

DS: **Sorting**

Liwei

What is sorting

- Definition: sorting refers to arranging data in a particular format: either ascending or descending.
- Example:
 - Before sorting
 - After sorting

Sorting

- On the basis of space used
 - In-Place
 - Out-of-place
- On the basis of Stability
 - Stable
 - Un-stable

In-Place vs Out-Place Sorting

- In-Place sort:
 - Sorting algorithms which does not require any extra space for sorting
 - Example – Bubble sort
- Out-Place sort:
 - Sorting algorithms which requires extra space for sorting
 - Example – Merge sort

Stable vs Un-Stable Sorting

- Stable sort:

- If a sorting algorithm after sorting the contents does not change the sequence of similar content in which they appear, is called Stable sorting.

- Example – Insertion sort

30	10	40	50	70	50	20	80
10	20	30	40	50	50	70	80

- Un-Stable sort:

- If a sorting algorithm after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.

- Example – Quick Sort

30	10	40	50	70	50	20	80
10	20	30	40	50	50	70	80

Why stable sort is important?

- Scenarios where sort key is not the entire identity of the item
- Consider a person object with a name and a age. Let's say we sorted based on their name. If we were to sort by age in a stable way. We'd guarantee that our original ordering would be perserved for people with the same age.

UnSorted Data		Sorted by Name		Sorted by Age (Stable)		Sorted by Age (Unstable)	
Name	Age	Name	Age	Name	Age	Name	Age
Reena	1	Nalini	2	Preeti	1	Preeti	1
Nalini	2	Preeti	1	Reena	1	Sita	1
Reshma	2	Reena	1	Sita	1	Reena	1
Preeti	1	Reshma	2	Nalini	2	Nalini	2
Sita	1	Sita	1	Reshma	2	Reshma	2

Terminologies

- Increasing order
 - If a successive element is greater than its previous one
 - Example: 1,3,4,6,8,9
- Decreasing order
 - If a successive element is less than its previous one
 - Example: 9,8,6,4,3,1
- Non-increasing order
 - If a successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values.
 - Example: 9,8,6,3,3,1
- Non-Decreasing order
 - If a successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values.
 - Example: 1,3,3,6,8,9

Bubble Sort

What is bubble sort

- Bubble sort, sometimes is also referred as Sinking sort
- Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

