12

Selection Algorithms

One interesting set of algorithms related to finding elements in an unordered list of items is selection algorithms. Given a list of elements, selection algorithms are used to find the k^{th} smallest or largest element from the list. So given a list of data elements and a number (k), the aim is to find the k^{th} smallest or largest element. The simplest case of selection algorithms is to find the minimum or maximum data element from the list. However, sometimes, we may need to find the k^{th} smallest or largest element in the list. The simplest way is to first sort the list using any sorting algorithm, and then we can easily obtain the k^{th} smallest (or largest) element. However, when the list is very large, then it is not efficient to sort the list to get the k^{th} smallest or largest element. In that case, we can use different selection algorithms that can efficiently produce the k^{th} smallest or largest element.

In this chapter, we will cover the following topics:

- Selection by sorting
- Randomized selection
- Deterministic selection

We will start with the technical requirements, and then we will discuss selection by sorting.

Technical requirements

All of the source code that's used in this chapter is provided in the given GitHub link: `https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Third-Edition/tree/main/Chapter12`.

Selection by sorting

Items in a list may undergo statistical inquiries such as finding the mean, median, and mode values. Finding the mean and mode values does not require the list to be ordered. However, to find the median in a list of numbers, the list must first be ordered. Finding the median requires you to find the element in the middle position of the ordered list. In addition, this can be used when we want to find the k^{th} smallest item in the list. To find the k^{th} smallest number in an unordered list of items, an obvious method is to first sort the list, and after sorting, you can rest assured that the element at index 0 will hold the smallest element in the list. Likewise, the last element in the list will hold the largest element in the list.

For more information on how to order data items within a list, see *Chapter 11*, *Sorting*. However, in order to obtain a k^{th} smallest element from the list, it is not a good solution to apply a sorting algorithm to a long list of elements to obtain the minimum or maximum or k^{th} smallest or largest value from the list since sorting is quite an expensive operation. Thus, if we need to find out the k^{th} smallest or largest element from a given list, there is no need to sort the complete list as we have other methods that we can use for this purpose. Let's discuss better techniques to find the k^{th} smallest element without having to sort the list in the first place, starting with randomized selection.

Randomized selection

The randomized selection algorithm is used to obtain the k^{th} smallest number that is based on the quicksort algorithm; the randomized selection algorithm is also known as quickselect. In *Chapter 11*, *Sorting*, we discussed the quicksort algorithm. The quicksort algorithm is an efficient algorithm to sort an unordered list of items. To summarize, the quicksort algorithm works as follows:

1. It selects a pivot.
2. It partitions the unsorted list around the pivot.
3. It recursively sorts the two halves of the partitioned list using *steps 1* and *2*.

One important fact about quicksort is that after every partitioning step, the index of the pivot does not change, even after the list becomes sorted. This means that after each iteration, the selected pivot value will be placed in its correct position in the list. This property of quicksort enables us to obtain the k^{th} smallest number without sorting the complete list. Let's discuss the randomized selection method, which is also known as the quickselect algorithm, to obtain the k^{th} smallest element from a list of n data items.

Quickselect

The quickselect algorithm is used to obtain the k^{th} smallest element in an unordered list of items. It is based on the quicksort algorithm, in which we recursively sort the elements of both the sublists from the pivot point. In each iteration, the pivot value reaches the correct position in the list, which divides the list into two unordered sublists (left and right sublists), where the left sublist has smaller values as compared to the pivot value, and the right sublist has greater values compared to the pivot value. Now, in the case of the quickselect algorithm, we recursively call the function only for the sublist that has the k^{th} smallest element.

In the quickselect algorithm, we compare the index of the pivot point with the `k` value to obtain the k^{th} smallest element from the given unordered list. There will be three cases in the quickselect algorithm, as follows:

1. If the index of the pivot point is smaller than `k`, then we are sure that the k^{th} smallest value will be present on the right-hand sublist of the pivot point. So we only recursively call the quickselect function for the right sublist.
2. If the index of the pivot point is greater than `k`, then it is obvious that the k^{th} smallest element will be present on the left-hand side of the pivot point. So we only recursively look for the i^{th} element in the left sublist.
3. If the index of the pivot point is equal to `k`, then it means that we have found out the k^{th} smallest value, and we return it.

