QUICK SELECT

Hume ek array given hai (sorted nahi hota array). Hume kaha jata hai ki Hume k^{th} smallest element nikal kr dijiye!

Quick Select Algorithm will enable use to solve this problem in LINEAR TIME COMPLEXITY.

- Hume ek Array given hai and hum 4^{th} → (K) smallest element

pivot

2	8	1	3	7	6	4	5

∴ 4^{th} smallest element = 4 → issko index me convert krlo → INDEX 3 → (K-1)

↓ After Partitioning

0	1	2	3	4	5	6	7
2	1	3	4	5	8	7	6

compare

Hum iss index ko dundh rahy hai!
↓
issko har baar pivot index se compare krengy!

- Aab hum pivot ke index ko humare desired index se compare krengy!

pivot index = 4
desired index = 3

$\left\{ \begin{array}{l} \therefore 4 \neq 3 \\ \therefore 4 > 3 \end{array} \right\}$

- pivot index bada hai humare desired index toh hum LHS ko agge use krengy! → LHS wale elements me partitioning hogi aur tab tak hogi jab tak hume desired element index nahi mil jata!

pivot

2	1	3	4

↓ After partitioning

0	1	2	3
2	1	3	4

comparison

- pivot index = 3
desired index = 3

$\therefore \boxed{\text{pivot index} = \text{desired index}}$

- Print that index element

```
public static int quickSelect (int []arr, int lo, int hi, int k)
{ int pivot = arr[hi];
  int pivotindex = partition (arr, lo, hi, pivot);
```

partition krke pivot ka correct index !

NOW PIVOT will at it's correct position !

```
    if (k== pivotindex) {
        return pivot;
    }
```

we are at right position!

```
    else if (k > pivotindex){
    return quickSelect (arr, pivotindex+1, hi, k);
    }
```

K bada hai toh RHS wale element me partition hoga

```
    else {
    return quickSelect (arr, lo, pivotindex-1, K);
    }
}
```

K chota hai toh LHS wale element me partition hoga

```
public static int partition (int [] arr, int lo, int hi, int pivot){
  int i = lo;
  int j = lo;
  while (lo <= hi)
  { if (arr[i] > pivot)
      {
        i++;
      }
    else {
      swap (arr, i, j);
      i++;
      j++;
    }
  }
  System.out.println ("pivot index ->"+ (j-1));
  return j-1;
}
public static void print (int [] arr) {
    for (int i = 0; i< arr.length; i++) {
        System.out.print (arr[i] + "");
    }
    System.out.println();
}
```

```java
public static void main(String [] args) {
  Scanner s = new Scanner(System.in);
   int n = s.nextInt();
   int arr[] = new int[n];
   for(int i=0; i<n; i++){
      arr[i] = s.nextInt();
   }
  int k = s.nextInt();
  System.out.println(quickSelect(arr, 0, arr.length-1, K-1));
 }
public static int swap (int[] arr, int i, int j) {
  System.out.println("Swapping" + arr[i] + "and" + arr[j]);
   int temp = arr[i];
   arr[i] = arr[j];
     arr[j] = temp;
 }
```

Time Complexity

$T(n) = n + T(n/2)$

Time Taken For Partitioning

Time for recursion calls (Assuming partition always happens in the middle of the list)

$T(n) = n + T(\cancel{n/2})$

$T(\cancel{n/2}) = n/2 + T(\cancel{n/4})$

$T(\cancel{n/4}) = n/4 + T(\cancel{n/8})$

$T(\cancel{n/2^{x-1}}) = n/2^x + T(n/2^x)$

$T(n) = n + n/2 + n/4 + \dots n/2^x + T(n/2^x)$

$T(n) = n\Big[\underbrace{1+\frac{1}{2}+\frac{1}{2^2}+\dots+\frac{1}{2^x}}_{G.P}\Big] + T(n/2^x)$

Since we stop at the base case

$\therefore T\left(\frac{n}{2^x}\right) = T(1) \Rightarrow \boxed{n = 2^x}$ or $\boxed{x = \log_2 n}$

$T(n) = n\left[1+\frac{1}{2}+\frac{1}{2^2}+\dots+\frac{1}{2^x}\right] + T(1)$

$= n\left[\frac{1(1-(1/2)^x)}{1/2}\right] + T(1)$

$\therefore$ put $\boxed{x = \log_2 n}$ and $\boxed{T(1) = 1}$

$T(n) = 2n\left[1-(2)^{-\log_2 n}\right] + 1$

$= 2n\left[1-(2)^{\log_2(1/n)}\right] + 1 \quad \{\because -\log_a b = \log_a(1/b)\}$

$= 2n\left[1-\frac{1}{n}\right] + 1 \quad \because \{a^{\log_a b} = b\}$

$T(n) = 2n - 2 + 1 = 2n - 1 \longrightarrow \therefore \boxed{T(n) = O(n)}$

Best Case $\rightarrow O(n)$

Worst Case $\rightarrow O(n^2)$