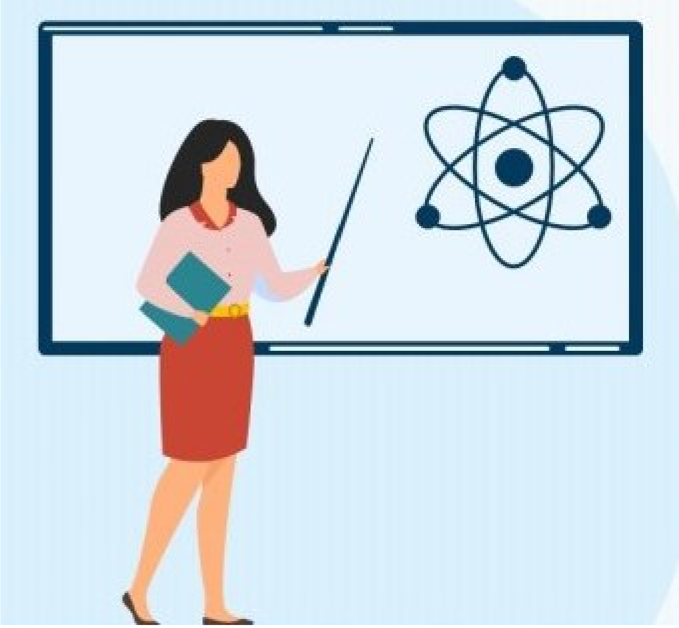
Selection Sort & Quick Sort



Suman Pandey

Selection Sort

Idea behind Selection Sort

Set the first element as minimum

1	2	3	4	5	6	7
6	3	5	4	2	1	9

Compare minimum with the second element, if the second element is smaller than minimum, assign the second element as minimum.

1	2	3	4	5	6	7
6	3	5	4	2	1	9

3 becomes min

1	2	3	4	5	6	7
6	3	5	4	2	1	9

2 becomes min

1	2	3	4	5	6	7
6	3	5	4	2	1	9

1 becomes min
Swap minimum to first
Element at the end of
iteration

1	2	3	4	5	6	7
1	3	5	4	2	6	9

Assign 3 as min

1	2	3	4	5	6	7
1	3	5	4	2	6	9

Find the next minimum
Which is 2
Swap minimum to
Second element

1	2	3	4	5	6	7
1	2	5	4	3	6	9

⋮

Divide-and-Conquer (Quick Sort)

n=8

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

Alg.:SELECTIONSORT(A, **n**)

```
for i=0 to n-2
    min = i
    for j = i +1 to n-1
        if A[ j ]< A[ min ] then
            min = j
    end for
    if min != i then interchange A[ i ] and A[ min ]
end for
```

Time Complexity is $O(n^2)$

Python code

```
# Selection sort in Python


def selectionSort(array, size):

    for step in range(size):
        min_idx = step

        for i in range(step + 1, size):

            # to sort in descending order, change > to < in this line
            # select the minimum element in each loop
            if array[i] < array[min_idx]:
                min_idx = i

        # put min at the correct position
        (array[step], array[min_idx]) = (array[min_idx], array[step])


data = [-2, 45, 0, 11, -9]
size = len(data)
selectionSort(data, size)
print('Sorted Array in Ascending Order:')
print(data)
```

