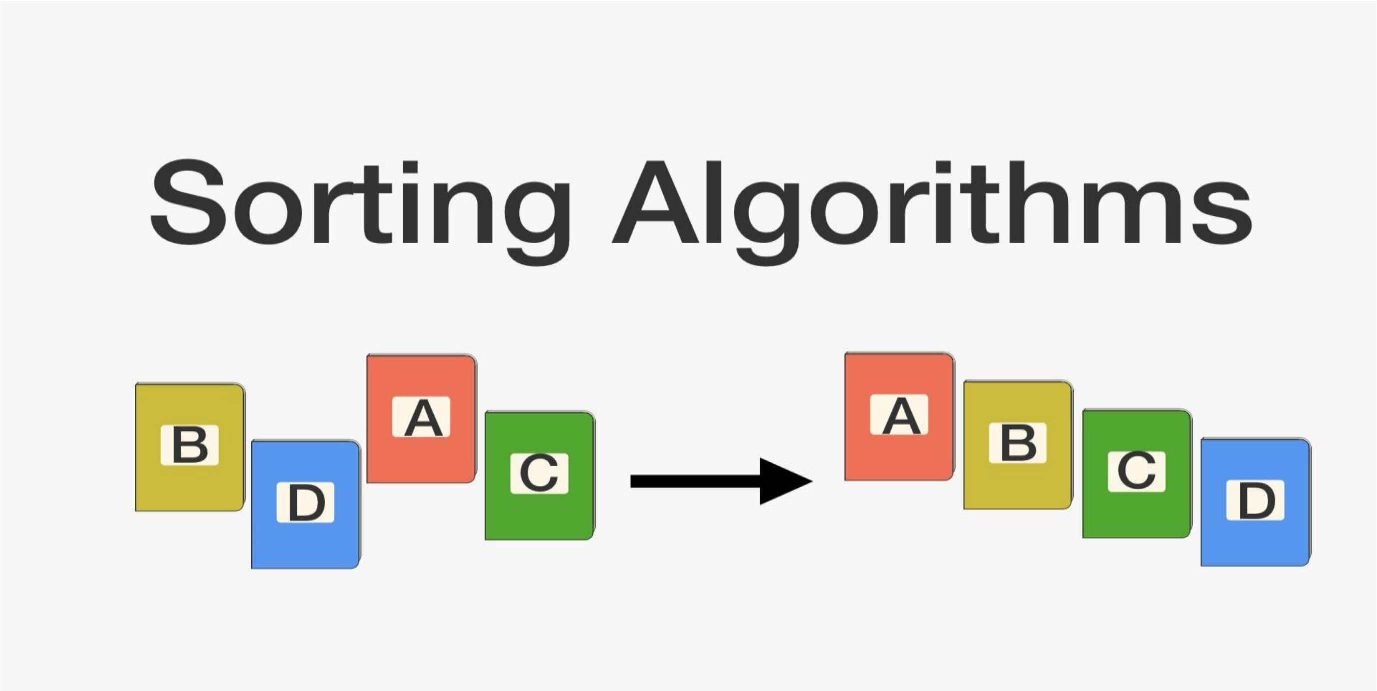
Lets Start

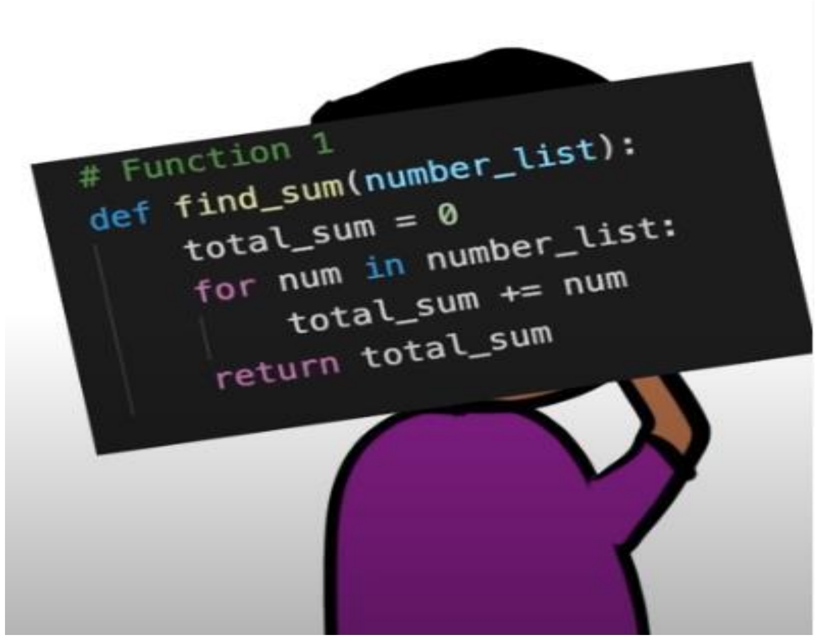


Types of Sorting Algorithm

- Criteria to test an algorithm - No of Operations. – Big 0 notation

Big O

- Big O Notation
  - A mathematical notation used to classify algorithms according to how their **run time** or space requirements grow **as the input size grows**.