Let's understand the working of the quickselect algorithm with an example. Consider a list of elements, `{45, 23, 87, 12, 72, 4, 54, 32, 52}`. We can use the quickselect algorithm to find the third smallest element in this list.

We start the algorithm by selecting a pivot value, that is, `45`. Here we are choosing the first element as the pivot element for simplicity; however, any other element can be chosen as a pivot element. After the first iteration of the algorithm, the pivot value moves to its correct position in the list, which in this example is at index `4` (the index is starting from `0`). Next, we check the condition `k<pivot` point (that is, `2<4`). Case- `2` is applicable, so we only consider the left sublist, and recursively call the function. Here, we compare the index of the pivot value (that is, `4`) with the value of `k` (that is, the 3^{rd} position or at index `2`).

Next, we take the left sublist and select the pivot point (that is, `4`). After the run, the `4` is placed in its correct position (that is, the 0^{th} index). As the index of the pivot is less than the value of `k`, we consider the right sublist.

Similarly, we take 23 as the pivot point, which is also placed in its correct position. Now, when we compare the index of the pivot point and the value of k, they are equal, which means we have found the 3rd smallest element, and it will be returned. The complete step-by-step process to find the 3rd smallest element is shown in *Figure 12.1*:

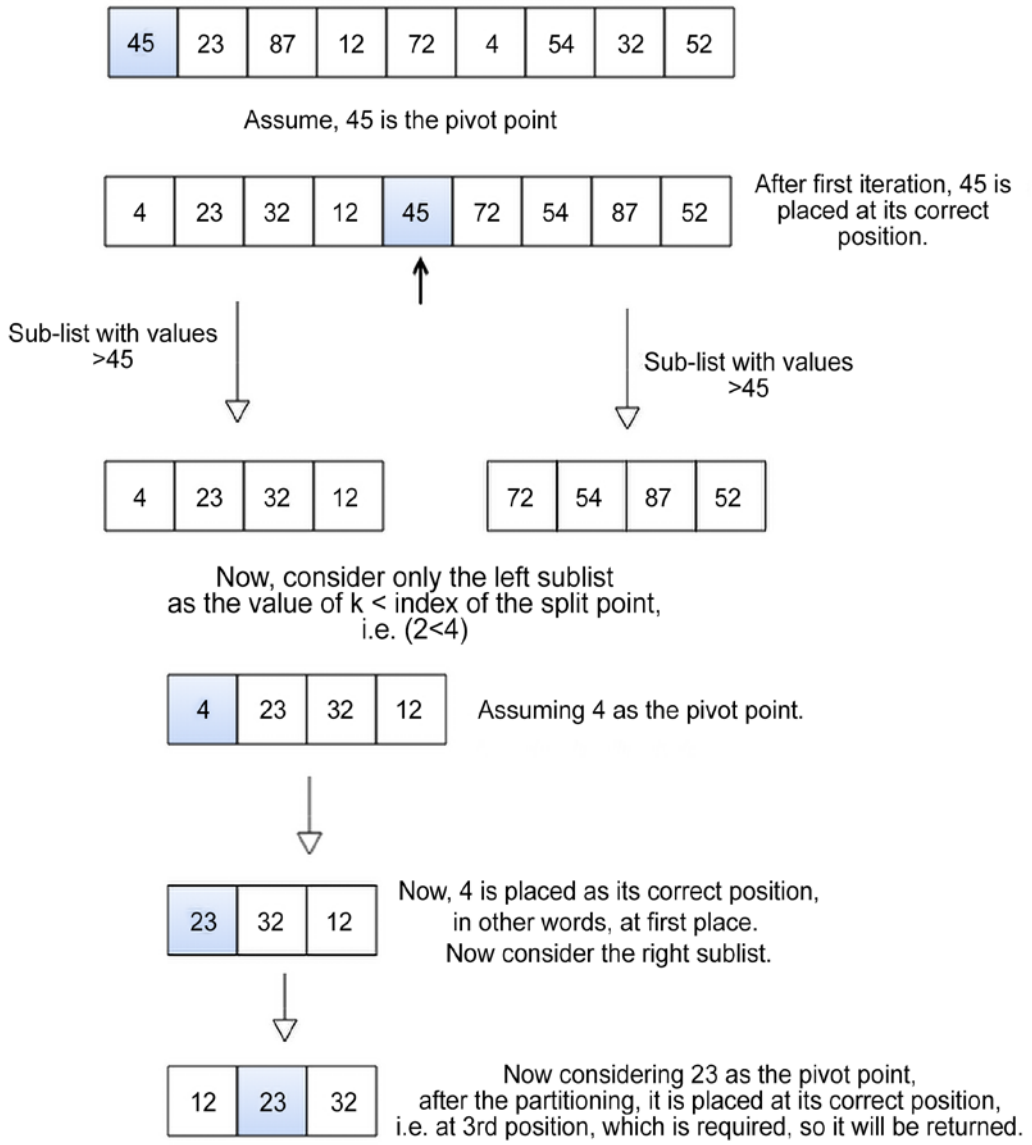

Figure 12.1: Step-by-step demonstration of the quickselect algorithm

Let's discuss the implementation of the `quick_select` method. It is defined as follows:

```
def quick_select(array_list, start, end, k):
    split = partition(array_list, start, end)
    if split == k:
        return array_list[split]
    elif split < k:
        return quick_select(array_list, split + 1, end, k)
    else:
        return quick_select(array_list, start, split-1, k)
```

In the above code, the `quick_select` function takes the complete array, the index of the first element of the list, the index of the last element, and the k^{th} element specified by value `k` as parameters. The value of `k` maps with the index that the user is searching for, meaning the k^{th} smallest number in the list.

Initially, we use the `partition()` method (which is defined and discussed in detail in *Chapter 11*, *Sorting*) to place the selected pivot point in such a way that it divides the given list of elements in the left sublist and the right sublist, in which the left sublist has data elements that are smaller than the pivot value, and right subtree has data elements that are greater than the pivot value. The `partition()` method is called `split = partition(array_list, start, end)` and returns the `split` index. Here, the `split` index is the position where the pivot element is placed in the array, and (`start`, `end`) is the starting and ending indices of the list. Once we get the split point, we compare the `split` index with the required value of `k` to find out whether we have reached the position of the k^{th} smallest data item or whether the required k^{th} smallest element will be on the left sublist or the right sublist. These three conditions are as follows:

1. If the `split` is equal to the value of `k`, then it means that we have reached the k^{th} smallest data item in the list.
2. If the `split` is less than `k`, then it means that the k^{th} smallest item should exist or be found between `split+1` and `right`.
3. If the `split` is greater than `k`, then it means that the k^{th} smallest item should exist or be found between `left` and `split-1`.

In the preceding example, a split point occurs at index 4 (index starting from 0). If we are searching for the 3^{rd} smallest number, then since `4 < 2` yields `false`, a recursive call to the right sublist is made using `quick_select(array_list, left, split-1, k)`.

Here, for the completeness of this algorithm, the `partition()` method is given as follows:

```
def partition(unsorted_array, first_index, last_index):
    pivot = unsorted_array[first_index]
    pivot_index = first_index
    index_of_last_element = last_index
    less_than_pivot_index = index_of_last_element
    greater_than_pivot_index = first_index + 1
    while True:
        while unsorted_array[greater_than_pivot_index] < pivot and
greater_than_pivot_index < last_index:
            greater_than_pivot_index += 1
        while unsorted_array[less_than_pivot_index] > pivot and less_than_
pivot_index >= first_index:
            less_than_pivot_index -= 1
        if greater_than_pivot_index < less_than_pivot_index:
            temp = unsorted_array[greater_than_pivot_index]
            unsorted_array[greater_than_pivot_index] = unsorted_
array[less_than_pivot_index]
            unsorted_array[less_than_pivot_index] = temp
        else:
            break
    unsorted_array[pivot_index] = unsorted_array[less_than_pivot_index]
    unsorted_array[less_than_pivot_index] = pivot
    return less_than_pivot_index
```

We can use the below code snippet to find out the k^{th} smallest element using the `quickselect` algorithm for a given array.

```
list1 = [3,1,10, 4, 6, 5]
print("The 2nd smallest element is", quick_select(list1, 0, 5, 1))
print("The 3rd smallest element is", quick_select(list1, 0, 5, 2))
```

The output of the above code is as follows:

```
The 2nd smallest element is 3
The 3rd smallest element is 4
```

In the above code, we get the 2^{nd} and 3^{rd} smallest elements from the given list of elements. The worst-case performance of a randomized selection-based `quick-select` algorithm is $O(n^2)$.