Quick Sort

Idea behind Quick Sort

Can you tell me which element in the list below are at the sorted position

1	2	3	4	5	6	7	8
6	3	5	4	2	1	9	

ans = 9

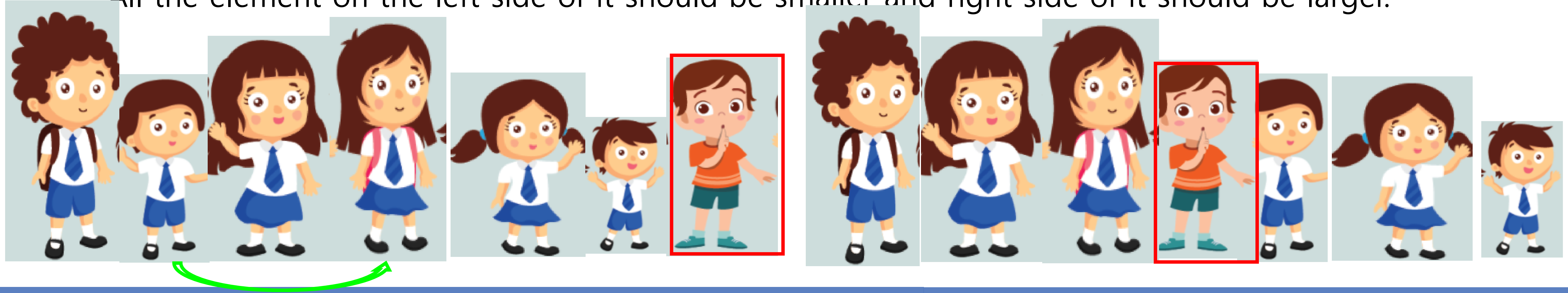
1	2	3	4	5	6	7	8
4	6	7	10	12	13	14	

ans = 10

Because all the element left of the element is smaller to it and all the element to the right of it is bigger

Quick Sort works on this idea.

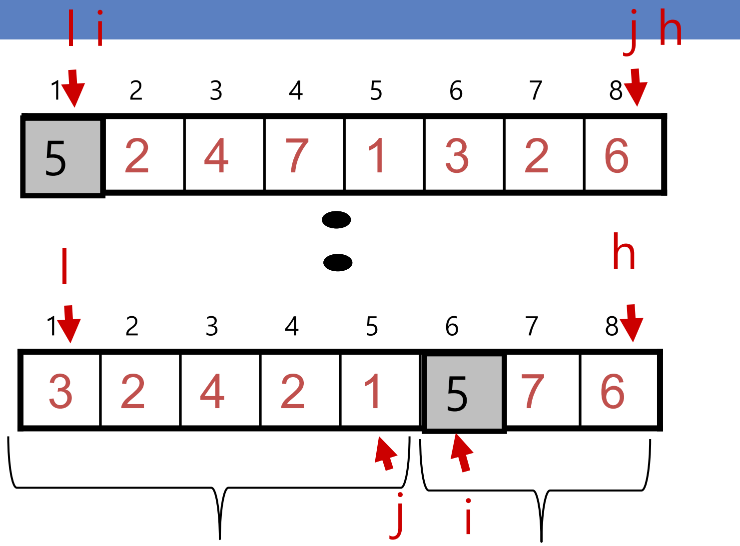
All the element on the left side of it should be smaller and right side of it should be larger.



Divide-and-Conquer (Quick Sort)

Alg.:PARTITION(*A*, l, h)

```
pivot = A[ l ]
i=l              j=h
while ( i < j )
      do
         i++
      while ( A[i] <= pivot)
      do
         j- -
      while ( A[j] > pivot )
      if ( i < j)
         Swap ( A[ i ] , A[ j ] )
 Swap ( A[ l ], A[ j ]
 return j
```



Now do the same task **recursively**

Alg.:QUICKSORT(*A*, l, h)

```
if ( l < h )
  j = PARTITION(A, l , h)
  QUICKSORT(A, l, j)
  QUICKSORT(A, j+1, h)
```

Partition

l h

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

l – low, beginning of the list
h – high, its end of the list

Lets take the first element as pivot pivot = 5

Now 5 should be placed in a way so that
all the element lesser than 5 comes to the left of it
all the element greater than 5 come to the right of it