1	2	2	3	4	5	6	7

1	2	2	3	4	5	6	7

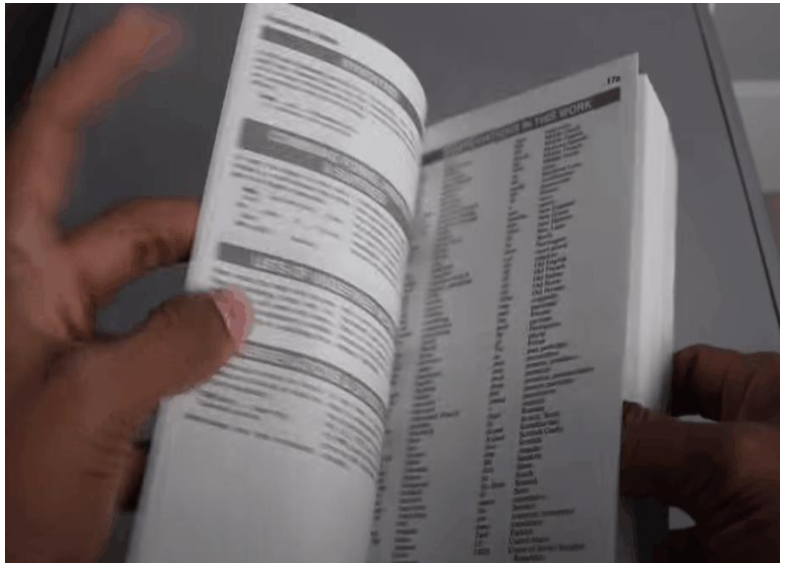
- 1 Operation - **O(1)**

n Operation - **O(n)**

Big 0 Searching Dictionary

- Big 0 Notation

Sequential Search

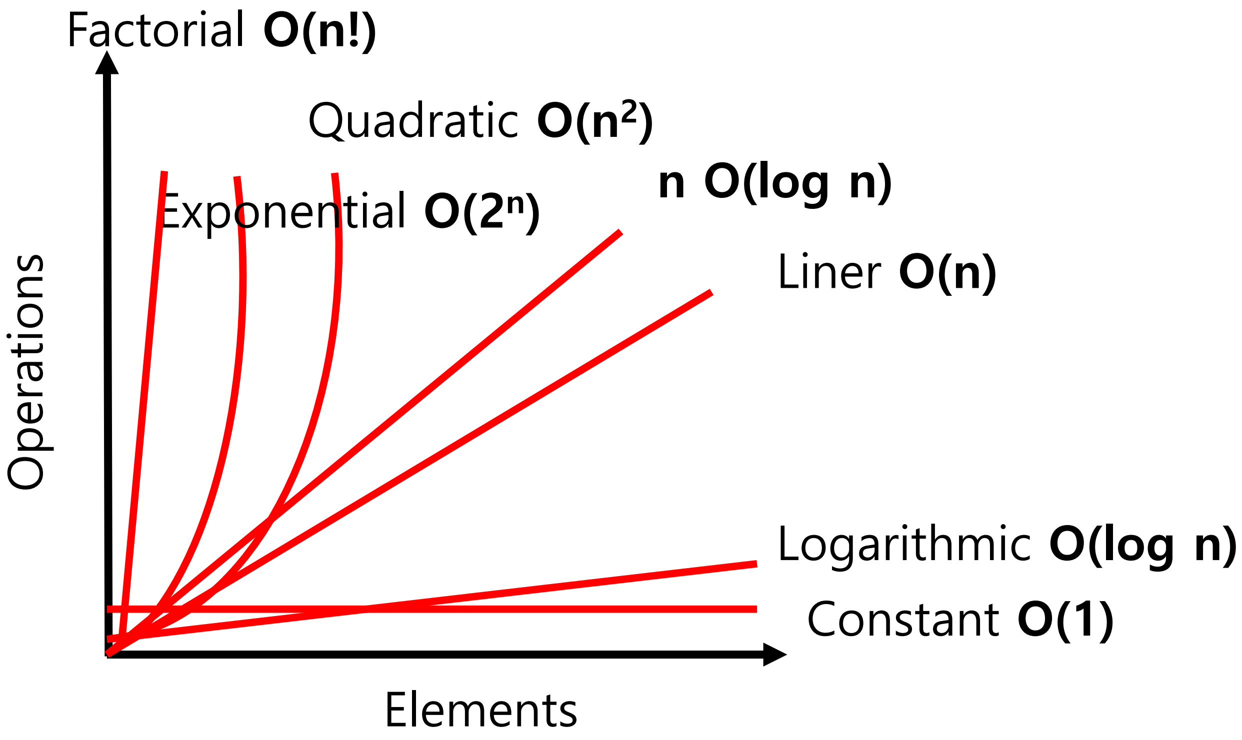


Binary Search

- 1 Operation - **O(n)**

n Operation - **O(log n)**