In the above implementation of the `partition()` method, we use the first element of the list as the pivot element for simplicity, but any element can be chosen from the list as the pivot element. A good pivot element is one that divides the list into almost equal halves. Therefore, it is possible to improve the performance of the quickselect algorithm by selecting the split point more efficiently in linear time with the worst-case complexity of `O(n)`. We discuss how to do this in the next section using deterministic selection.

Deterministic selection

Deterministic selection is an algorithm for finding out the k^{th} item in an unordered list of elements. As we have seen in the `quickselect` algorithm, we select a random "pivot" element that partitions the list into two sublists and calls itself recursively for one of the two sublists. In a deterministic selection algorithm, we choose a pivot element more efficiently instead of taking any random pivot element.

The main concept of the deterministic algorithm is to select a pivot element that produces a good split of the list, and a good split is one that divides the list into two halves. For instance, a good way to select a pivot element would be to choose the median of all the values. But we will need to sort the elements in order to find out the median element, which is not efficient, so instead, we try to find a way to select a pivot element that divides the list roughly in the middle.

The median of medians is a method that provides us with the approximate median value, that is, a value close to the actual median for a given unsorted list of elements. It divides the given list of elements in such a way that in the worst case, at least 3 out of 10 (3/10) of the list will be below the pivot element, and at least 3 out of 10 of the elements will be above the list.

Let's take an example to understand this. Let's say we have a list of 15 elements: `{11, 13, 12, 111, 110, 15, 14, 16, 113, 112, 19, 18, 17, 114, 115}`.

Next, we divide it into groups of 5 elements and sort them as follows: `{{11, 12, 13, 110, 111}, {14, 15, 16, 112, 113}, {17, 18, 19, 114, 115}}`.

Next, we compute the median of each of these groups, and they are 13, 16, and 19, respectively. Further, the median of these median values `{13, 16, 19}` is 16. This is the median of medians for the given list. Here, we can see that 5 elements are smaller, and 9 elements are greater than the pivot element. When we select this median of the median as a pivot element, the list of `n` elements is divided in such a way that at least `3n/10` elements are smaller than the pivot element.

The deterministic algorithm to select the k^{th} smallest element works as follows:

1. Split the list of unordered items into groups of five elements each (the number 5 is not mandatory; it can be changed to any other number, for example, 8)
2. Sort these groups (in general, we use insertion sort for this purpose) and find the median of all these groups
3. Recursively, find the median of the medians obtained from these groups; let's say that is point **p**
4. Using this point **p** as the pivot element, recursively call the partition algorithm similar to quickselect to find out the k^{th} smallest element

Let's consider an example list of 15 elements to understand the working of the deterministic algorithm to find out the 3^{rd} smallest element from the list, as shown in *Figure 12.2*. First, we divide the list into groups of 5 elements each, and then we sort these groups/sublists. Once we have sorted the lists, we find out the median of the sublists. For this example, items **23**, **52**, and **34** are the medians of these three sublists, as shown in *Figure 12.2*.

Next, we sort the list of medians for all the sublists. Further, we find out the median of this list, that is, the median of the median, which is **34**. This median of medians is used to select the partition/pivot point for the whole list. Further, we divide the given list using this pivot element to partition the list into 2 sublists, placing the given pivot element at its correct position in the list. For this example, the index of the pivot element is 7 (index starting from 0; this is shown in *Figure 12.2.*).