l h

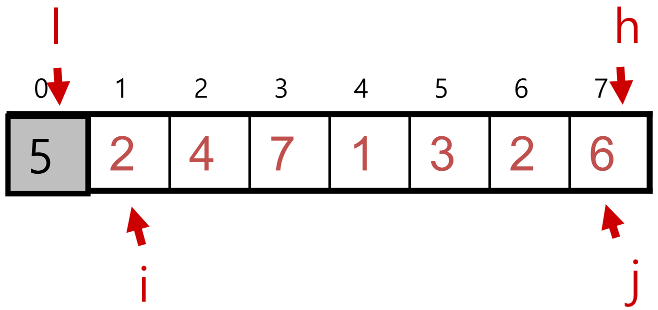
1	2	3	4	5	6	7	8
5	2	4	7	1	3	2	6

i j

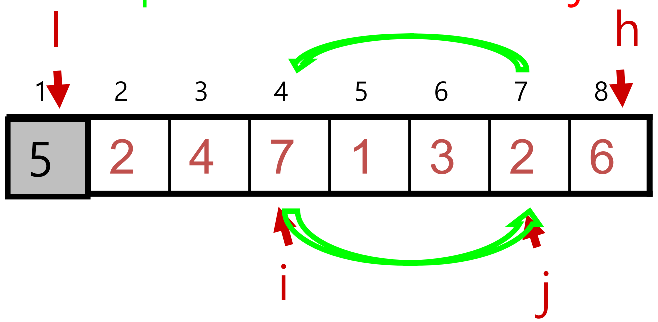
i will search the element smaller than pivot

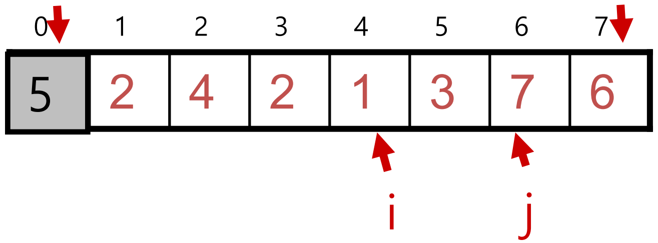
j will search the element greater than pivot

Partition



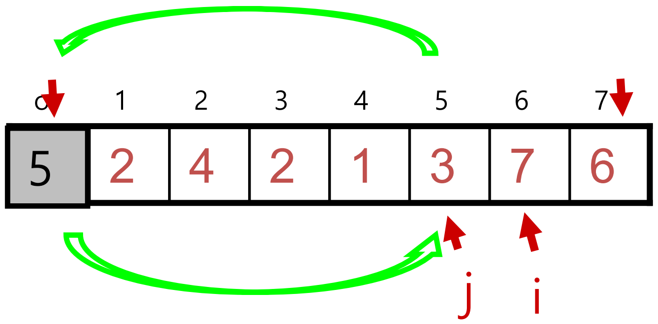
Increment i
2 > 5 is false – increase i
4 > 5 is false – increase i
7 > 5 is true – stop here
Decrement j
6 < 5 is false – decrement j
2 < 5 is true – stop here
Swap element at i with j





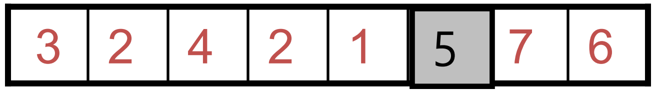
Increment i
3 > 5 is false – increase i
7 > 5 is true – stop here
Decrement j
3 < 5 is true – stop here

Increment i element[i] > pivot
Decrement j element[j]< pivot
Swap the elements at i and j



Stop the loop if i > j

Note: at this point don't interchange
At this point we found the position of pivot (j)
Swap 5 with 3
5 is now at its sorted position



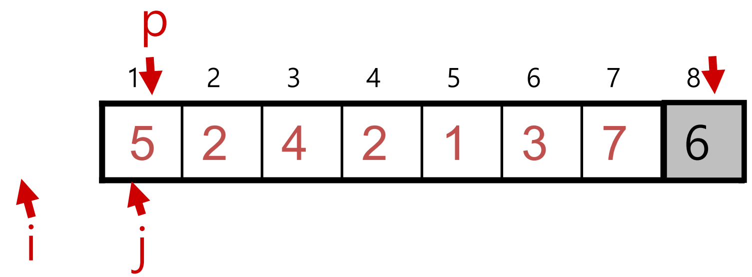
Divide-and-Conquer (Quick Sort improved)

*Alg.:*PARTITION(*A*, p, r)

```
pivot = A[ r ]
i=p-1
for j=p to r-1
    if ( A[j] <= pivot)
        i++
        Swap ( A[ i ] , A[ j ] )
Swap ( A[ i+1 ], A[ r ] )
return i+1
```

