1 Selection

1.1 Approach-1

Approach is described in Algorithm 1.

Algorithm 1 SELECT-SORT(`arr[ ]`, `low`, `high`, k)

```
SORT(arr[low, ..., high])          ▷ Sort the array arr[ ] within the range low to high
return arr[k]                      ▷ k^th smallest element
```

$$T(n) = \mathcal{O}(n \log n) \tag{1}$$

1.2 Approach-2

Approach is described in Algorithm 2.

Algorithm 2 SELECT-MIN-HEAP(`arr[ ]`, `low`, `high`, k)

```
n ← high − low + 1                 ▷ Number of elements in the array arr[ ] within the range low to high
BUILD-MIN-HEAP(arr[ ])             ▷ Create min-heap from n elements of the array arr[ ]
for i ← 1 to k − 1 do
    element ← HEAP-EXTRACT-MIN(arr[ ])   ▷ Delete the smallest element from the heap
end for
element ← HEAP-MINIMUM(arr[ ])     ▷ Retrieve the smallest element from the heap
return element                     ▷ k^th smallest element
```

$$T(n) = \mathcal{O}(n + k \log n) \tag{2}$$

1.3 Approach-3

Approach is described in Algorithm 3.

Algorithm 3 SELECT-MAX-HEAP(`arr`, `low`, `high`, k)

```
n ← high − low + 1                 ▷ Number of elements in the array arr[ ] within the range low to high
arr'[1, ..., k] ← arr[low, ..., low + k − 1]   ▷ Create an array arr'[ ] from the initial k elements of the array arr[ ]
BUILD-MAX-HEAP(arr')               ▷ Create max-heap from k elements of the array arr'[ ]
for i ← low + k to high do         ▷ Check remaining n − k elements of the array arr[ ]
    element ← arr[i]               ▷ i^th element of the array arr[ ]
    if element < arr'[1] then      ▷ element is less than the root of the heap
        arr'[1] ← element          ▷ Make element root of the heap
        MAX-HEAPIFY(arr', 1)
    else
                                   ▷ Do nothing
    end if
end for
element ← HEAP-MAXIMUM(arr')
return element                     ▷ k^th smallest element
```

$$T(n) = \mathcal{O}(k + (n - k) \log k) \tag{3}$$

1.4 Approach-4

Approach is described in Algorithm 4.

Algorithm 4 SELECT($\texttt{arr}, \texttt{low}, \texttt{high}, k$)

```
1:  n ← high − low + 1                          ▷ Number of elements in the array arr[ ] within the range low to high
2:  if k > 0 && k ≤ n then                      ▷ Check whether rank k is valid or not
3:      if low = high then
4:          return arr[low]
5:      end if
6:      pos_pivot ← PARTITION(arr, low, high)   ▷ Partition the array arr[ ] within the range low to high, around
        the pivot element and get the actual position (position in the sorted array) of pivot element in the array
7:      rank_pivot ← pos_pivot − low + 1        ▷ Rank of pivot in the array arr[ ] within the range low to high
8:      if rank_pivot = k then                  ▷ k is same as the rank of the pivot
9:          return A[rank_pivot]
10:     else if k < rank_pivot then             ▷ k is smaller than the rank of the pivot
11:         return SELECT(arr, low, pos_pivot − 1, k)
12:     else                                    ▷ k is greater than the rank of the pivot
13:         return SELECT(arr, pos_pivot + 1, high, k − rank_pivot)
14:     end if
15: else
16:     k is not valid
17: end if
```

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2) \tag{4}$$

1.5 Approach-5

Approach is described in Algorithm 5. This approach finds the k^{th} smallest element in an array `arr` whose start and end indices are `low` and `high` respectively. This algorithm assumes that the array elements are distinct.

