# Algorithms Analysis and Design

## Chapter 4

## Divide and Conquer
## Part 2

## Quick sort

- Quicksort is a widely used sorting algorithm that follows the divide-and-conquer approach.
- It works by selecting a pivot element from the array and partitioning the other elements into two subarrays, according to whether they are less than or greater than the pivot. The subarrays are then recursively sorted using the same process.

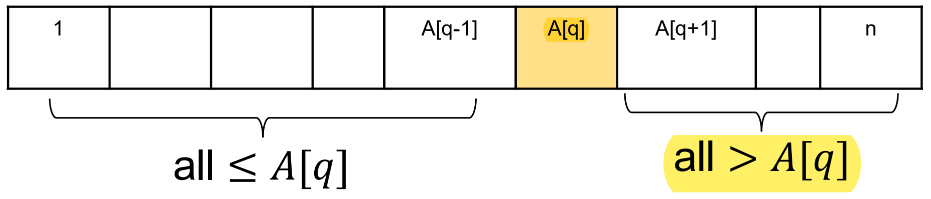
# Quick sort

- A *sub-problem* of one element is already sorted [Base case]

**To sort a large instance:**

- Select an element as a **pivot**.
- Partition the array into two subarrays around the selected pivot such that:
  - All elements in the left group are < pivot
  - All elements in the right group are ≤pivot

Divide



- Sort left and right groups recursively
- Combine: Trivial.

Conquer

## Quick sort

- The key process in quicksort is a **partition**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

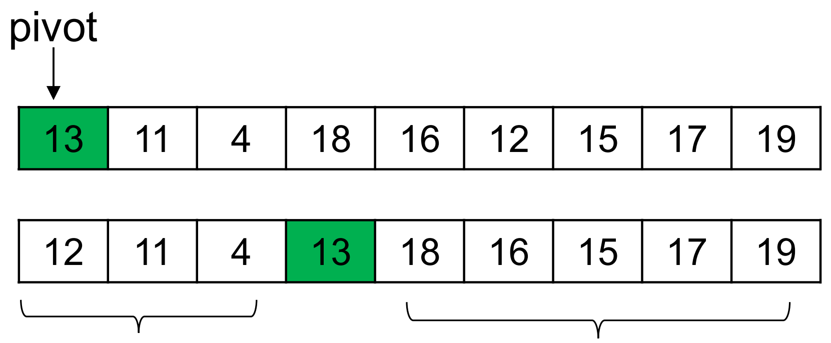
- This partition is done recursively which finally sorts the array. See the below image for a better understanding.

# Pivot selection

- There are many different versions of quickSort that pick pivot in different ways:
  - Pick the **first element** as pivot.
  - Pick the **last element** as pivot
  - Pick a **random** element as pivot.
  - Pick the **median** as pivot.
  - Other techniques……..
- Example: Select a random pivot.

[illegible]

For now, we use the simplest strategy of selecting the subarray's **first element** (pivot = A[1])



- Put the smalls on the left and bigs on the right

**The partitioning procedure is linear-time**

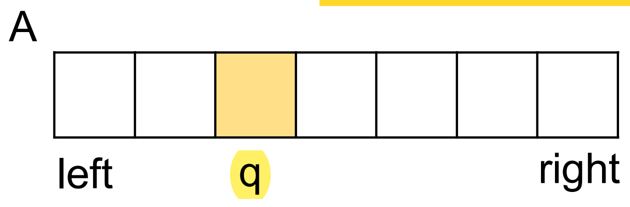
*It requires O(n) time*

# Pseudocode of Quick sort

**q** is the pivot index

```
ALGORITHM $QuickSort\ (\ Arr\ ,left,\ \ right\ )$
{
        $\boldsymbol{if}\ (left\ \ < right)\ \boldsymbol{then}$
Pivot [illegible]   $q\ \ = \boldsymbol{Partitiopn}\ (Arr\ ,left,$ ~~mid~~ right $)$   divide
            $QuickSort(\ Arr\ ,left,q\ \ - 1)$
            $QuickSort(\ Arr\ ,q + 1,right)$
        endif
}
```