First i=-1

Previous 2 while loops can be combined with this approach in one loop



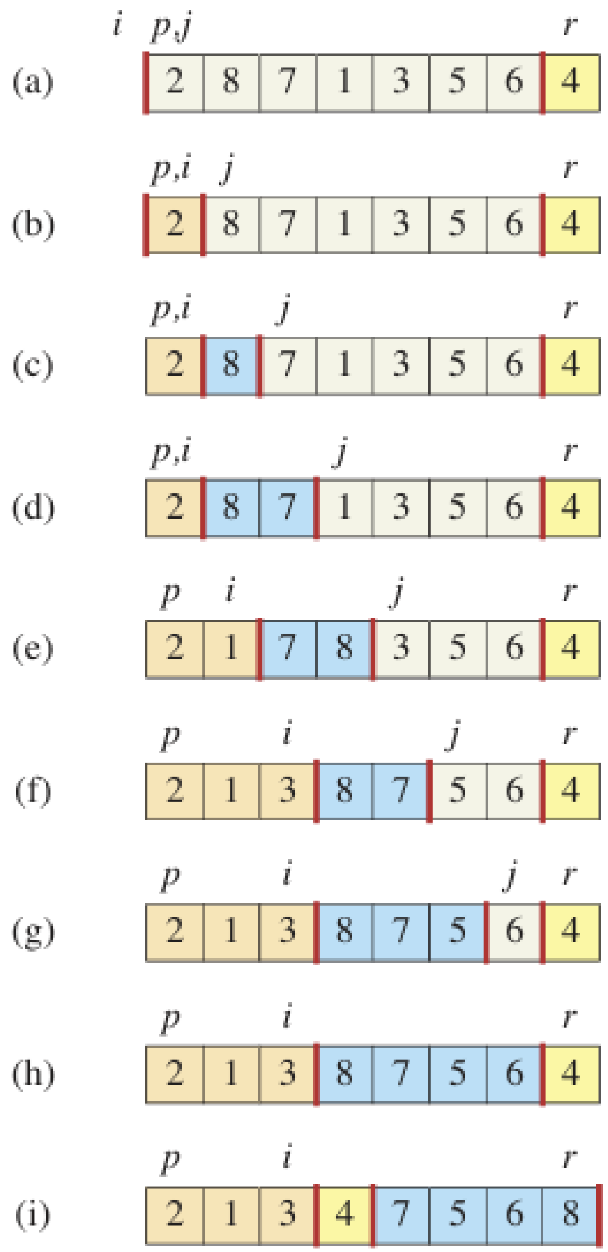
Now do the same task **recursively**

*Alg.:*QUICKSORT(*A*, p, r)

```
if ( l < h )
    j = PARTITION(A, l , h)
    QUICKSORT(A, l, j-1)
    QUICKSORT(A, j+1, h)
```

*Alg.:*PARTITION(*A*, p, r)

```
pivot = A[ r ]
i=p-1
for j=p to r-1          # process each element other than pivot
    if ( A[j] <= pivot)   # check if element belong to the lower side
        i++             # index of the new slot in low side
        Swap ( A[ i ] , A[ j ] )
Swap ( A[ i+1 ], A[ r ] )
return i+1
```



Python Code (Quick Sort)

```
def partition(arr, low, high):
        i = (low-1)                 # index of smaller element
        pivot = arr[high]           # pivot

        for j in range(low, high):

                # If current element is smaller than or
                # equal to pivot
                if arr[j] <= pivot:

                        # increment index of smaller element
                        i = i+1
                        arr[i], arr[j] = arr[j], arr[i]

        arr[i+1], arr[high] = arr[high], arr[i+1]
        return (i+1)
```

```
def quickSort(arr, low, high):
        if len(arr) == 1:
                return arr
        if low < high:

                # pi is partitioning index, arr[p] is now
                # at right place
                pi = partition(arr, low, high)

                # Separately sort elements before
                # partition and after partition
                quickSort(arr, low, pi-1)
                quickSort(arr, pi+1, high)


# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr, 0, n-1)
print("Sorted array is:")
for i in range(n):
        print("%d" % arr[i]),
```

Big O Analysis (Quick Sort)

- **Best case - O(n log n)**
- Cost of partitioning : O(n)
- Have to quicksort Log(n) splits $n = 2^k$ $8=2^k \rightarrow k = \log_2 8$

  $k = \log(n)$
- Quick sort runs : **O(n log n)**
- This is when the partitioning is done always in the middle
- That is if the pivot element is a median of the list.