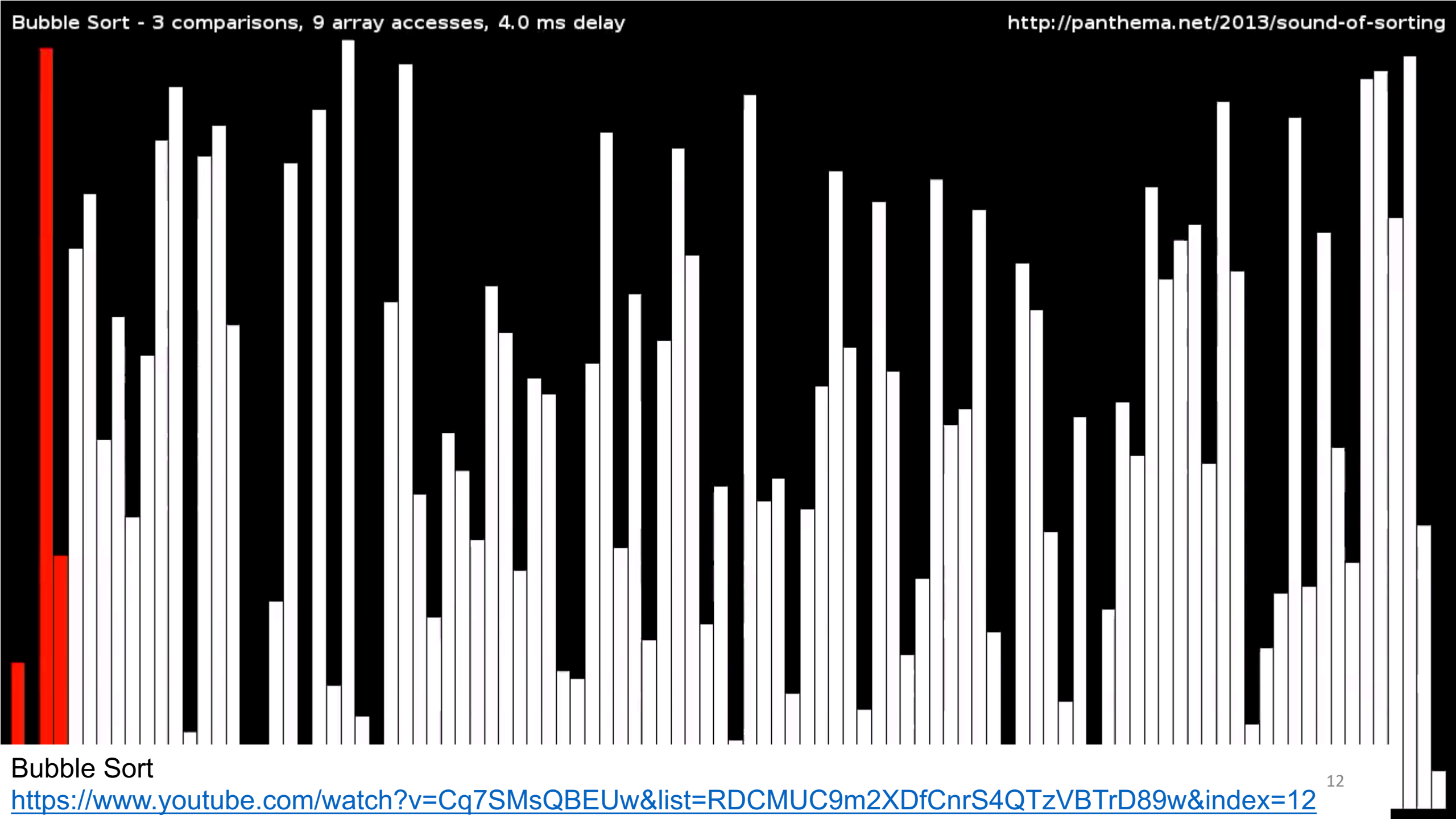
Bubble sort algorithm

```
bubbleSort(int arr[])  
    int n = arr.length  
    loop i = 0 to n-1  
        loop j = 0 to n-i-1  
            if(arr[j] > arr[j+1])  
                swap(arr[j], arr[j+1])
```

Time Complexity: $O(n^2)$

When to use / avoid Bubble sort

- When to use
 - Space is a concern
 - Easy to implement
- When not to use
 - Average case time complexity is poor



