# Quick Sort

**(Class 13)**

From Book’s Page No 182 (Chapter 7)

- Our next sorting algorithm is Quicksort.
- It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries.
- Quicksort is based on the divide and conquer strategy.
- Here is the algorithm:

```
QUICKSORT (array A, int p, int r)
if (p < r)
    i ← a random index from [p…r]
    swap A[i] with A[p]
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

# Partition Procedure

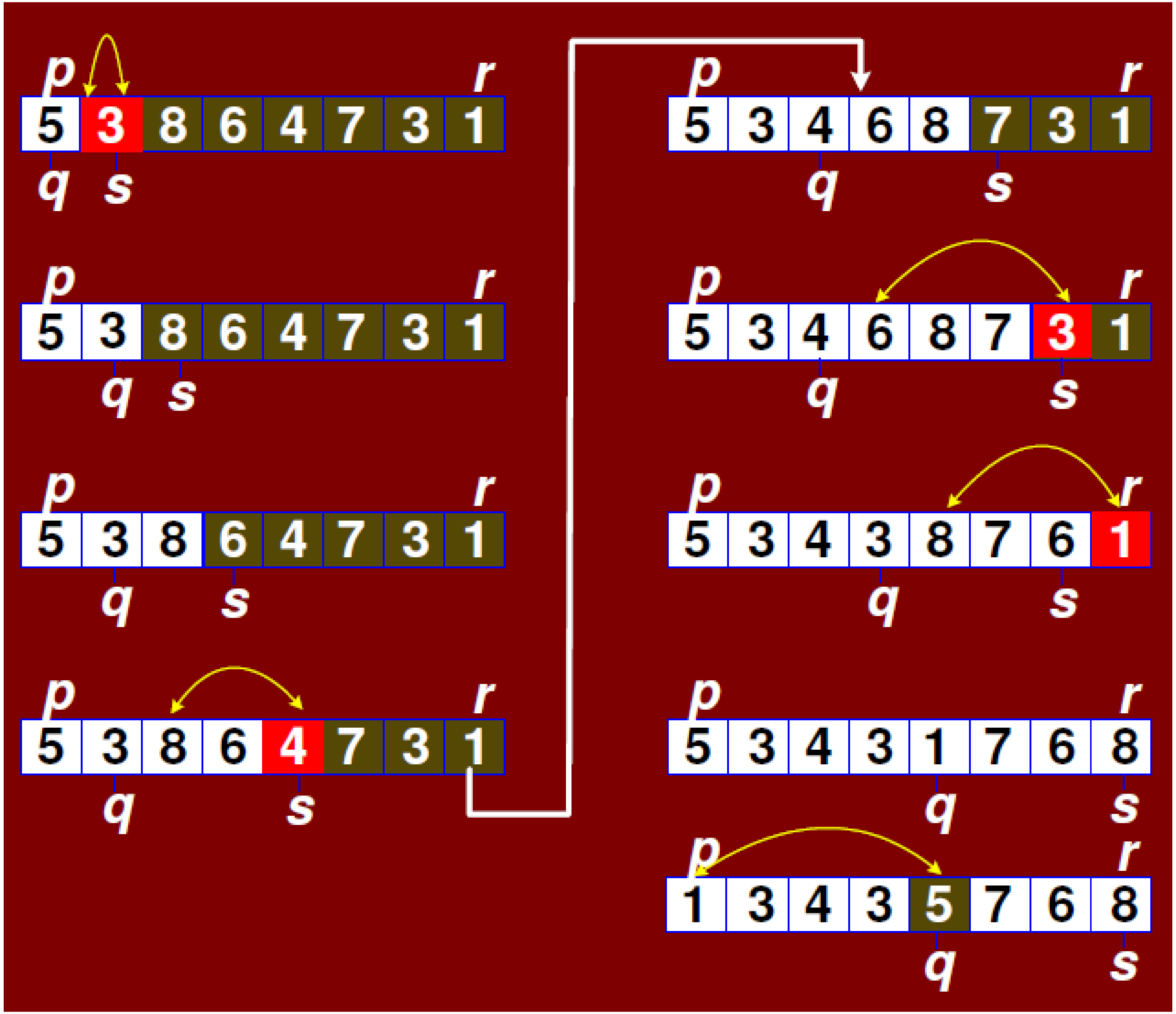
- Recall that the partition algorithm partitions the array $A[p \ldots r]$ into three sub arrays about a pivot element $x$.
  - $A[p \ldots q-1]$ whose elements are less than or equal to $x$.
  - $A[q] = x$ was pivot.
  - $A[q+1 \ldots r]$ whose elements are greater than $x$.

# Choosing the Pivot

- In quick sort, we will choose the first element of the array as the pivot, i.e., $x = A[p]$.
- If a different rule is used for selecting the pivot, we can swap the chosen element with the first element.