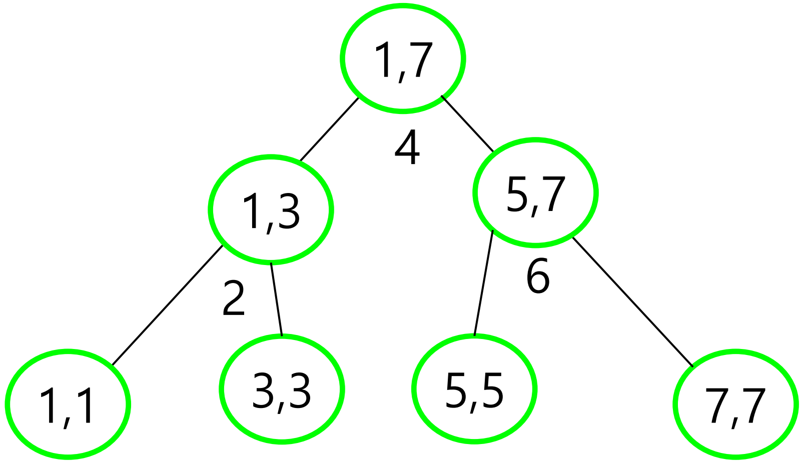
- **Worst case - O(n^2)**
- This will happen when elements are already sorted

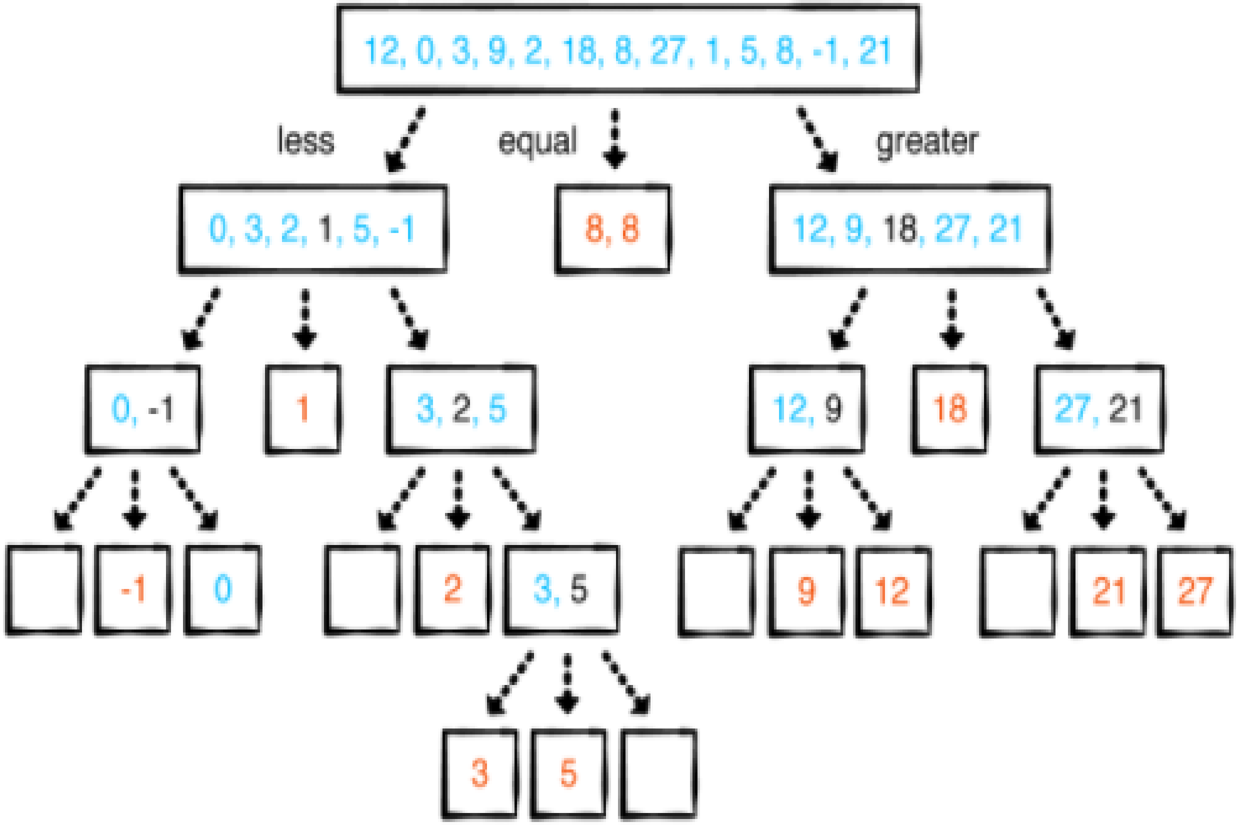
1	2	3	4	5	6	7	8
2	4	8	10	16	17	18	19
i							j