**Initial call QuickSort (A, 1, *n*)**



- When **left = right**, there is one element (the problem is small enough)
- **left < right** : indicates that there is at least two elements (**solved recursively**)

***Partitiopn***() **Partition the array into two subarrays around a pivot A[q] such that**

**elements in lower subarray ≤ A[q] ≤ elements in upper subarray.**

***It requires O(n) time***

## Pseudocode of Quick sort

**q** is the pivot index

```
ALGORITHM $QuickSort\ (\ Arr\ , left,\ \ right\ )$
{
        $\boldsymbol{if}\ (left\ < right)\ \boldsymbol{then}$
            $q\ = \boldsymbol{Partitiopn}\ (Arr\ , left, mid)$        // Get the pivot index
            $QuickSort(\ Arr\ , left, q\ - 1)$        // Recursively sort the left subarray
            $QuickSort(\ Arr\ , q + 1, right)$        // Recursively sort the right subarray
        endif
}
```

# Analysis of Quick Sort

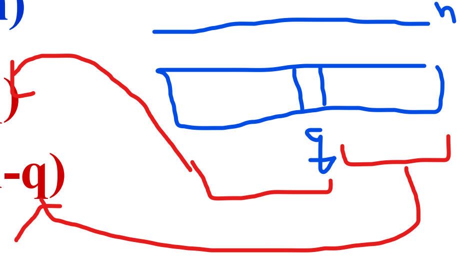
```
ALGORITHM $QuickSort\,(Arr, left, right)$
{
        $\textbf{if}\ (left < right)\ \textbf{then}$
            $q = \textbf{Partitiopn}\,(Arr, left, mid)$
            $QuickSort(Arr, left, q-1)$
            $QuickSort(Arr, q+1, right)$
        endif
}
```

- Merge sort divides the array into two nearly equal parts
- **Quick sort depends on the choice of the pivot.**

$O(n)$

$T(q)$

$T(n\text{-}q)$



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

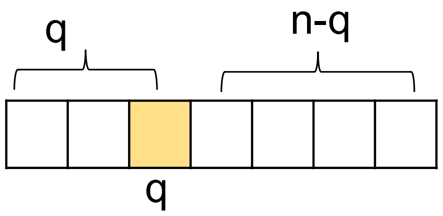
$$T(n) = T(n-q) + T(q) + an + b$$

## Analysis of Quick Sort

We need to study the three cases **(Best , worst, and Average)**

- **How efficient is Quick sort???**
- Based on the general recurrence : $T(n) = a\,T\left(\frac{n}{b}\right) + cost\ of\ divide + cost\ of\ merge$

- The problem is divided into 2 subproblems. **The size of each subproblem depends on the choice of the pivot.**
- Cost of partitioning the problem: ***O*** **( n )**



- solving 2 sub-problems. The first of size q takes T(q) and the other subproblem of size n-q takes T(n-1)
- Cost of combining the solutions : ***Not required***
- Total:

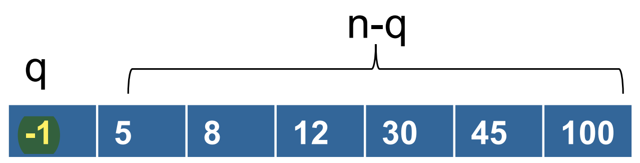
$$T(n) = T(n-q) + T(q) + O(n) \qquad \textbf{if } n > 1$$

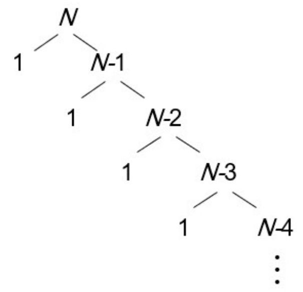
## Analysis of Quick Sort [Worst case]

$$T(n) = T(n-q) + T(q) + O(n) \qquad \text{if } n > 1$$

We need to study the three cases **(Best , worst, and Average)**

- Worst case of quick sort: **(Completely unbalanced partitioning / Skewed)**
  - Input sorted or reversed sorted.
  - partitioning around min or max element.
  - One side of partitioning always has no elements.