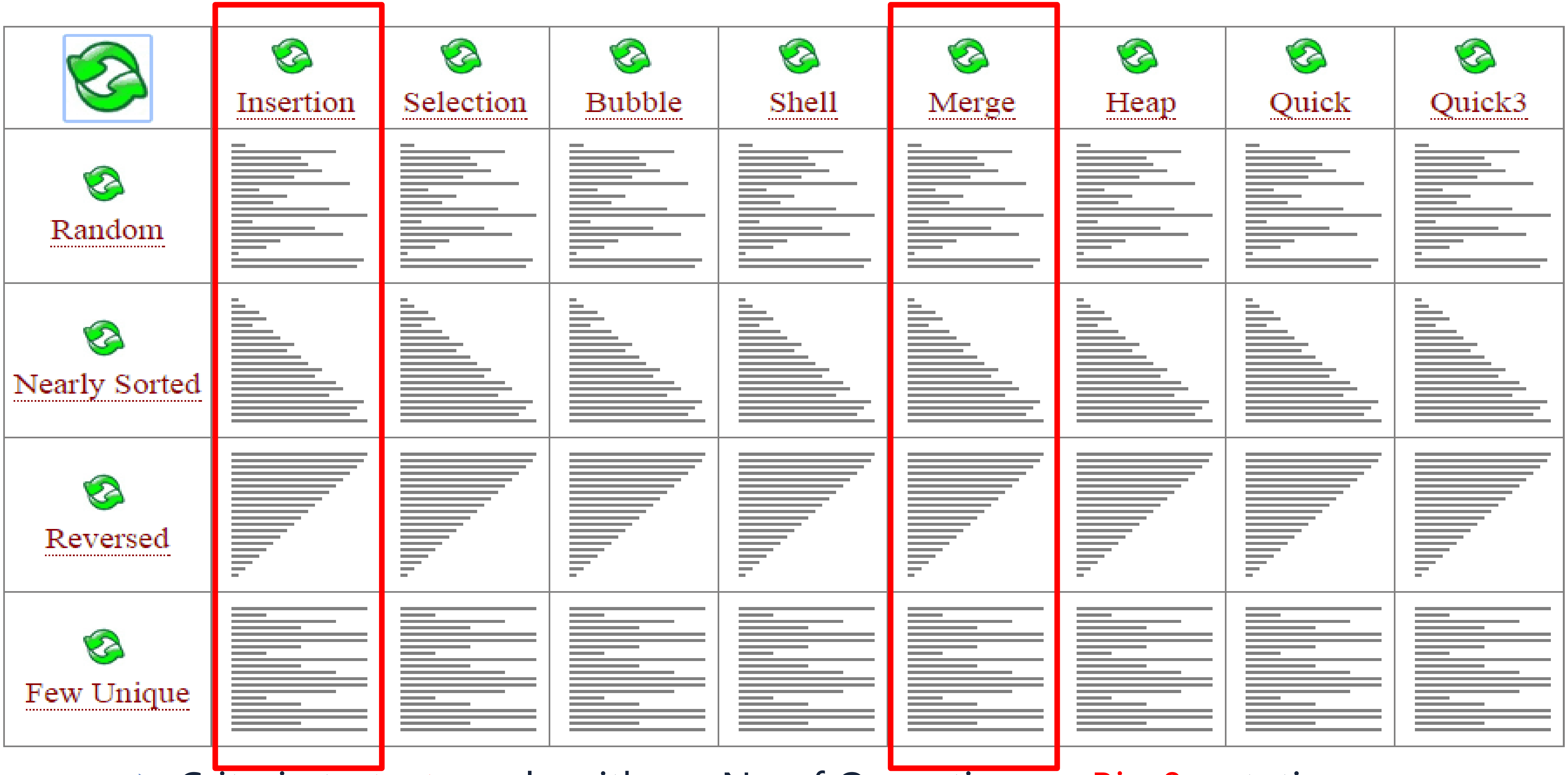
Time Complexities of Big 0



N: 17

O(1):	1
O(Log N):	4
O(N):	17
$O(N^2)$:	289
$O(2^N)$:	131072
O(N!):	3556874280960

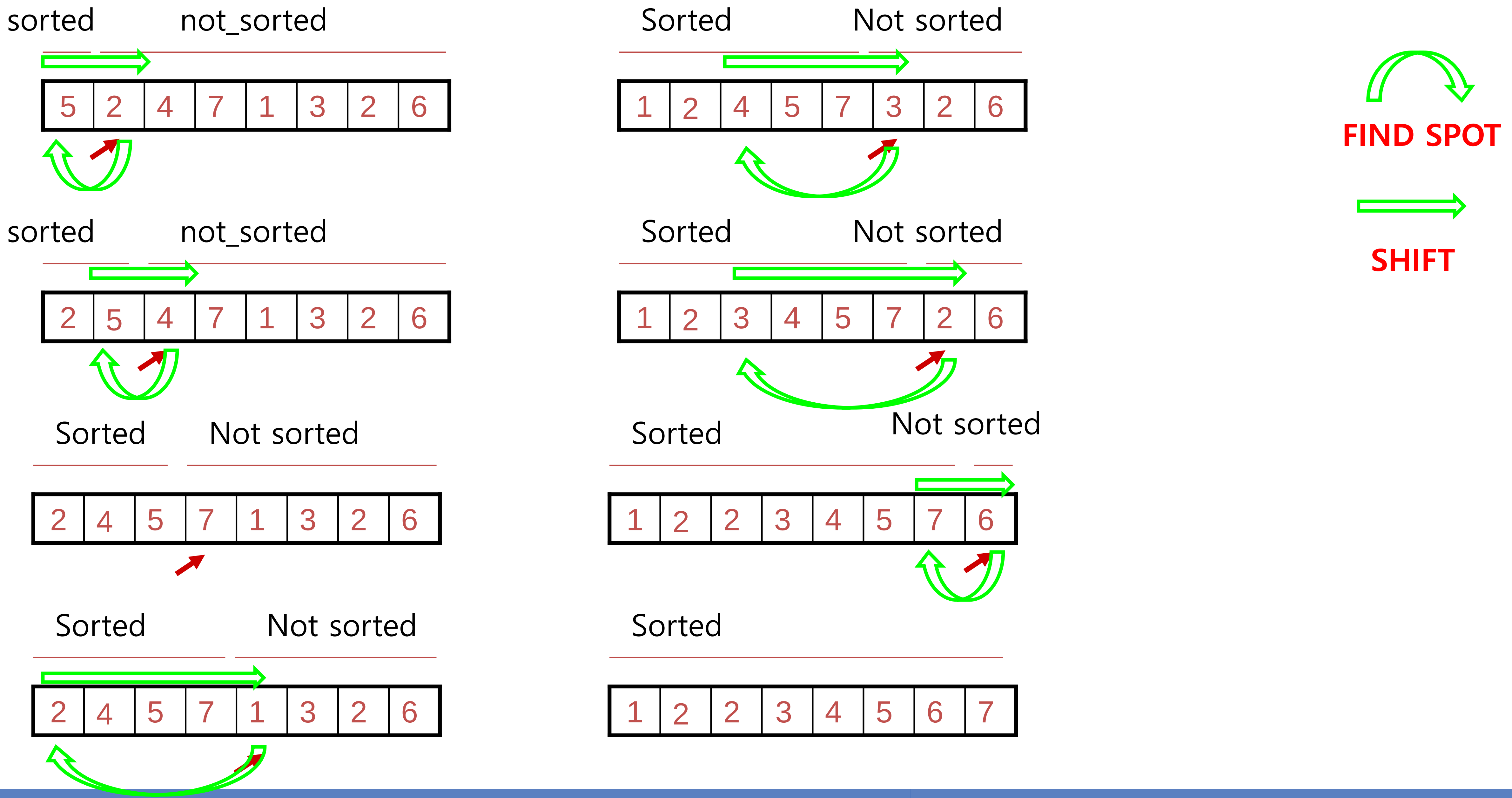
Types of Sorting Algorithm



- Criteria to test an algorithm - No of Operations. – Big 0 notation

Insertion Sort

Insertion Sort



Insertion Sort (pseudocode)

Alg.: INSERTION-SORT(*A*)

declare variables – i, key, j

for i = 1 to n – 1 // outer loop

key = a[i] //pick the next element

j = i – j; // decrement j value

for : (j>=0 && A[j]>key) // inner loop

A[j+1] = A[j]