- i will stop at 2nd index and j will stop at 1st index, partitioning will happen at 1st index
- So always the partitioning will happen at the beginning of the list.

Big O analysis Best Case (Quick Sort)

Elements at 1 ....7 index



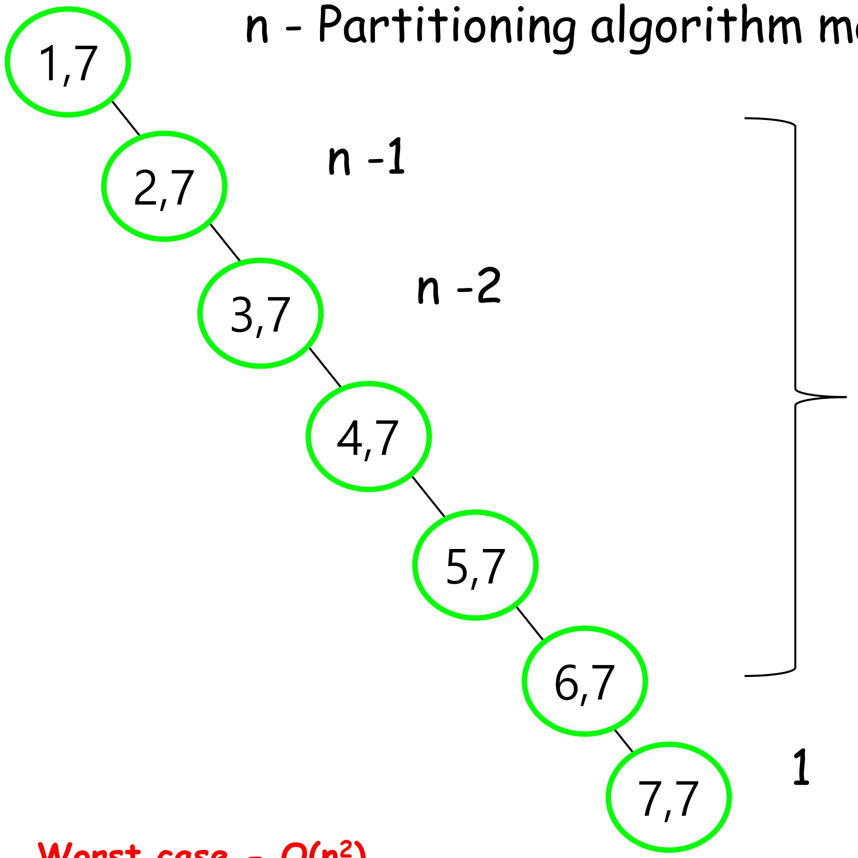


Best Case - Quick sort runs : **O(n log n)**
This is when the partitioning is done always in the middle
That is if the pivot element is a median of the list.
But this is rare situation.