$$T(n) = T(n-1) + T(0) + O(n) \qquad \text{if } n > 1$$

$T(1)$ **is constant (O(1))**

$$T(n) = T(n-1) + an + b \qquad \text{if } n > 1$$

$$T(n) = O(n^2)$$

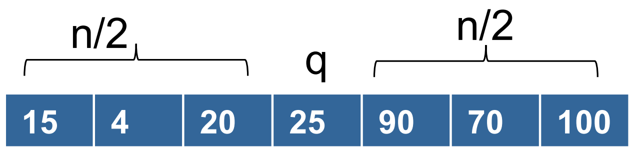
***Hint: Solve*** $T(n) = T(n-1) + n$ **if** $n > 1$ **T(1) = 1 using iterative method**

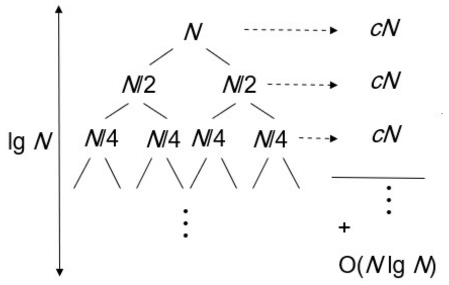
# Analysis of Quick Sort [Best case]

$$T(n) = T(n-q) + T(q) + O(n) \qquad \text{if } n > 1$$

We need to study the three cases **(Best , worst, and Average)**

- Best case of quick sort: (**Balanced partitioning)**
  - The Partition always balances the two sides of the partition equally
  - 2 parts each of size n/2





$$T(n) = T(n/2\ ) + T(n/2) + O(n) \qquad \text{if } n > 1$$

**$T(1)$ is constant (O(1))**

$$T(n) = 2T(n/2\ ) + an + b$$

$$T(n) = O(\ n \log n\ )$$

***Hint: Solve*** $T(n) = 2\,T(n/2\ ) + n$ **if** $n > 1$ **using master method**

# Analysis of Quick Sort

$T(n) = T(n-q) + T(q) + O(n)$ if $n > 1$

We need to study the three cases **(Best , worst, and Average)**

**The worst case of QUICKSORT in which the Partition procedure always puts only a single element on one side of the partition.**

**The resulting running time is $\Theta(n^2)$**

**The best case of QUICKSORT in which the Partition always balances the two sides of the partition equally (the best case).**

**The resulting running time is $\Theta(n \lg n)$**

**If we're lucky, PARTITION splits the array evenly $T(n) = 2\,T(n/2) + O(n)$**

# Exercise

a) In quick sort algorithm, What kind of input produces the worst-case behavior?