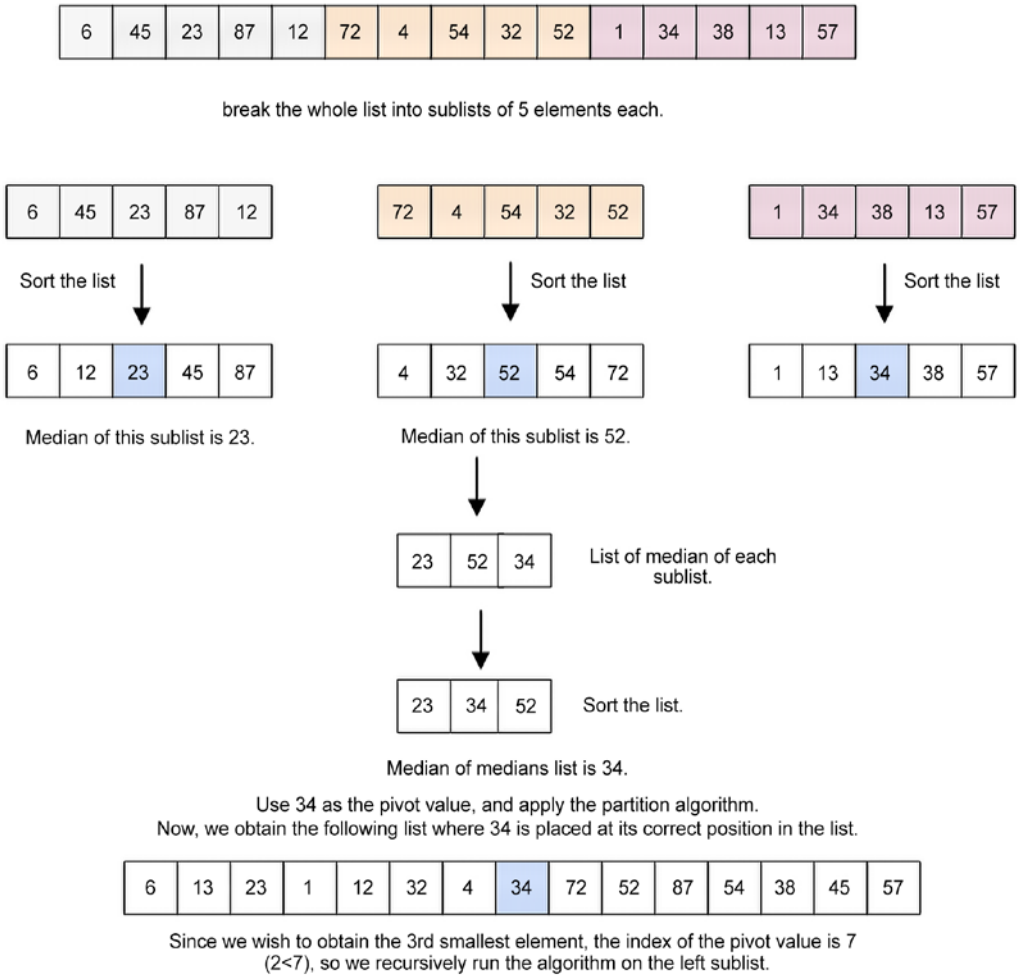

Figure 12.2: Step-by-step procedure for the deterministic selection algorithm

The index of the pivot element is greater than the k^{th} value, and hence, we recursively call the algorithm on the left sublist to obtain the required k^{th} smallest element.

Next, we will discuss the implementation of the deterministic selection algorithm.

Implementation of the deterministic selection algorithm

To implement the deterministic algorithm for determining the k^{th} smallest value from the list, we start implementing the updated `partition()` method, which divides the list where we select the pivot element using the median of medians method. Let's now understand the code for the `partition` function:

```
def partition(unsorted_array, first_index, last_index):
    if first_index == last_index:
        return first_index
    else:
        nearest_median = median_of_medians(unsorted_array[first_
index:last_index])

    index_of_nearest_median = get_index_of_nearest_median(unsorted_array,
first_index, last_index, nearest_median)
    swap(unsorted_array, first_index, index_of_nearest_median)

    pivot = unsorted_array[first_index]
    pivot_index = first_index
    index_of_last_element = last_index
    less_than_pivot_index = index_of_last_element
    greater_than_pivot_index = first_index + 1

    ## This while loop is used to correctly place pivot element at its
correct position
    while 1:
        while unsorted_array[greater_than_pivot_index] < pivot and
greater_than_pivot_index < last_index:
            greater_than_pivot_index += 1
        while unsorted_array[less_than_pivot_index] > pivot and less_than_
pivot_index >= first_index:
            less_than_pivot_index -= 1

        if greater_than_pivot_index < less_than_pivot_index:
            temp = unsorted_array[greater_than_pivot_index]
```

```
            unsorted_array[greater_than_pivot_index] = unsorted_
array[less_than_pivot_index]
            unsorted_array[less_than_pivot_index] = temp
        else:
            break

    unsorted_array[pivot_index]=unsorted_array[less_than_pivot_index]
    unsorted_array[less_than_pivot_index]=pivot
    return less_than_pivot_index
```

In the above code, we implement the partition method, which is very similar to what we did in the quickselect algorithm. In the quickselect algorithm, we used a random pivot element (for simplicity, the first element of the list), but in the deterministic selection algorithm, we select the pivot element using the median of medians. The partition method divides the list into two sublists – the left and right sublists, in which the left sublist has elements that are smaller than the pivot element, and the right sublist has elements that are greater than the pivot element. The main benefit of using the pivot element with the median of medians is that it, in general, divides the list into almost two halves.