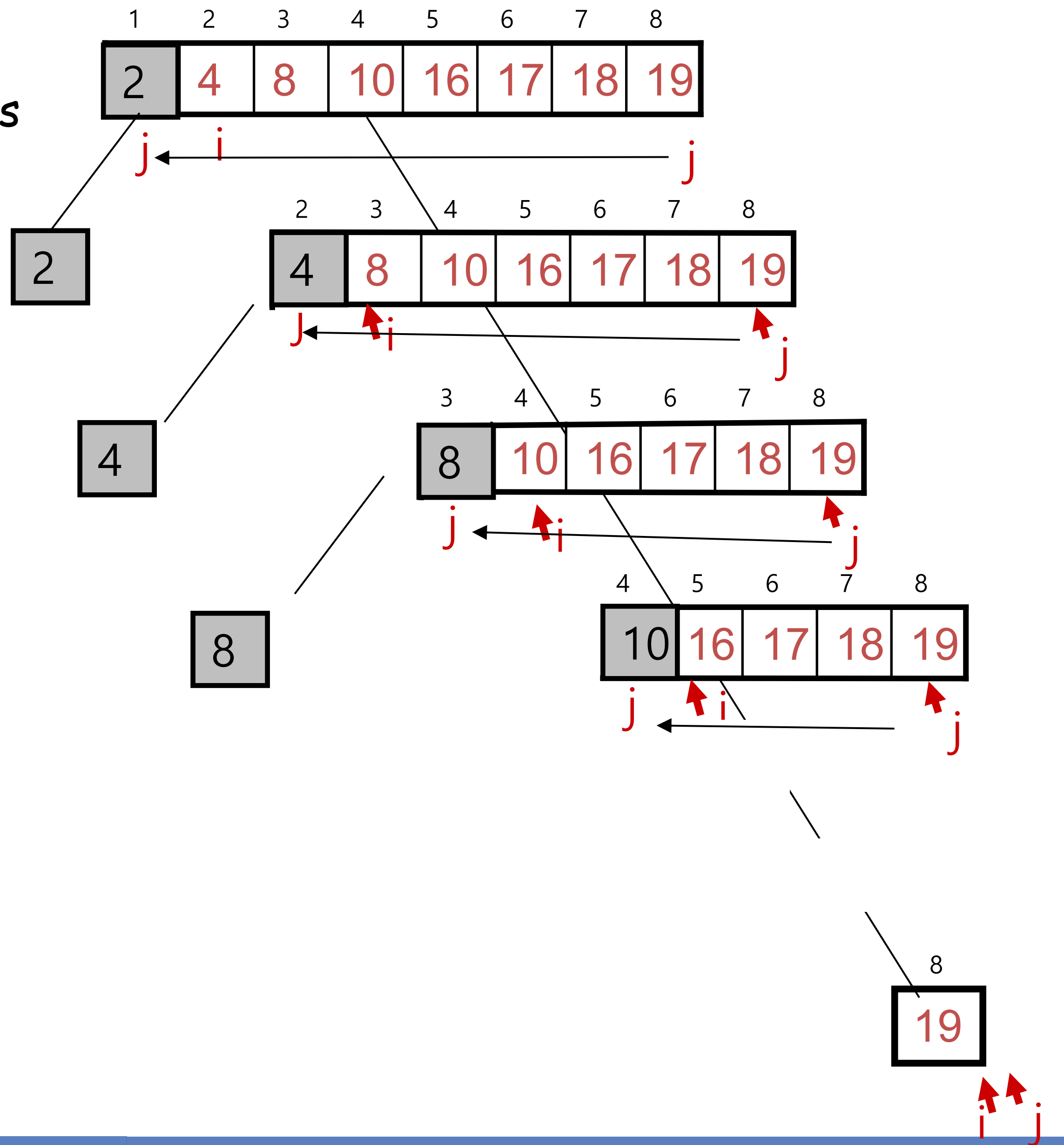
Big O analysis Worst Case (Quick Sort)

Elements at 1 ....7 index

n - Partitioning algorithm makes n comparisons



n(n+1)/2
= O (n^2)



- **Worst case - O(n^2)**
- This will happen when elements are **already sorted**
- i will stop at 2nd index and j will stop at 1st index, partitioning will happen at 1st index
- So always the partitioning will happen at the beginning of the list.

Big O analysis (Quick Sort)

- How to solve the worst case of Quick Sort
  - Don't always select the pivot as first element
  - Select middle element as pivot
  - Select random element as pivot

  - Worst case of Quick sort is O (n2)
  - Best case of Quick sort is O (n log n)