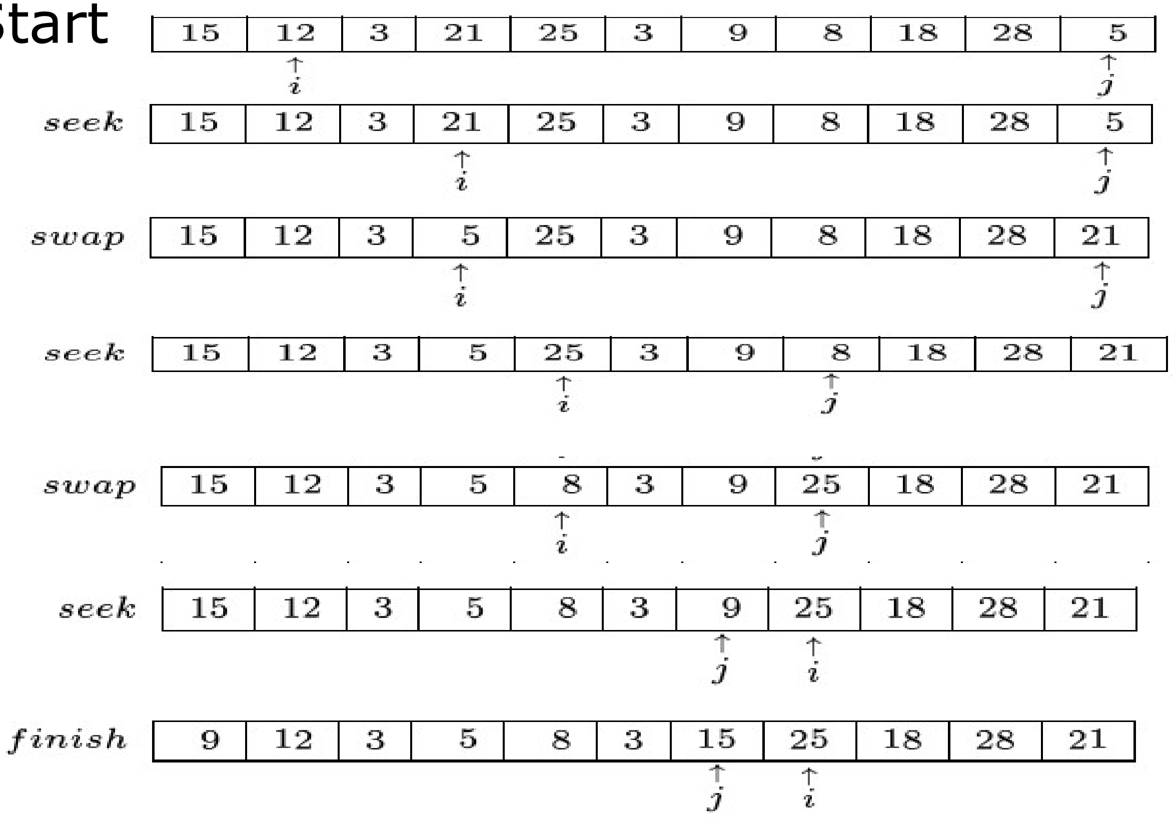
b) Setup the recurrence relation of T(n) for the Quick Sort algorithm when the split is always 1/10 : 9/10

