 & 3057 \\
11 & 21 & 70 & 22 & 33 & 3158 \\
12 & 99 & 71 & 37 & 60 & 3673 \\
\end{array}
\]

We can see that our median of medians, 30, is guaranteed to be greater than or equal to 3 entries in each of \(4 = \left\lceil \frac{\text{Columns}}{2} \right\rceil = \left\lceil \frac{n/5}{2} \right\rceil \) columns. So it is greater than or equal to 3 \([n/10]\) entries in the entire list.

Similarly here is the picture showing the values which are guaranteed to be greater than or equal to our median of medians:

\[
\begin{array}{cccccc}
5 & 6 & 3 & 9 & 28 & 150 \\
8 & 7 & 17 & 13 & 29 & 251 \\
10 & 15 & 35 & 16 & 32 & 3057 \\
11 & 21 & 70 & 22 & 33 & 3158 \\
12 & 99 & 71 & 37 & 60 & 3673 \\
\end{array}
\]

If the number of elements is not an odd multiple of 5 then we need to tweak the argument slightly because it may not be true for smaller \(n\) but turns out to be true for large enough \(n\), which is sufficient for time complexity arguments. I have opted out of fiddling with the details because I think keeping it simple helps understand what’s going on.

We’ll explore some of these quirks in the exercises.
4.5 Return to Time Complexity

Recalling that we designated \( f(n) \) to be the function returning the size of the sublist after `partition` is called, we now know that for a list of length \( n \) if we choose the pivot value according to the median of medians approach we have:

\[
\frac{3}{10}n \leq f(n) \leq \frac{7}{10}n
\]

So now suppose that a call to `selectkth` takes time \( T(n) \). This call will then involve a call to `mom`, a call to `partition`, and a call to `selectkth`. Observe:

- The call to `mom` involves the division into sublists, the small sorts, the new list construction (length \( n/5 \)) and a further call to `selectkth`. For a list of length \( n \) this takes time \( \Theta(n) + T(0.2n) \)
- The call to `partition` takes time \( \Theta(n) \).
- The new call to `selectkth` will be on a sublist of length at most \( \frac{7}{10}n \) hence will take time \( T(0.7n) \).

Thus we now have a worst-case recurrence inequality involving \( T(n) \):

\[
T(n) \leq \Theta(n) + T(0.2n) + \Theta(n) + T(0.7n)
\]

Or simply:

\[
T(n) \leq T(0.7n) + T(0.2n) + \Theta(n)
\]

We’d like to use this to figure out the time complexity of \( T(n) \). Because of the inequality we’ll only be able to get a \( O \) time complexity on the worst-case. Note that it is certainly possible to use the lower bound of \( \frac{3}{10}n \) to obtain a \( \Omega \) time complexity on the worst-case but we won’t do that here.
Note: What follows needs some cleaning up. I can’t vouch for its accuracy right now.

We will now prove by strong induction that \( T(n) = \Theta(n) \).

First look at the \( \Theta(n) \) term. This term is also \( O(n) \) and so whatever it is we know there is some \( C_0 \) and some \( n_0 \) such that this is less than or equal to \( C_0 n \) for \( n \geq n_0 \).

We now assign:

\[
C_1 = \max \left\{ \frac{T(n_0)}{n_0}, 10C_0 \right\}
\]

\[
n_1 = 5n_0
\]

We claim that \( T(n) \leq C_1 n \) for all \( n \geq n_1 \), thereby proving that \( T(n) = \Theta(n) \).

For the base case observe that:

\[
T(n_1) = T(5n_0) = \frac{T(n_0)}{n_0} n_0 \leq C_1 n_0 = C_1 n_1
\]

For the inductive step we assume that \( T(n') \leq C_1 n' \) for all \( n' \) with \( n_1 \leq n' < n \) and we’ll show that \( T(n) \leq C_1 n \).

First observe that \( 0.7n < n \) and so \( T(0.7n) = C_1 (0.7n) \) and \( 0.2n < n \) and so \( T(0.2n) < C_1 (0.2n) \), both by the inductive hypothesis.

From here we get:

\[
T(n) \leq T(0.7n) + T(0.2n) + \Theta(n)
\]

\[
\leq 0.7C_1 n + 0.2C_1 n + 0.1C_1 n
\]

\[
\leq C_1 n
\]

Then we are done.

Note: This is a bit of a lie. We also need to know that \( 0.2n > n_1 \) and \( 0.7n > n_1 \) in order to invoke the inductive hypothesis.

The first of these requires \( n > 5n_1 \) and the second requires \( n > \frac{10}{7} n_1 \).

What this actually means is that we need to do base cases all the way from \( n_0 \) to \( 5n_0 \).

But how can we use this to find the time complexity for \( T(n) \)? We can’t use the Master Theorem but we can use constructive induction.