Selection Sort

Selection sort

- Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

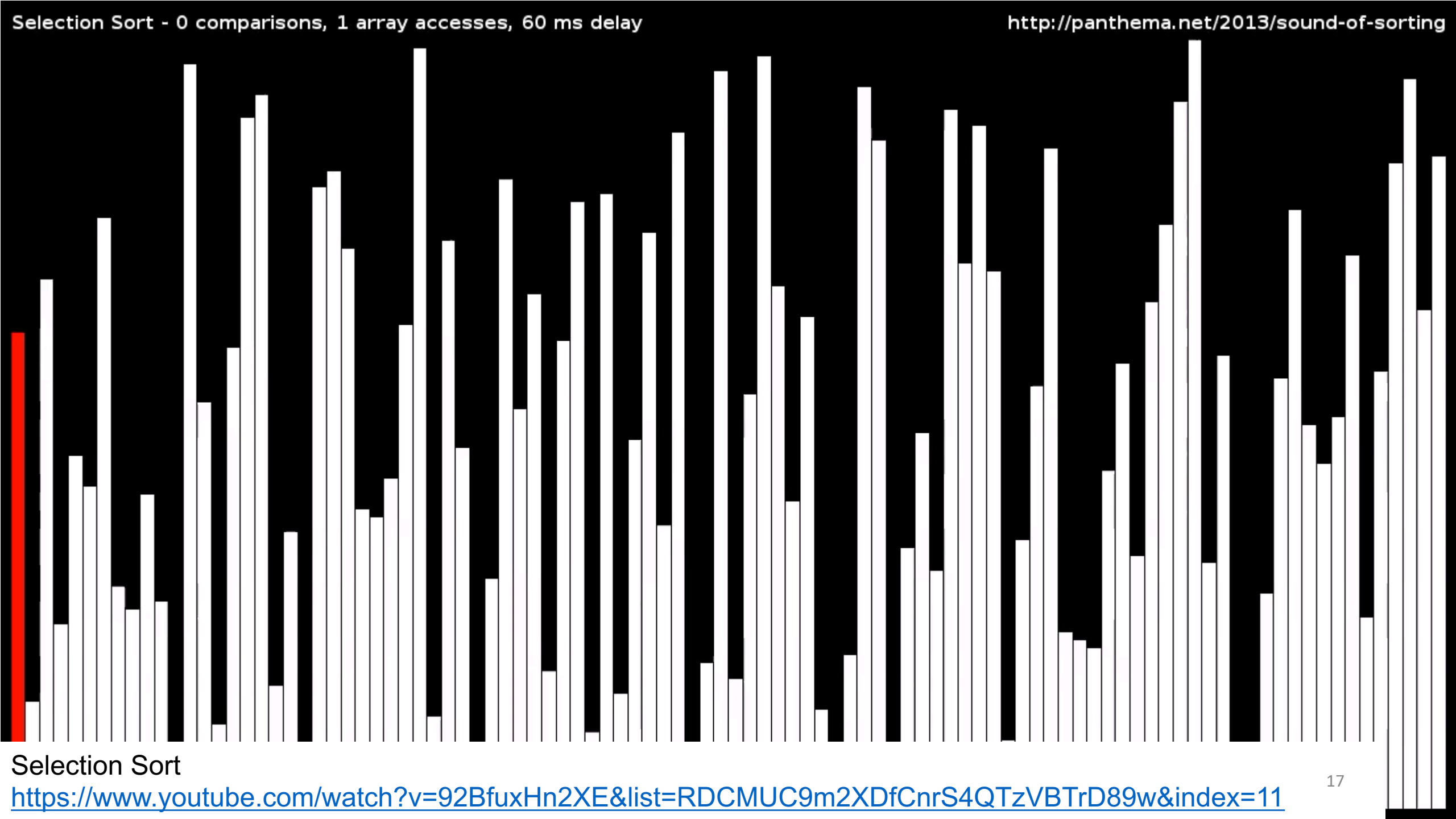
Selection sort algorithm

```
selectionSort(int arr[])  
    loop j = 0 to n-1  
        int iMin = j;  
        loop i = j+1 to n-1  
            if(arr[i] < arr[iMin])  
                iMin = i;  
        if (iMin != j)  
            swap(arr[j], arr[iMin])
```

Time Complexity: $O(n^2)$

When to use / avoid Selection sort

- When to use
 - When we don't have additional memory
 - Easy to implement
- When not to use
 - When time complexity is a concern



Insertion Sort

Insertion sort

- In Insertion sort algorithm, we divide the given array into 2 parts. i.e., Sorted & Unsorted.
- Then from Unsorted we pick the first element and find its correct position in sorted array
- Repeat till Unsorted array is empty.