- We will choose the pivot randomly.

- The algorithm works by maintaining the following invariant condition.
  - $A[p] = x$ is the pivot value.
  - $A[p \ldots q-1]$ contains elements that are less than $x$.
  - $A[q+1 \ldots s-1]$ contains elements that are greater than or equal to $x$.
  - $A[s \ldots r]$ contains elements whose values are currently unknown.

```
PARTITION (array A, int p, int r)
x ← A[p]
q ← p
for s ← p+1 to r
  if (A[s] < x)
    q ← q+1
    swap A[q] with A[s]
swap A[p] with A[q]
return q
```


p r
5 3 8 6 4 7 3 1
q s
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 8 6 4 7 3 1
q s
p r
5 3 4 6 8 7 3 1
q s
p r
5 3 4 6 8 7 3 1
q s
p r
5 3 4 3 8 7 6 1
q s
p r
5 3 4 3 1 7 6 8
q s
p r
1 3 4 3 5 7 6 8
q s

# Quick Sort Example

- With the help of a diagram, we trace out the quick sort algorithm.
- The first partition is done using the last element, 10, of the array.
- The left portion are then partitioned about 5 while the right portion is partitioned about 13.
- Notice that 10 is now at its final position in the eventual sorted order.
- The process repeats as the algorithm recursively partitions the array eventually sorting it.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |

⇓

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |

⇓

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |

⇓

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |
| 1 | 2 | 4 | 3 | 5 | 8 | 6 | 7 | 9 | 10 | 12 | 11 | 13 | 14 | 15 | 17 | 16 |

$\Downarrow$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |
| 1 | 2 | 4 | 3 | 5 | 8 | 6 | 7 | 9 | 10 | 12 | 11 | 13 | 14 | 15 | 17 | 16 |

$\Downarrow$

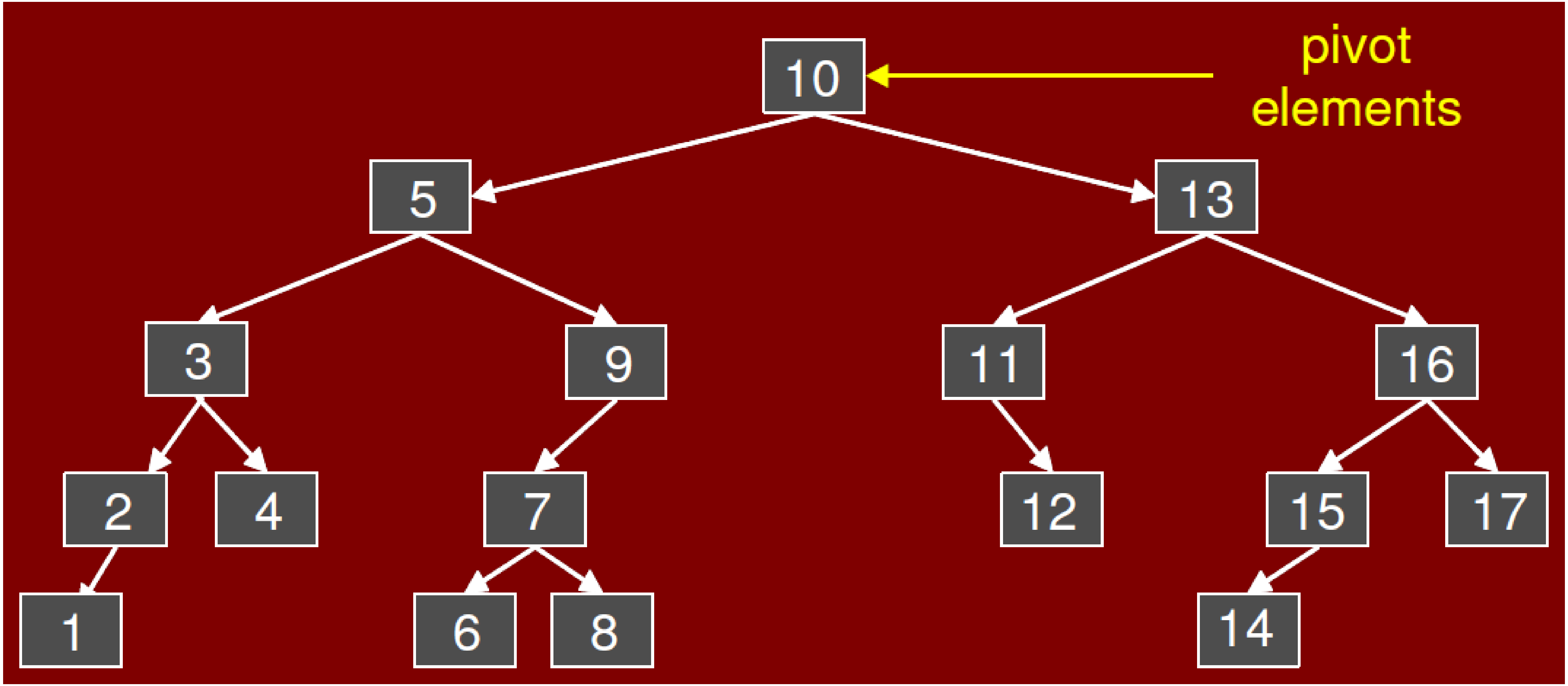
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |
| 1 | 2 | 4 | 3 | 5 | 8 | 6 | 7 | 9 | 10 | 12 | 11 | 13 | 14 | 15 | 17 | 16 |
| 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

$\Downarrow$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |
| 1 | 2 | 4 | 3 | 5 | 8 | 6 | 7 | 9 | 10 | 12 | 11 | 13 | 14 | 15 | 17 | 16 |
| 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

⇓

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 12 | 3 | 11 | 8 | 7 | 1 | 15 | 13 | 17 | 5 | 16 | 14 | 9 | 4 | 10 |
| 7 | 6 | 4 | 3 | 9 | 8 | 2 | 1 | 5 | 10 | 17 | 15 | 16 | 14 | 11 | 12 | 13 |
| 1 | 2 | 4 | 3 | 5 | 8 | 6 | 7 | 9 | 10 | 12 | 11 | 13 | 14 | 15 | 17 | 16 |
| 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |


pivot
elements
10
5
13
3
9
11
16
2
4
7
12
15
17
1
6
8
14

# Analysis of Quicksort

- The running time of quicksort depends heavily on the selection of the pivot.
- If the rank of the pivot is very large or very small, then the partition (BST) will be unbalanced.