$\boxed{1/10 \mid 9/10}$

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$

## Example of partitioning an array (Hoare's algorithm)

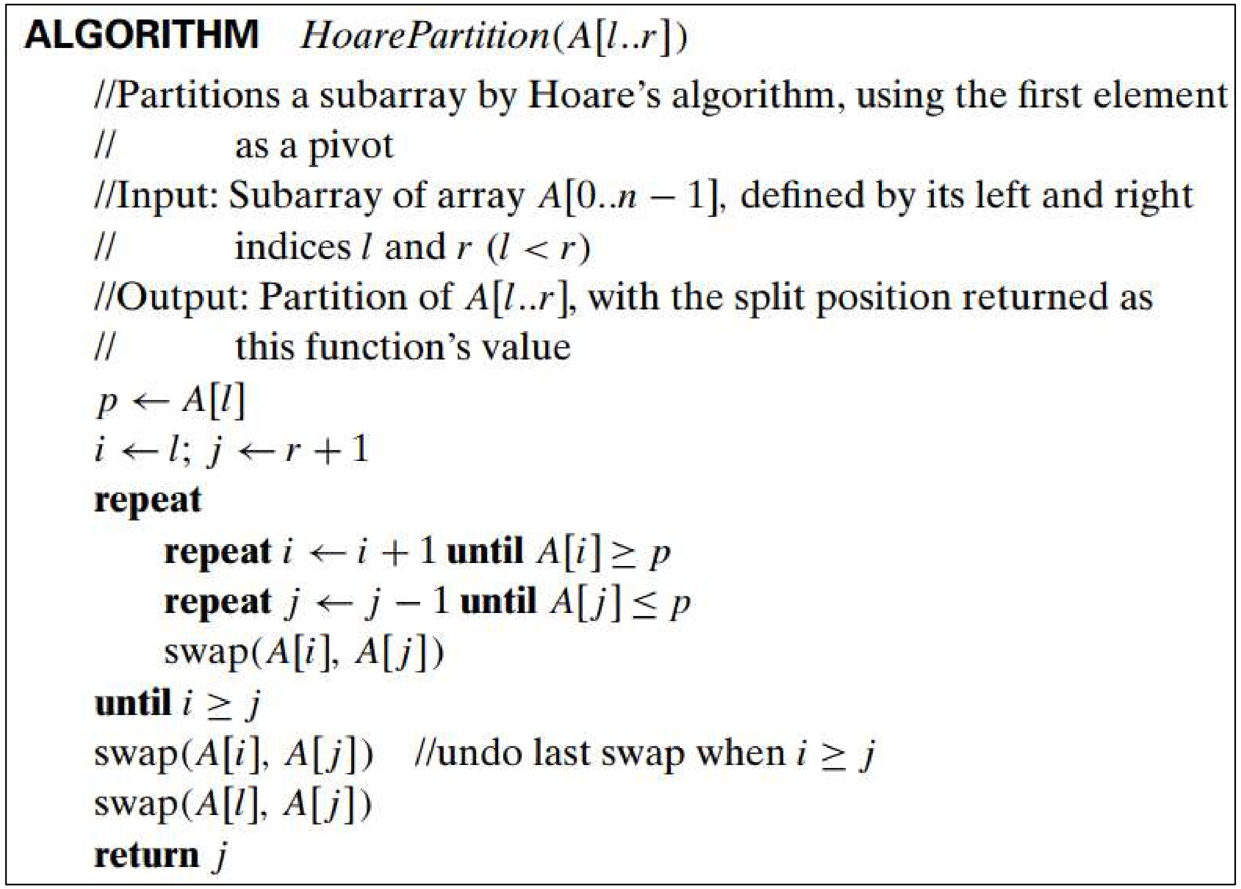
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Input | 15 | 12 | 3 | 21 | 25 | 3 | 9 | 8 | 18 | 28 | 5 |



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Start | 15 | 12 | 3 | 21 | 25 | 3 | 9 | 8 | 18 | 28 | 5 |
| | | ↑ $i$ | | | | | | | | | ↑ $j$ |
| *seek* | 15 | 12 | 3 | 21 | 25 | 3 | 9 | 8 | 18 | 28 | 5 |
| | | | | ↑ $i$ | | | | | | | ↑ $j$ |
| *swap* | 15 | 12 | 3 | 5 | 25 | 3 | 9 | 8 | 18 | 28 | 21 |
| | | | | ↑ $i$ | | | | | | | ↑ $j$ |
| *seek* | 15 | 12 | 3 | 5 | 25 | 3 | 9 | 8 | 18 | 28 | 21 |
| | | | | | ↑ $i$ | | | ↑ $j$ | | | |
| *swap* | 15 | 12 | 3 | 5 | 8 | 3 | 9 | 25 | 18 | 28 | 21 |
| | | | | | ↑ $i$ | | | ↑ $j$ | | | |
| *seek* | 15 | 12 | 3 | 5 | 8 | 3 | 9 | 25 | 18 | 28 | 21 |
| | | | | | | | ↑ $j$ | ↑ $i$ | | | |
| *finish* | 9 | 12 | 3 | 5 | 8 | 3 | 15 | 25 | 18 | 28 | 21 |
| | | | | | | | ↑ $j$ | ↑ $i$ | | | |