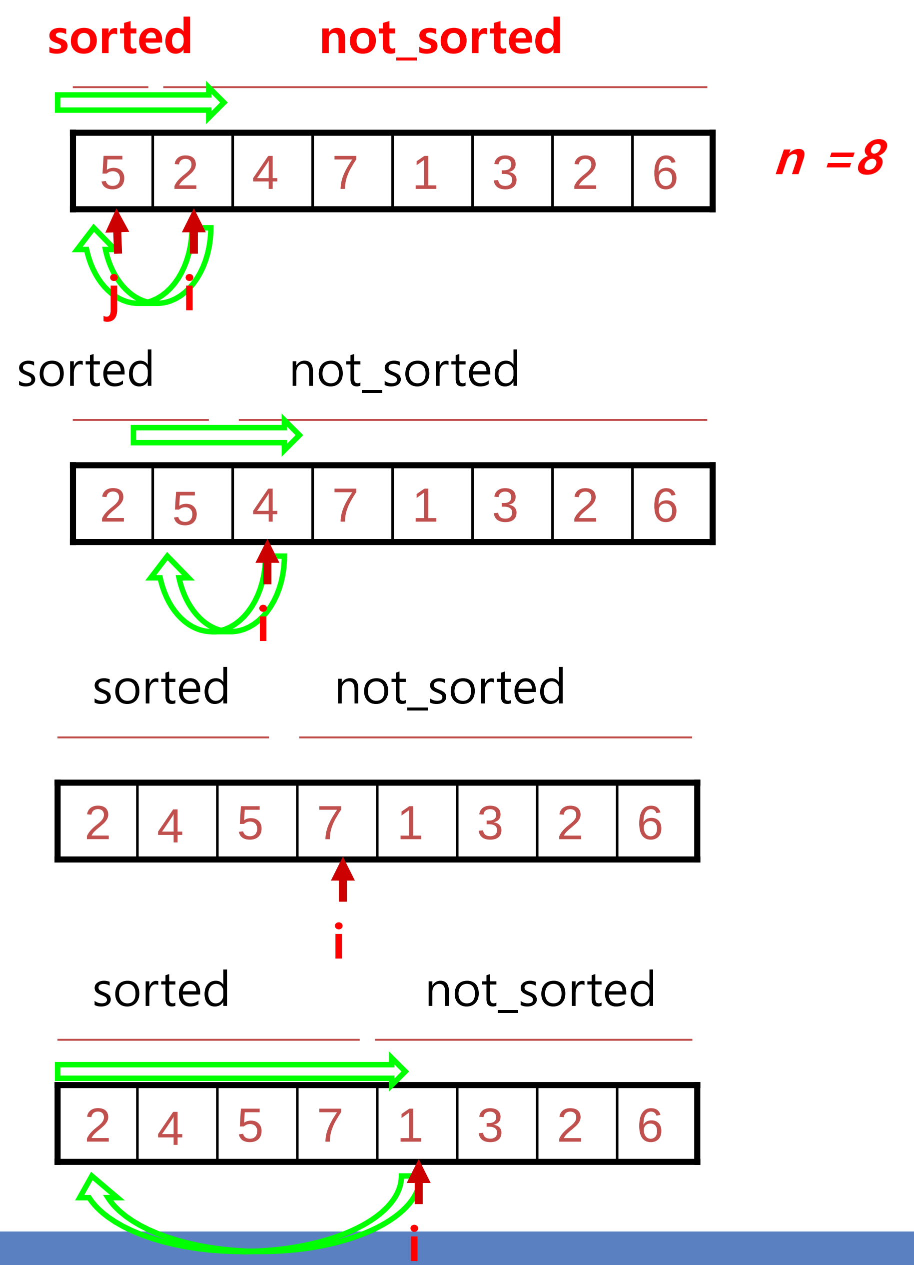
j = j – 1

end loop // outer loop

arr[j+1] = key

end loop // outer loop

$O(n^2)$



- Initial call: INSERTION-SORT(*A*)