At the start of the code, firstly, in the `if-else` condition, we check the length of the given list of elements. If the length of the list is 1, then we return the index of that element, so if the `unsorted_array` has only one element, `first_index` and `last_index` will be equal. Therefore, `first_index` is returned. And, if the length is greater than 1, then we call the `median_of_medians()` method to compute the median of medians of the list passed to this method with the starting and ending indices as `first_index` and `last_index`. The return median of medians value is stored in the `nearest_median` variable.

Now, let's understand the code of the `median_of_medians()` method. It is given as follows:

```
def median_of_medians(elems):
    sublists = [elems[j:j+5] for j in range(0, len(elems), 5)]
    medians = []
    for sublist in sublists:
        medians.append(sorted(sublist)[int(len(sublist)/2)])
    if len(medians) <= 5:
        return sorted(medians)[int(len(medians)/2)]
    else:
        return median_of_medians(medians)
```

In the above code of the `median_of_medians` function, recursion is used to compute the median of medians for the given list. The function begins by splitting the given list, `elems`, into groups of five elements each. As discussed earlier in the deterministic algorithm, we divide the given list into groups of 5 elements. Here, we choose 5 elements since it mostly performs well. However, we could have used any other number as well. This means that if `elems` contains 100 items, there will be 20 groups that are created by the `sublists = [elems[j:j+5] for j in range(0, len(elems), 5)]` statement, with each containing a maximum of five elements.

After creating sublists of five elements each, we create an empty array, `medians`, that stores the medians of each of the five-element arrays (i.e., `sublists`). Further, the `for` loop iterates over the list of lists inside `sublists`. Each sublist is sorted, the median is found, and it is stored in the `medians` list. The `medians.append(sorted(sublist)[len(sublist)//2])` statement will sort the list and obtain the element stored in its middle index. The `medians` variable becomes the median list of all the sublists of which there are five elements in each sublist. In this implementation, we use an existing sorting function of Python; it will not impact the performance of the algorithm due to the list's small size.

Thereafter, the next step is to recursively compute the median of medians, which we will use as a pivot element. It is important to note here that the length of the median array can itself be a large array because if the original length of the array is n, then the length of the median array will be n/5, and sorting this may be time-consuming in itself. Hence, we check the length of the `medians` array, and if it is less than 5, we sort the `medians` list and return the element located in its middle index. If, on the other hand, the size of the list is greater than five, we recursively call the `median_of_medians` function again, supplying it with the list of the medians stored in `medians`. Finally, the function returns the median of medians of the given list of elements.

Let's take another example to better understand the concept of the median of medians with the following list of numbers:

```
[2, 3, 5, 4, 1, 12, 11, 13, 16, 7, 8, 6, 10, 9, 17, 15, 19, 20, 18, 23,
21, 22, 25, 24, 14]
```

We can break this list down into groups of five elements, each with the `sublists = [elems[j:j+5] for j in range(0, len(elems), 5)]` code statement, in order to obtain the following list:

```
[[2, 3, 5, 4, 1], [12, 11, 13, 16, 7], [8, 6, 10, 9, 17], [15, 19, 20, 18,
23], [21, 22, 25, 24, 14]]
```

Each of the five-element lists will be sorted as follows:

```
[[1, 2, 3, 5, 5], [7, 11, 12, 13, 16], [6, 8, 9, 10, 17], [15, 18, 19, 20,
23], [14, 21, 22, 24, 25]]
```

Next, we obtain their medians to produce the following list:

```
[3, 12, 9, 19, 22]
```

We sort the above list:

```
[3, 9, 12, 19, 22]
```

Since the list is five elements in size, we only return the median of the sorted list, which is `12` in this case. Otherwise, if the length of this array had been greater than 5, we would have made another call to the `median_of_median` function.