The pivot is in the correct position

## Pseudocode of partitioning procedure (Hoare’s algorithm )



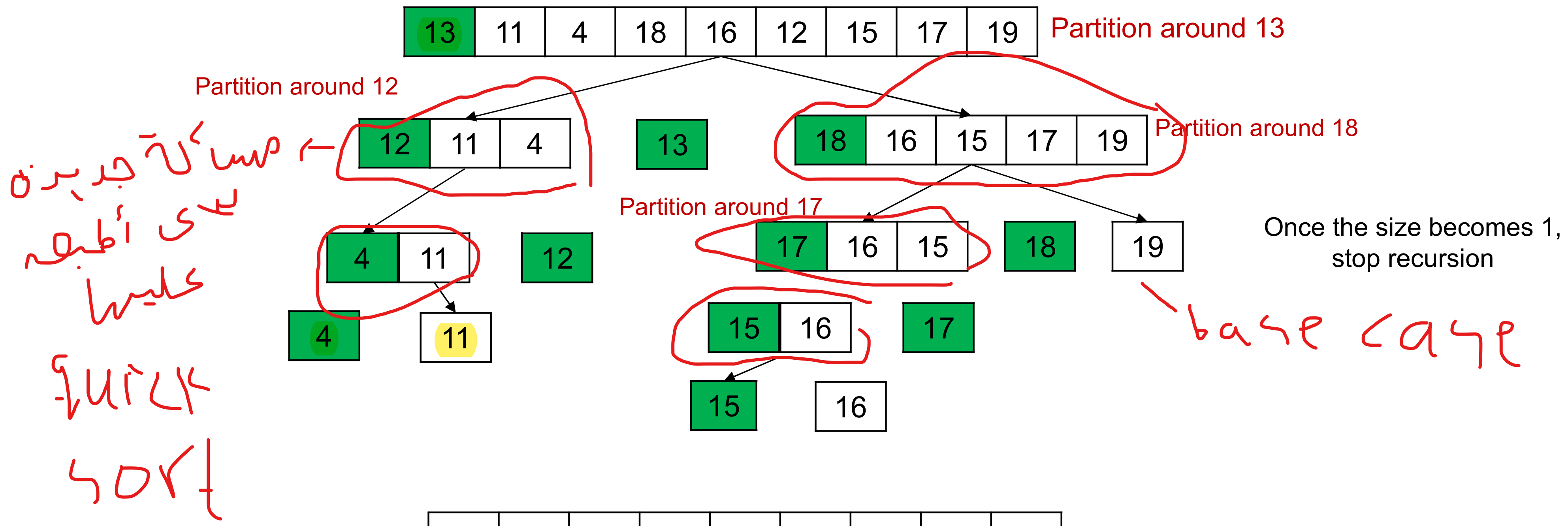
```
ALGORITHM  HoarePartition($A[l..r]$)
    //Partitions a subarray by Hoare’s algorithm, using the first element
    //        as a pivot
    //Input: Subarray of array $A[0..n-1]$, defined by its left and right
    //        indices $l$ and $r$ $(l < r)$
    //Output: Partition of $A[l..r]$, with the split position returned as
    //        this function’s value
    $p \leftarrow A[l]$
    $i \leftarrow l$; $j \leftarrow r+1$
    repeat
        repeat $i \leftarrow i+1$ until $A[i] \geq p$
        repeat $j \leftarrow j-1$ until $A[j] \leq p$
        swap($A[i]$, $A[j]$)
    until $i \geq j$
    swap($A[i]$, $A[j]$)   //undo last swap when $i \geq j$
    swap($A[l]$, $A[j]$)
    return $j$
```

Note that index *i* can go out of the subarray’s bounds in this pseudocode.

Rather than checking for this possibility every time index *i* is incremented, we can append to array *A*[0..*n* - 1] a “sentinel” that would prevent index *i* from advancing beyond position *n*

# Quick sort Examples

Given the following list: 13 , 11 , 4 , 18 , 16 , 12 , 15 , 17 , 19
Draw the tree generated by applying **Quick Sort**. Pick the first element as a pivot