Python Code

```
def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key

# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Merge Sort

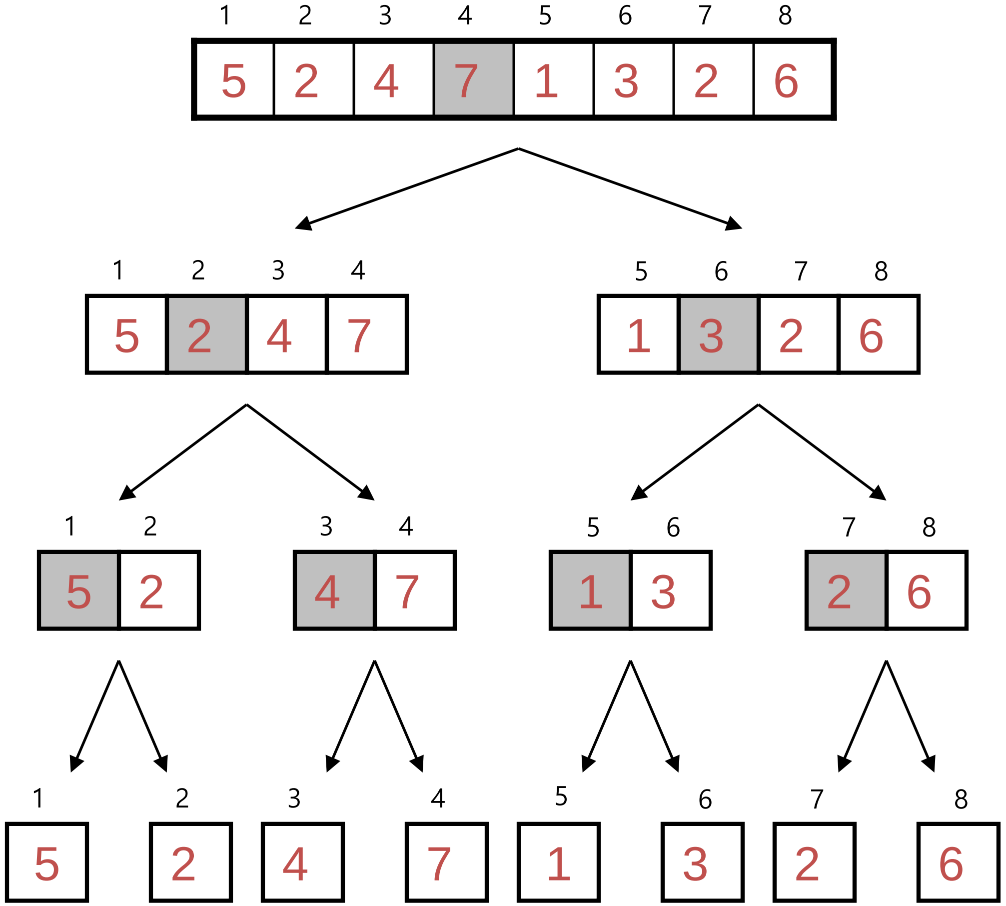
Divide-and-Conquer (Merge Sort)

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems **recursively**
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

Example

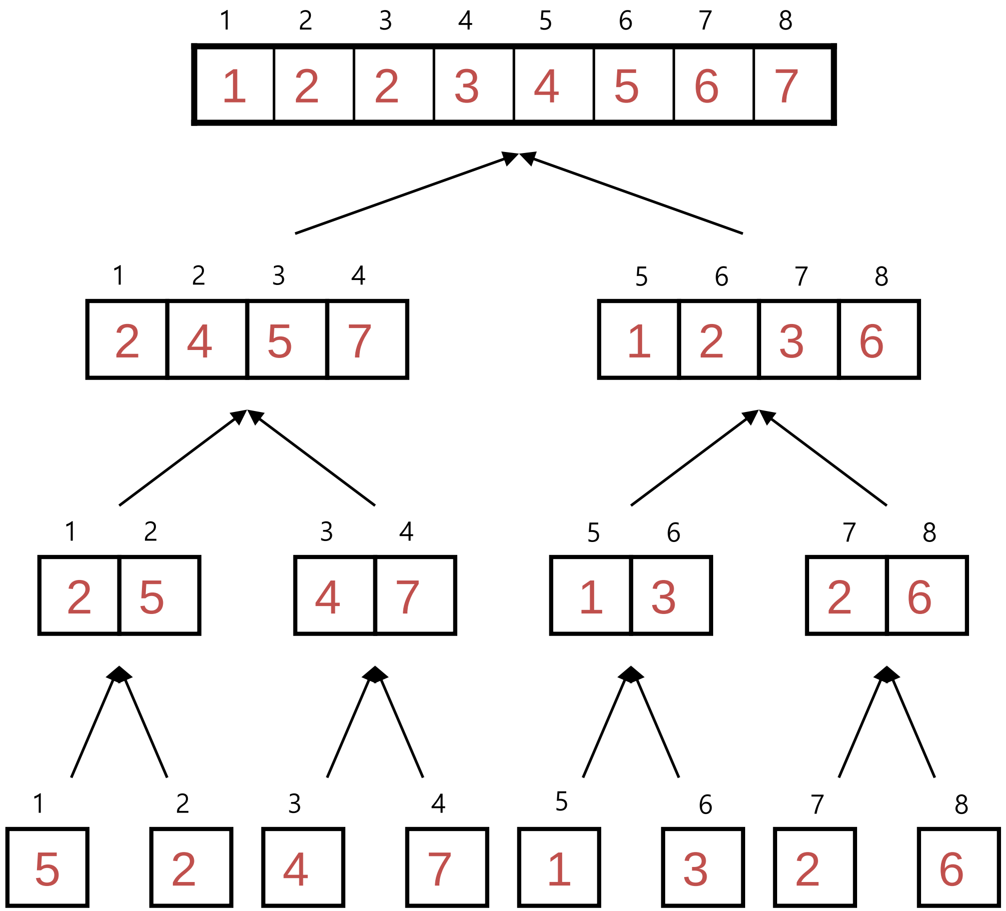
Divide

q = 4



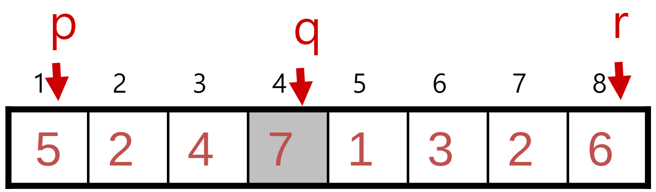
Example

Conquer
and
Merge



Divide-and-Conquer (Merge Sort)

Alg.: MERGE-SORT(A, p, r)



if p < r ▷ Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MERGE-SORT(A, p, q) ▷ Conquer

MERGE-SORT(A, q + 1, r) ▷ Conquer