Suppose our goal is to show that \( T(n) = \Theta(n) \), a definite improvement over \( \Theta(n^2) \). This means that our goal is to find \( C \) such that \( T(n) \leq Cn \) for all sufficiently large \( n \).

We know that the recurrence relation holds for \( n \geq n_0 \) so for what follows let’s assume \( n \geq n_0 \) and set the base case at \( n_0 \). Since \( n_0 \) is some fixed value we can be sure that \( T(n_0) \) is bound by some constant, say \( T(n_0) \leq C_2 \). Since \( T(n_0) \leq C_2 = (C_2/n_0)n_0 \) our constant \( C \) must be at least \( C_2/n_0 \) to ensure that \( T(n) \leq Cn \) for our base case \( n = n_0 \).
For the inductive step assume \( n > n_0 \) and assume we have found a \( C \geq C_2/n_0 \) such that \( T(n') \leq Cn' \) for all \( n_0 \leq n' < n \) and consider \( T(n) \):

\[
T(n) \leq T(0.7n) + T(0.2n) + C_1 n
\leq C (0.7n) + C (0.2n) + C_1 n
\leq n (0.9C + C_1)
\]

In order to guarantee that this is less than or equal to \( Cn \) we need:

\[
n(0.9C + C_1) \leq Cn
0.9C + C_1 \leq C
-0.1C \leq -C_1
C \geq 10C_1
\]

To conclude we set \( C = \max\{10C_1, C_2/n_0\} \) and we are done. If we use choose our pivot value this way and use our decrease-and-conquer algorithm we can choose the \( k \)th order statistic with worst-case time \( \mathcal{O}(n) \).
5 QuickSort-Related Note

Recall that the time complexity of QuickSort was challenging because the best-case is achieved by somehow choosing the pivot value such that the two sublists are of equal size. However this involves choosing the median at each step and this is a challenging addition to the code. In response the typical approach is to select the pivot value randomly which gives rise to $O(n \log n)$ average-case time complexity.

Another choice is to use our new median-of-medians approach to select the pivot value. In this case each time the list gets split we can guarantee that one side is at least $3/10$ the size of the original list and the other side is at most $7/10$ the size of the original list.

The worst-case then arises when we get exactly the $30/70$ split each time. The resulting recurrence relation is then:

$$T(n) = T\left(\frac{3}{10}n\right) + T\left(\frac{7}{10}n\right) + \Theta(n)$$

This cannot be solved by the Master Theorem but the Akra-Bazzi method yields a solution of $\Theta(n \log n)$.

As theoretically interesting as this is, for what it’s worth the time overhead hidden in the repeated application of the median-of-medians approach makes this approach not useful in practice.
6 Thoughts, Problems, Ideas:

1. If we used a max heap instead of a min heap to find the $k$th order statistic and assuming we could build the max heap in $O(n)$ time, what would the $O$ time complexity of extracting the $k$th smallest element be?

2. Assume $k$ is given. Modify BubbleSort so that after $i$ iterations the first $i$ elements are correctly positioned and so that it stops when exactly the first $k$ elements are correctly positioned.

3. Given the list of 45 distinct elements:

   3, 43, 29, 41, 32, 59, 85, 7, 60, 4, 31, 11, 20, 8, 80,
   66, 22, 50, 16, 90, 26, 79, 2, 96, 6, 54, 81, 93, 53, 99,
   61, 36, 62, 14, 40, 17, 30, 95, 26, 79, 2, 96, 6, 54, 81, 93, 53, 99,

   (a) Separate the list into columns containing five elements each and sort each column.

   (b) Identify the median of medians.

   (c) Draw a table identifying the values guaranteed to be less than or equal to the MOM.

   (d) Draw a table identifying the values guaranteed to be greater than or equal to the MOM.

4. In determining the median of medians of a list containing $n$ distinct elements suppose $n$ is an even multiple of 5. Consequently the median of medians will not actually be an element in the list so instead we choose the “lower median” which is the largest value below the median. Show that the MOM is still greater than or equal to $\frac{3}{10} n$ elements in the original list but the less than or equal to bound is slightly better.

5. In determining the median of medians of a list containing $n$ distinct elements suppose $n$ is an even multiple of 5 plus 1. We group the elements into an odd number of groups of five elements and a single group of one element and proceed as before. Show that the MOM is still greater than or equal to $\frac{3}{10} n$ elements.