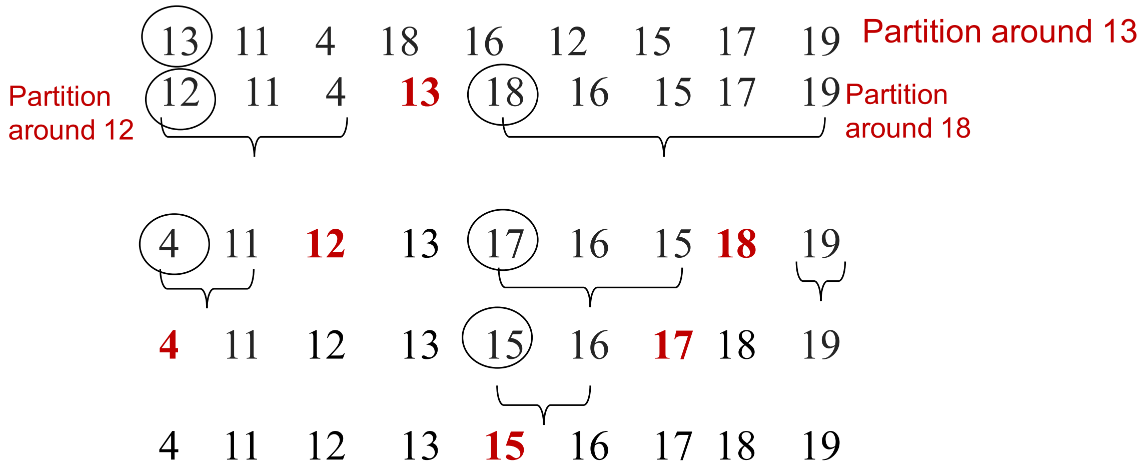


**Output**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 11 | 12 | 13 | 15 | 16 | 17 | 18 | 19 |

# Quick sort Examples

Apply merge sort on the following list: 13 , 11 , 4 , 18 , 16 , 12 , 15 , 17 , 19



Note that in quick sort we can sort "in place" (without using extra space)

## Notes on Quick sort

- On randomly ordered arrays which are the most common, on average the quick sort is more efficient (run faster) than merge sort and heapsort (both are n log n algorithms).

This certainly justifies the name (quick) given to the algorithm by its inventor

- Because of quicksort’s importance, there have been persistent efforts over the years to improve the basic algorithm:
  - Better pivot selection methods such as **randomized quicksort** that uses a random element or the ***median-of-three method*** that uses the median of the leftmost, rightmost, and the middle element of the array.

## Improving Quicksort

**How can you make it better?**

Choose a pivot that balances the sub problems more evenly.