- Since the pivot is chosen randomly in our algorithm, the expected running time is $O(n \log n)$.
- The worst-case time, however, is $O(n^2)$.
- Luckily, this happens rarely.

# Worst Case Analysis of Quick Sort

- Let’s begin by considering the worst-case performance because it is easier than the average case.
- Since this is a recursive program, it is natural to use a recurrence to describe its running time.
- But unlike Merge-Sort, where we had control over the sizes of the recursive calls, here we do not.
- It depends on how the pivot is chosen.

- Suppose that we are sorting an array of size $n$, $A[1{:}\,n]$, and further suppose that the pivot that we select is of rank $q$, for some $q$ in the range $1$ to $n$.
- It takes $O(n)$ time to do the partitioning and other overhead, and we make two recursive calls.

- The first is to the subarray $A[1: q \ - \ 1]$ which has $q - 1$ elements.
- And the other call is to the subarray $A[q + 1: n]$ which has $n - q$ elements.
- So, if we ignore the $O(n)$ (as usual) we get the recurrence:

$$T(n) = T(q - 1) + T(n - q) + n$$

- This depends on the value of $q$.
- To get the worst case, we maximize over all possible values of $q$.
- Putting is together, we get the recurrence.

$$T(n) = \begin{cases} 1, & if\ n \leq 1 \\ \max\limits_{1 \leq q \leq n} \left(T(q-1) + T(n-q) + n\right), & if\ n \geq 2 \end{cases}$$

- Recurrences that have max’s and min’s embedded in them are very messy to solve.
- The key is determining which value of $q$ gives the maximum.
- A rule of thumb of algorithm analysis is that the worst-cases tends to happen either at the extremes or in the middle.

- So, we would plug in the value $q = 1$, $q = n$, and $q = \frac{n}{2}$ and work each out.
- In this case, the worst case happens at either of the extremes.
- If we expand the recurrence for $q = 1$, we get:

$$T(n) = T(0) + T(n-1) + n$$

$$= 1 + T(n-1) + n$$

$$T(n) = T(n-1) + (n+1)$$

Following are the iterations:

$$= T(n-2) + \ n + (n+1)$$

$$= T(n-3) + (n-1) + n + (n+1)$$

$$= T(n-4) + (n-2) + (n-1) + n + (n+1)$$

$$T(n) = T(n-k) + \sum_{i=-1}^{k-2} (n - \mathrm{i})$$

- As the pivot is 1st element, there will be 1 element in left subarray and $n - 1$ elements in right subarray at each recursion.
- For the basis $T(1) = 1$ we set $k = n - 1$ and get:

$$T(n) = T(1) + \sum_{i=-1}^{n-3} (n - \mathrm{i})$$

$$= 1 + (3 + 4 + 5 + \ \ldots + (n - 1) + n + (n + 1))$$

$$\leq \sum_{i=1}^{n+1} i = \frac{(n + 1)(n + 2)}{2}$$

$$\boldsymbol{T(n) \in O(n^2)}$$

- In the next class we will discuss the average-case running time of the quick sort.