MERGE(A, p, q, r) ▷ Combine

- Initial call: MERGE-SORT(A, 1, n)

Divide-and-Conquer (Merge Sort)

Alg.: MERGE-SORT(*A*, p, r)

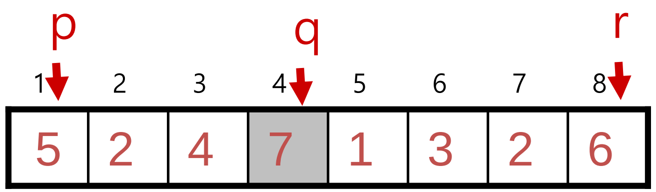
if p < r

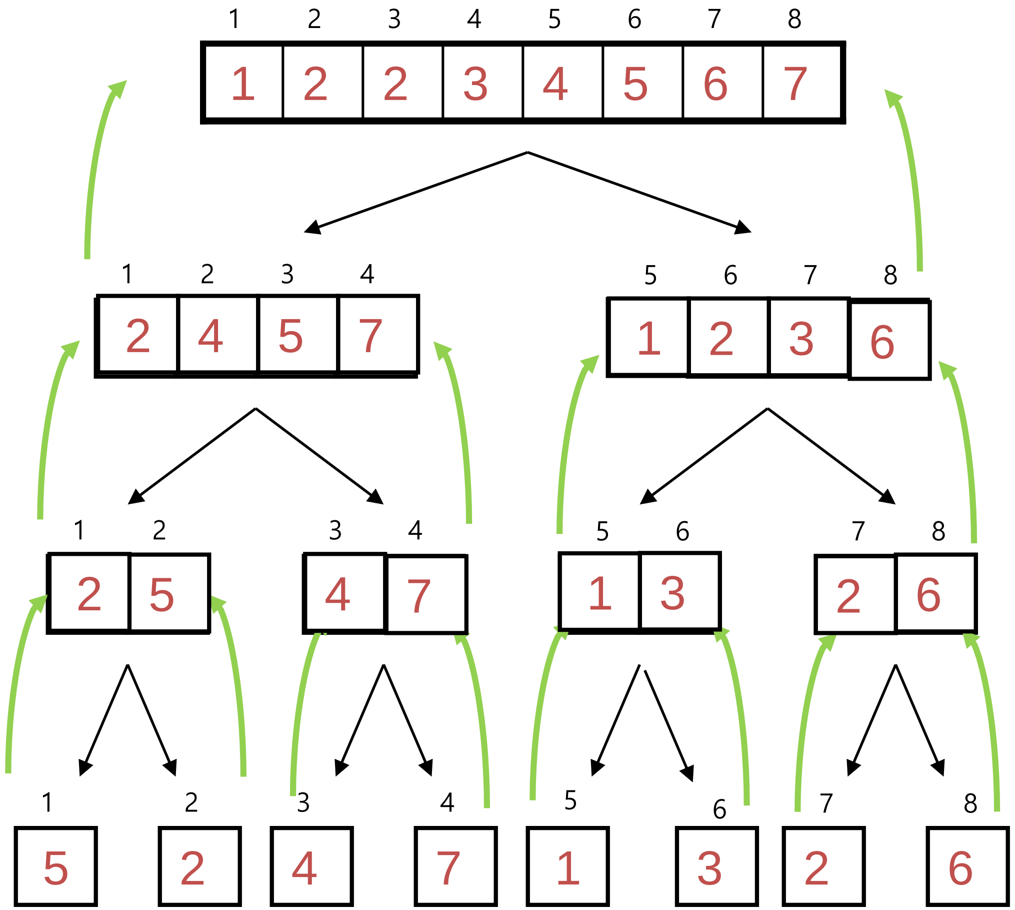
then *q* ← ⌊(p + r)/2⌋

MERGE-SORT(*A*, p, *q*)

MERGE-SORT(*A*, *q* + 1, r)

MERGE(*A*, p, *q*, r)

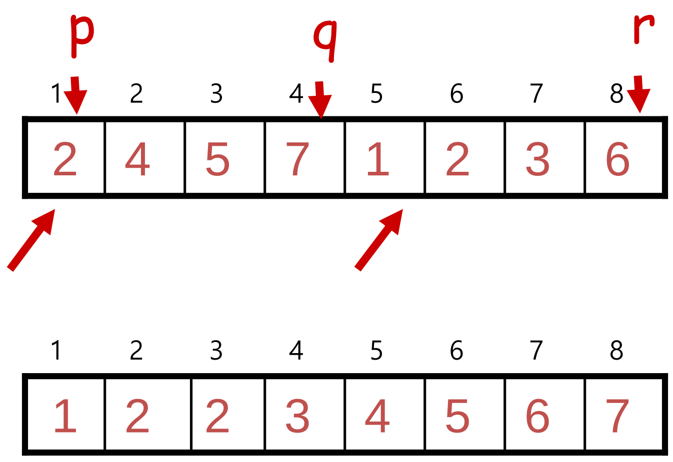




- Initial call: MERGE-SORT(*A*, 1, n)

Merge

- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are sorted
- **Output:** One single sorted subarray $A[p\,..\,r]$

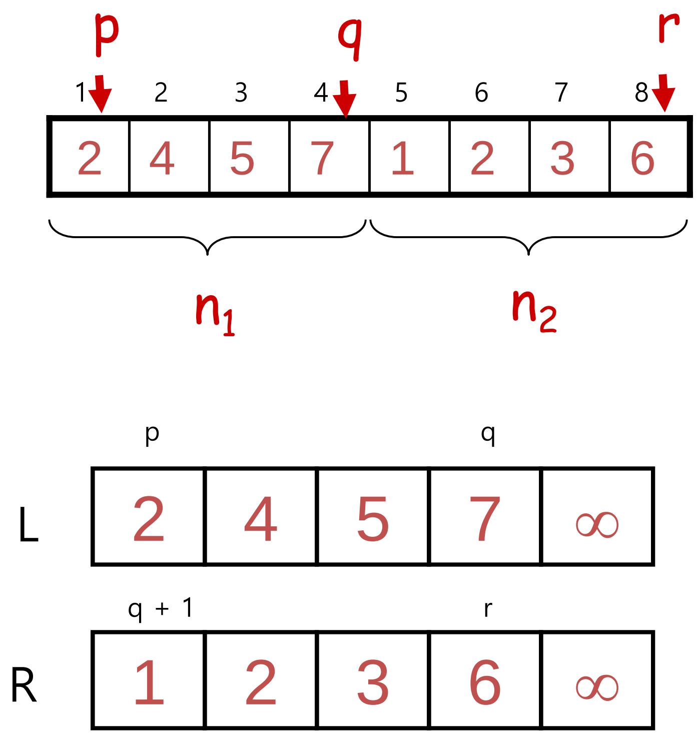


- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

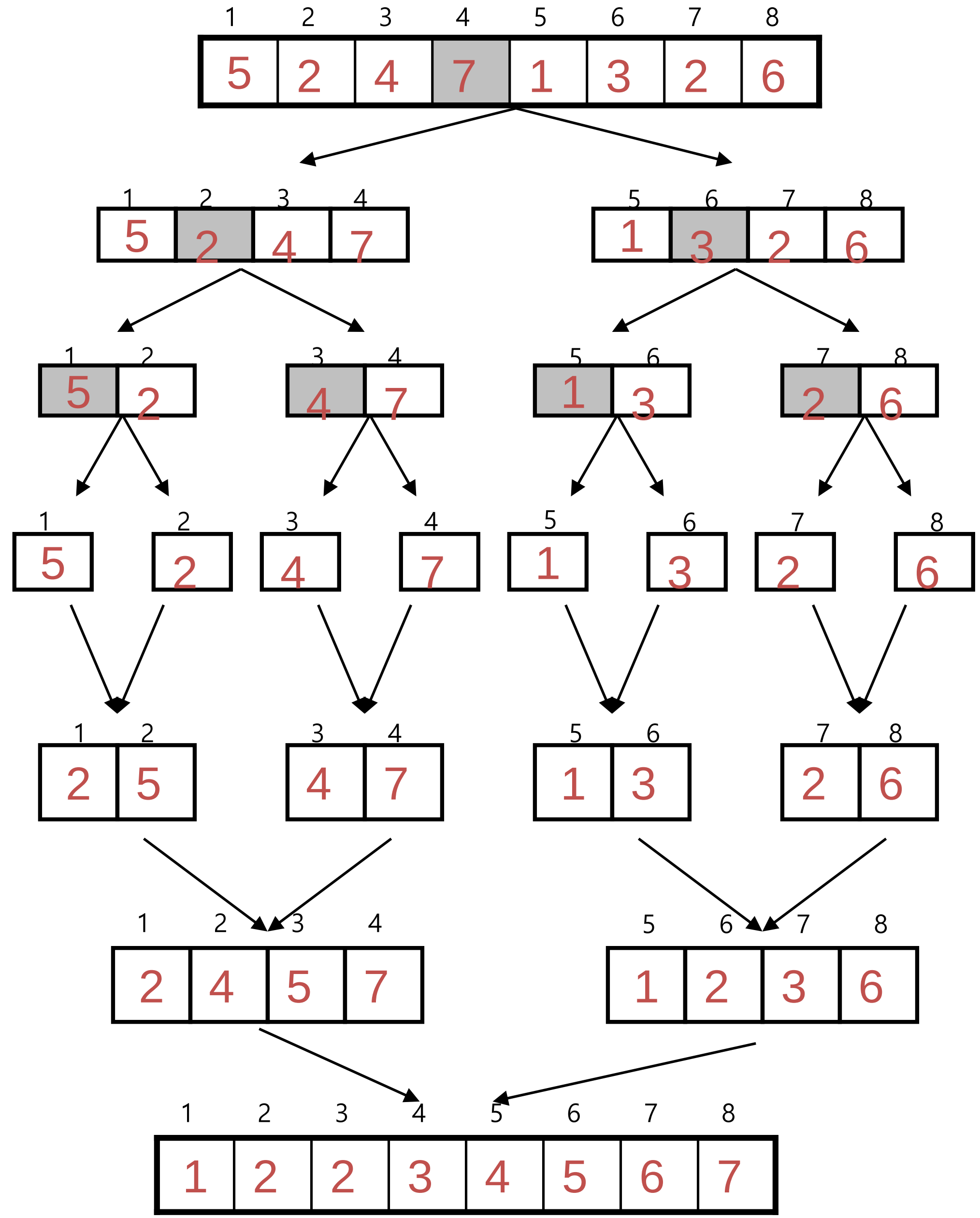
Merge

Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1\ ..\ n_1 + 1]$ and the next n_2 elements into $R[1\ ..\ n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \le R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$



Big O analysis



- Have to merge Log(n) splits
  $n = 2^k$ $8=2^k \rightarrow k = \log_2 8$
  $k = \log(n)$
- Cost of merging two sorted collection to one list : O(n)
- Merge sort runs : **O(n log n)**

- No algorithm can sort an arbitrary collection in better time than this.

```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)// 2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    mergeSort(arr,0,len(arr)-1)
    print(arr)
```

Thank You !!