**Median-of-Three** Or **random partiton**

## Discussion

**Is QuickSort in-place?**

Quick Sort is an 'in place' like insertion sort, but not like merge sort

**Is QuickSort stable?**

unstable

## Pair Activity

Implement quicksort iteratively

## Search Topics

- 3-Way quick sort
- Comparison of pivot selection methods

## Pair Activity

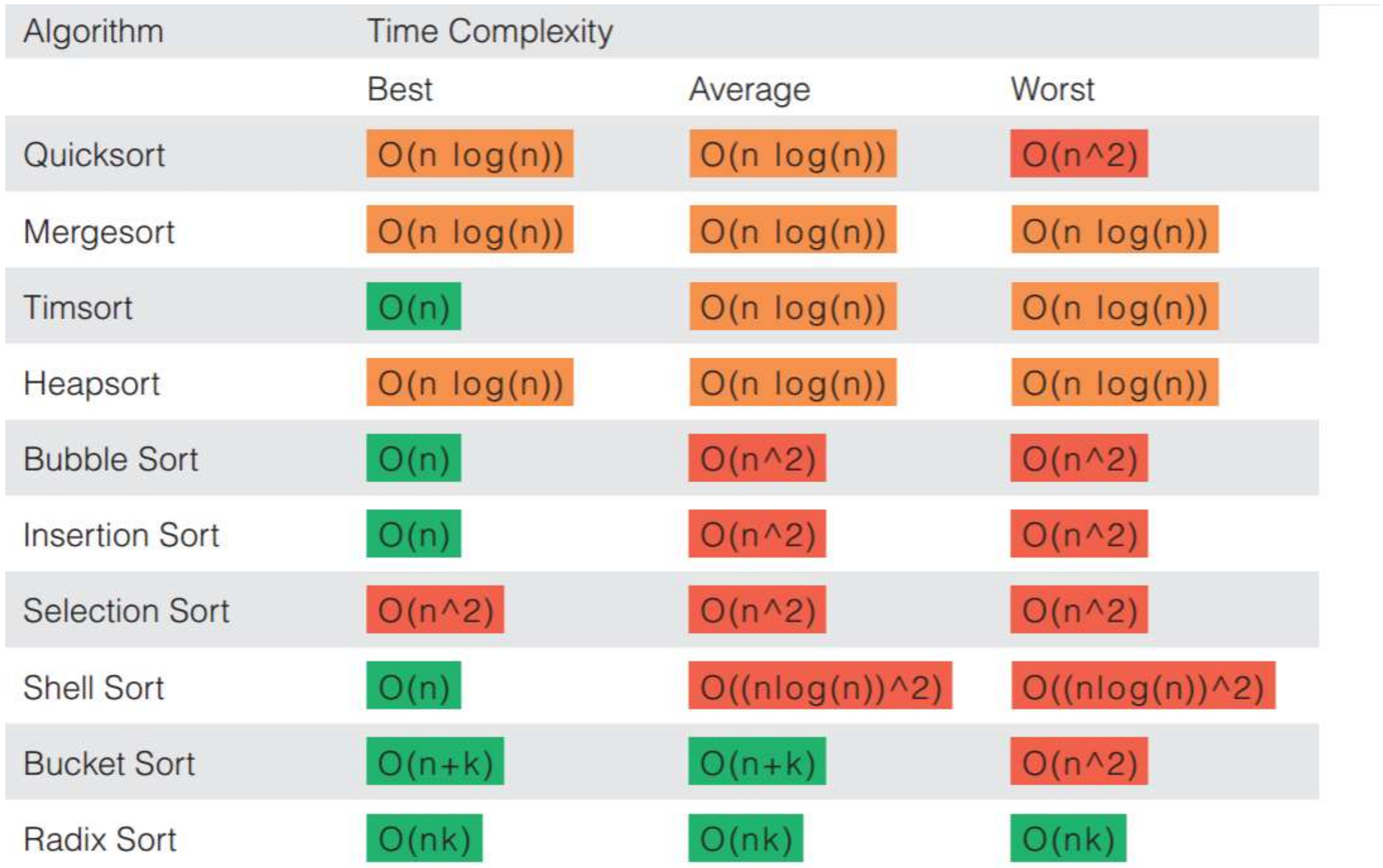
## Problem: inversions count in an array

# Sorting Algorithms

| Algorithm | Worst-case running time | Best-case running time | Average-case running time |
|---|---|---|---|
| Selection sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Insertion sort | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n^2)$ |
| Merge sort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ |
| Quicksort | $\Theta(n^2)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ |
| *Bubble sort* | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n^2)$ |

ما أخذناها

# Sorting Algorithms



| Algorithm | Time Complexity | | |
|---|---|---|---|
| | Best | Average | Worst |
| Quicksort | O(n log(n)) | O(n log(n)) | O(n^2) |
| Mergesort | O(n log(n)) | O(n log(n)) | O(n log(n)) |
| Timsort | O(n) | O(n log(n)) | O(n log(n)) |
| Heapsort | O(n log(n)) | O(n log(n)) | O(n log(n)) |
| Bubble Sort | O(n) | O(n^2) | O(n^2) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) |
| Shell Sort | O(n) | O((nlog(n))^2) | O((nlog(n))^2) |
| Bucket Sort | O(n+k) | O(n+k) | O(n^2) |
| Radix Sort | O(nk) | O(nk) | O(nk) |

## Exercises

1) Apply quick sort to sort the list: 50, 23, 9, 18, 61, 32

2) Give an example of an array of *n* elements for which the sentinel mentioned in the text is actually needed. What should be its value?

3) For the discussed version of quicksort:

a. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

b. Are arrays made up of all equal elements the worst-case input, the best case input, or neither?