Once we have the median of the median value, we need to find out its index in the given list. We write the `get_index_of_nearest_median` function for this purpose. This function takes the starting and ending indices of the list indicated by the `first` and `last` parameters:

```
def get_index_of_nearest_median(array_list, first, last, median):
    if first == last:
        return first
    else:
        return array_list.index(median)
```

Next in the partition method, we swap the median of medians value with the first element of the list, that is, we swap `index_of_nearest_median` with `first_index` of the `unsorted_array` using the `swap` function:

```
swap(unsorted_array, first_index, index_of_nearest_median)
```

The `utility` function to swap two array elements is shown here:

```
def swap(array_list, first, index_of_nearest_median):
    temp = array_list[first]
    array_list[first] = array_list[index_of_nearest_median]
    array_list[index_of_nearest_median] = temp
```

We swap these two elements. The rest of the implementation is quite similar to what we discussed in the `quick_select` algorithm. Now, we have the median of the median for the given list, which is stored in `first_index` of the unsorted list.

Now, the rest of the implementation is similar to the partition method of the `quick_select` algorithm and also the quicksort algorithm, which is discussed in detail in *Chapter 11*, *Sorting*. For the completeness of the algorithm here, we discuss this again.

We consider the first element as a pivot element, and we take two pointers, that is, left and right. The left pointer moves from the left to the right direction in the list to keep elements that are smaller than the pivot element on the left hand side of the pivot element. It is initialized with the second element of the list, that is, `first_index+1`, whereas the right pointer moved from the right to the left direction, which maintains the list in a way that elements greater than the pivot element are on the right-hand side of the pivot element in the list. It is initialized with the last element of the list. So we have two variables `less_than_pivot_index` (the right pointer) and `greater_than_pivot_index` (the left pointer) in which `less_than_pivot_index` is initialized with `index_of_last_element` and `greater_than_pivot_index` with `first_index + 1`:

```
    less_than_pivot_index = index_of_last_element
    greater_than_pivot_index = first_index + 1
```

Next, we move the left and right pointers in such a way that after one iteration, the pivot element is placed in its correct position in the list. That means it divides the list into two sublists such that the left sublist has all the elements that are smaller than the pivot element, and the right sublist has elements greater than the pivot element. We do this with these three steps given below:

```
    ## This while loop is used to correctly place pivot element at its
correct position
    while 1:
        while unsorted_array[greater_than_pivot_index] < pivot and
greater_than_pivot_index < last_index:
            greater_than_pivot_index += 1
        while unsorted_array[less_than_pivot_index] > pivot and less_than_
pivot_index >= first_index:
            less_than_pivot_index -= 1

        if greater_than_pivot_index < less_than_pivot_index:
            temp = unsorted_array[greater_than_pivot_index]
            unsorted_array[greater_than_pivot_index] = unsorted_
array[less_than_pivot_index]
            unsorted_array[less_than_pivot_index] = temp
        else:
            break
```

1. The first while loop will move `greater_than_pivot_index` to the right side of the array until the element pointed out by `greater_than_pivot_index` is less than the pivot element and `greater_than_pivot_index` is less than `last_index`:

```
while unsorted_array[greater_than_pivot_index] < pivot and greater_
than_pivot_index < last_index: greater_than_pivot_index += 1
```

2. In the second `while` loop, we'll be doing the same thing but for the `less_than_pivot_index` in the array. We'll move `less_than_pivot_index` to the left direction until the element pointed out by `less_than_pivot_index` is greater than the pivot element and `less_than_pivot_index` is greater than or equal to `first_index`:

```
while unsorted_array[less_than_pivot_index] > pivot and less_than_
pivot_index >= first_index: less_than_pivot_index -= 1
```

3. Now, we check if `greater_than_pivot_index` and `less_than_pivot_index` have crossed each other or not. If `greater_than_pivot_index` is still less than `less_than_pivot_index` (that is, we have not found the correct position for the pivot element yet), we swap the elements indicated by `greater_than_pivot_index` and `less_than_pivot_index`, and then we will repeat the same three steps again. If they have crossed each other, that means we have found the correct position for the pivot element, and we will break from the loop:

```
if greater_than_pivot_index < less_than_pivot_index:
    temp = unsorted_array[greater_than_pivot_index]
    unsorted_array[greater_than_pivot_index] = unsorted_array[less_
than_pivot_index]
    unsorted_array[less_than_pivot_index] = temp
else:
    break
```