Algorithm 5 SELECT-LINEAR(`arr[ ]`, `low`, `high`, k)

1: $n \leftarrow$ `high` $-$ `low` $+ 1$ ▷ Number of elements in the array `arr[ ]` within the range `low` to `high`
2: **if** $k > 0$ **&&** $k \leq n$ **then** ▷ Check whether rank k is valid or not
3: **if** $n \leq 5$ **then**
4: Sort the array `arr[ ]` within the range `low` to `high`
5: **return** `arr`$[k]$
6: **end if**
7: `noGroup` $\leftarrow \lceil \frac{n}{5} \rceil$
8: `medians`$[1 \ldots$ `noGroup`$] \leftarrow \emptyset$ ▷ Array to store the median of small arrays (maximum size 5)
9: `startIndex` $\leftarrow$ `low`
10: **for** $i \leftarrow 1$ **to** `noGroup` $- 1$ **do**
11: `medians`$[i] \leftarrow$ MEDIAN(`arr[ ]`, `startIndex`, `startIndex` $+ 4$) ▷ Median of 5 elements `arr`[`startIndex`, `startIndex` $+ 1, \ldots,$ `startIndex` $+ 4$]. Use Algorithm 6
12: `startIndex` $\leftarrow$ `startIndex` $+ 5$ ▷ Update `startIndex`
13: **end for**
14: `medians`[`noGroup`] $\leftarrow$ MEDIAN(`arr[ ]`, `startIndex`, `high`) ▷ Median of last group which can have less than 5 elements. Use Algorithm 6
15: `medofMedians` $\leftarrow$ SELECT-LINEAR(`medians[ ]`, 1, `noGroup`, $\frac{\texttt{noGroup}}{2}$) ▷ Recur to obtain median of medians
▷ Find the position of `medofMedians` in the array `arr[ ]` within the range `low` to `high`
16: **for** $i \leftarrow$ `low` **to** `high` **do**
17: **if** `arr`$[i] =$ `medofMedians` **then**
18: BREAK
19: **end if**
20: **end for**
21: $\texttt{pos}_{\texttt{medofMedians}} \leftarrow i$ ▷ Position of `medofMedians`
22: Swap `arr`[`high`] with `arr`$[\texttt{pos}_{\texttt{medofMedians}}]$ so that the `medofMedians` take the last place in the array. This is required because the "PARTITION()" procedure which we know for Quick Sort consider the last element as the pivot, and we need to partition the array around the `medofMedians`
23: $\texttt{pos}_{\texttt{medofMedians}} \leftarrow$ PARTITION(`arr[ ]`, `low`, `high`) ▷ Partition the array `arr[ ]` within the range `low` to `high`, around the `medofMedians` and get the actual position (*position in the sorted array*) of `medofMedians` in the array. Use Algorithm 7
24: $\texttt{rank}_{\texttt{medofMedians}} \leftarrow \texttt{pos}_{\texttt{medofMedians}} - \texttt{low} + 1$ ▷ Rank of `medofMedians` in the array `arr[ ]` within the range `low` to `high`
25: **if** $\texttt{rank}_{\texttt{medofMedians}} = k$ **then** ▷ k is same as rank of `medofMedians`
26: **return** `arr`$[\texttt{pos}_{\texttt{medofMedians}}]$
27: **else if** $k < \texttt{rank}_{\texttt{medofMedians}}$ **then** ▷ k is smaller than the rank of `medofMedians`
28: **return** SELECT-LINEAR(`arr[ ]`, `low`, $\texttt{pos}_{\texttt{medofMedians}} - 1, k$)
29: **else** ▷ k is greater than the rank of `medofMedians`
30: **return** SELECT-LINEAR(`arr[ ]`, $\texttt{pos}_{\texttt{medofMedians}} + 1$, `high`, $k - \texttt{rank}_{\texttt{medofMedians}}$)
31: **end if**
32: **else**
33: k is not valid
34: **end if**

Algorithm 6 MEDIAN(arr[ ], start, end)

Sort arr[start, ..., end] ▷ Sort the array arr[ ] within the range start to end
pos ← start + $\frac{\text{end}-\text{start}}{2}$ ▷ Find the position of the median of the array arr[ ] within the range start to end
return arr[pos] ▷ Median of the array arr[ ] within the range start to end

Algorithm 7 PARTITION(arr[ ], low, high)

$x \leftarrow$ arr[high]
$i \leftarrow$ low $- 1$
for $j \leftarrow$ low **to** high $- 1$ **do**
    if arr$[j] \leq x$ **then**
        $i \leftarrow i + 1$
        Swap arr$[i]$ with arr$[j]$
    end if
end for
Swap arr$[i + 1]$ with arr[high]
return $i + 1$