6. In the “General Argument, Nice Case” for the median of medians we assumed $n$ to be an odd multiple of 5 and obtained $\frac{3}{10} n \leq f(n) \leq \frac{7}{10} n$. Calculate the bounds by modifying the argument for the case in we choose an odd multiple of 7 instead.
7 Python Test and Output - Naïve

Code:

```python
import random

def kthorder(A,k):
    n = len(A)
    mini = A[0]
    maxi = A[0]
    for i in range(0,n):
        if A[i] < mini:
            mini = A[i]
        if A[i] > maxi:
            maxi = A[i]
    kth = mini
    print('Minimum: ' + str(kth))
    for i in range(1,k):
        nextsmallest = maxi
        for i in range(0,n):
            if (A[i] < nextsmallest) and (A[i] > kth):
                nextsmallest = A[i]
        kth = nextsmallest
        print('Next Smallest: ' + str(kth))
    return (kth)

A = []
while len(A) < 20:
    r = random.randint(0,100)
    if r not in A:
        A.append(r)

print(A)

print('Final Answer: ' + str(kthorder(A,5)))
```

Output:

```
[67, 84, 27, 56, 82, 57, 15, 13, 66, 41, 30, 97, 52, 99, 87, 47, 45, 20, 53, 64]
Minimum: 13
Next Smallest: 15
Next Smallest: 20
Next Smallest: 27
Next Smallest: 30
Final Answer: 30
```
8 Python Test and Output - Median of Medians

Code Comment:
The process of repeatedly partitioning a given list alters that list and this is reflected in the fact that the functions take the array argument by reference (the default in Python).

Observe however that the Median of Median function creates a new list (the list of medians of groups of mostly five) on which the process is recursively called. When creating that new list, \texttt{mom} needs to alter the working list by sorting in groups, hence it makes a copy using \texttt{AA = A[:]}, and then works on \texttt{AA}.

Consequently it may be helpful to keep in mind that there are several arrays being managed, each of them by reference.

Code Part 1:

```python
import random
import math

def selectkth(A,l,r,kwish):
    # Find the pseudomedian.
    pmed = mom(A[l:r+1])
    # Find the index of the pseudomedian
    pmedi = 0
    while A[pmedi] != pmed:
        pmedi = pmedi + 1
    # Swap that entry with the final entry.
    # Partition on the final entry.
    pivotindex = partition(A,l,r)
    if kwish < pivotindex +1:
        return ( selectkth (A,l, pivotindex -1 , kwish ))
    elif kwish > pivotindex +1:
        return ( selectkth (A, pivotindex +1 ,r, kwish ))
    else :
        return (A[pivotindex])

def partition(A,l,r):
    pivot = A[r]
    t = 1
    for i in range(l,r):
        if A[i] <= pivot:
            t = t + 1
    temp = A[t]
    A[r] = temp
```

18
return t
def mom(A):
    # Make a copy because we're going to mess it up doing our grouped sorting.
    AA = A[:]
    n = len(AA)
    mlist = []
    for i in range(0,int(math.ceil(float(n)/5))):
        Li = 5*i
        Ri = Li + 5
        if Ri > n-1:
            Ri = n-1
        AA[Li:Ri] = sorted(AA[Li:Ri])
        mlist.append(AA[Li+(Ri-Li-1)//2])
    if len(mlist)==1:
        return mlist[0]
    s = selectkth(mlist,0,len(mlist)-1,len(mlist)+1//2)
    return (s)

LIST = []
while len(LIST) < 20:
    r = random.randint(0,100)
    if r not in LIST:
        LIST.append(r)
print('Array: '+str(LIST))
kwish = random.randint(1,len(LIST))
kth = selectkth(LIST,0,len(LIST)-1,kwish)
print("I'm looking for rank: " + str(kwish))
print('It is: ' + str(kth))
print('Here is the Python sorted array for checking: ')
LIST.sort()
print(LIST)
if LIST[kwish-1] == kth:
    print('Success!')
Output:

Array: [97, 22, 52, 92, 34, 54, 2, 30, 3, 43, 13, 86, 28, 16, 33, 51, 1, 0, 32, 41]
I'm looking for rank: 12
It is: 34
Here is the Python sorted array for checking:
[0, 1, 2, 3, 13, 16, 22, 28, 30, 32, 33, 34, 41, 43, 51, 52, 54, 86, 92, 97]
Success!

Output:

Array: [58, 77, 27, 41, 92, 19, 44, 7, 65, 31, 85, 84, 57, 94, 81, 71, 20, 82, 55, 5]
I'm looking for rank: 3
It is: 19
Here is the Python sorted array for checking:
[5, 7, 19, 20, 27, 31, 41, 44, 55, 57, 58, 65, 71, 77, 81, 82, 84, 85, 92, 94]
Success!

Output:

Array: [48, 11, 63, 59, 21, 41, 72, 43, 61, 74, 34, 9, 47, 100, 73, 38, 28, 87, 42, 89]
I'm looking for rank: 18
It is: 87
Here is the Python sorted array for checking:
[9, 11, 21, 28, 34, 38, 41, 42, 43, 47, 48, 59, 61, 63, 72, 73, 74, 87, 89, 100]
Success!