After exiting the loop, the variable `less_than_pivot_index` will point to the correct index of the pivot, so we will just swap the values that are present at `less_than_pivot_index` and `pivot_index`:

```
    unsorted_array[pivot_index]=unsorted_array[less_than_pivot_index]
    unsorted_array[less_than_pivot_index]=pivot
```

Finally, we will simply return the pivot index, which is stored in the variable `less_than_pivot_index`.

After the partition method, the pivot element reaches its correct position in the list. Thereafter, we recursively call the partition method to one of the sublists (the left sublist or the right sublist) depending on the required value of `k` and the pivot element position to find out the k^{th} smallest element. This process is the same as the quickselect algorithm.

The implementation of the deterministic select algorithm is given as follows:

```
def deterministic_select(array_list, start, end, k):
    split = partition(array_list, start, end)
    if split == k:
        return array_list[split]
    elif split < k:
        return deterministic_select(array_list, split + 1, end, k)
    else:
        return deterministic_select(array_list, start, split-1, k)
```

As you may have observed, the implementation of the deterministic selection algorithm looks exactly the same as the quickselect algorithm. The only difference between the two is how we select the pivot element; apart from that, everything is the same.

After the initial `array_list` has been partitioned by the selected pivot element, which is the median of medians of the list, a comparison with the k^{th} element is made:

1. If the index of the split point, that is, `split`, is equal to the required value of `k`, it means that we have found the required k^{th} smallest element.
2. If the index of the split point, the, `split` is less than the required value of `k`, then a recursive call to the right sublist is made as `deterministic_select(array_list, split + 1, right, k)`. This will look for the k^{th} element on the right-hand side of the array.
3. Otherwise, if the split index is greater than the value of `k`, then the function call to the left sublist is made as `deterministic_select(array_list, left, split-1, k)`.

The following code snippet can be used to create a list and further use the deterministic algorithm to find out the k^{th} smallest element from the list:

```
list1= [2, 3, 5, 4, 1, 12, 11, 13, 16, 7, 8, 6, 10, 9, 17, 15, 19, 20, 18,
23, 21, 22, 25, 24, 14]

print("The 6th smallest element is", deterministic_select(list1, 0,
len(list1)-1, 5))
```

The output of the above code is as follows.

```
The 6th smallest element is 6
```

In the output of the above code, we have the 6^{th} smallest element from a given list of 25 elements. The deterministic selection algorithm improves the quickselect algorithm by using the median of medians element as a pivot point for selecting the k^{th} smallest element from a list. It improves performance because the median of medians method finds out the estimated median in linear time, and when this estimated median is used as a pivot point in the quickselect algorithm, the worst-case running time's complexity improves from $O(n^2)$ to the linear $O(n)$.

The median of medians algorithm can also be used to choose a pivot point in the quicksort algorithm for sorting a list of elements. This significantly improves the worst-case performance of the quicksort algorithm from $O(n^2)$ to a complexity of $O(nlogn)$.

Summary

In this chapter, we discussed two important methods to find the k^{th} smallest element in a list, randomized selection and deterministic selection algorithms. The simple solution of merely sorting a list to perform the operation of finding the k^{th} smallest element is not optimal as we can use better methods to determine the k^{th} smallest element. The quickselect algorithm, which is the random selection algorithm, divides the list into two sublists. One list has smaller values, and the other list has greater values as compared to the selected pivot element. We reclusively use one of the sublists to find the location of the k^{th} smallest element, which can be further improved by selecting the pivot point using the median of medians method in the deterministic selection algorithm.

In the next chapter, we will discuss several important string matching algorithms.

Exercise

1. What will be the output if the quickselect algorithm is applied to the given array

   `arr = [3, 1, 10, 4, 6, 5]` with k given as 2?

2. Can quickselect find the smallest element in an array with duplicate values?
3. What is the difference between the quicksort algorithm and the quickselect algorithm?
4. What is the main difference between the deterministic selection algorithm and the quickselect algorithm?
5. What triggers the worst-case behavior of the selection algorithm?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`