Insertion sort algorithm

InsertionSort(A)

 loop i=1 to n

 currentNumber = A[i], j=i

 while (A[j-1] > currentNumber && j>0)

 A[j] = A[j-1];

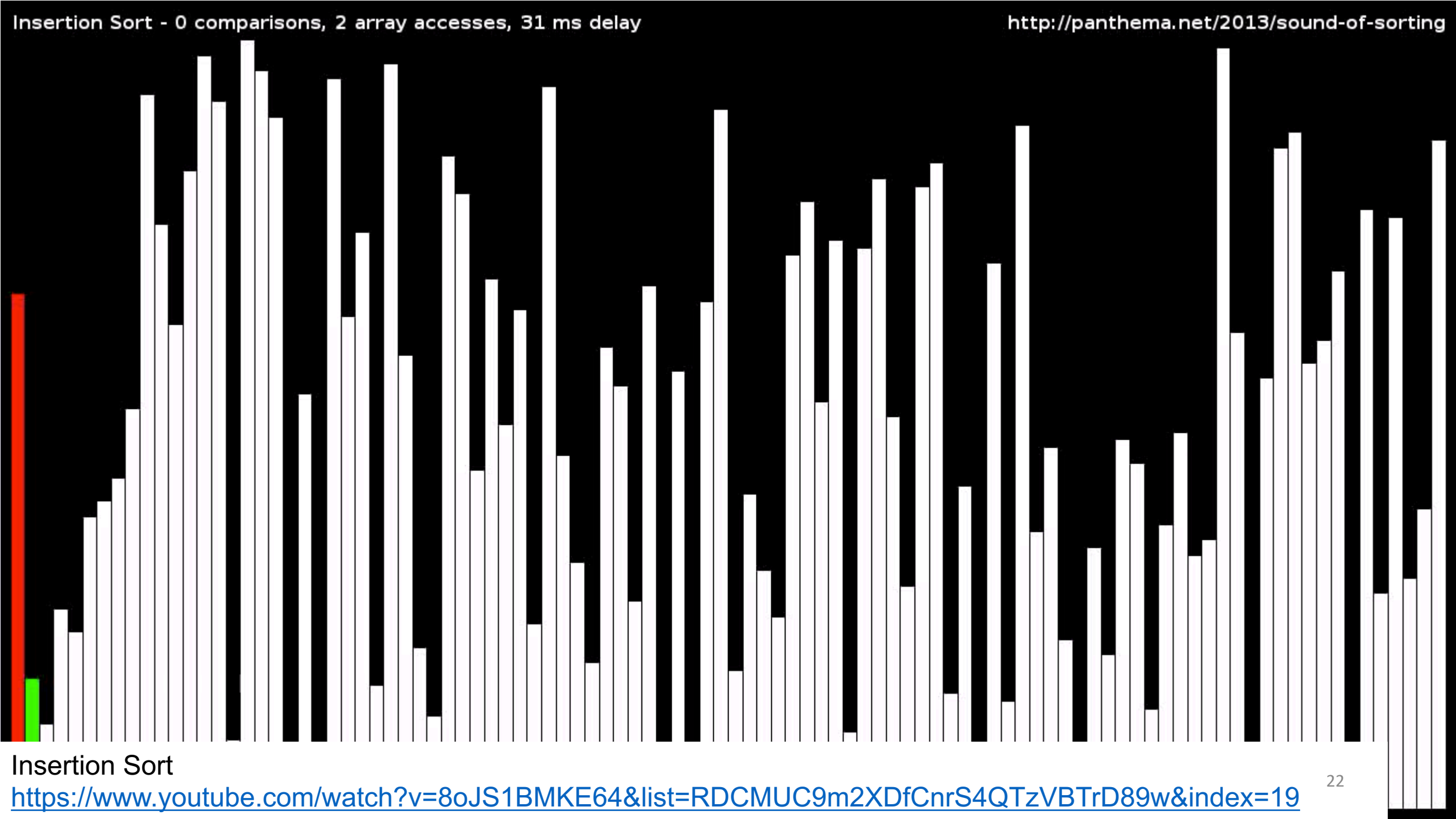
 j--;

 A[j] = currentNumber;

Time Complexity: $O(n^2)$

When to use / avoid Insertion sort

- When to use
 - No extra space
 - Easy to implement
 - Best when we have continuous inflow of numbers and we want to keep the list sorted
- When not to use
 - Average case is bad

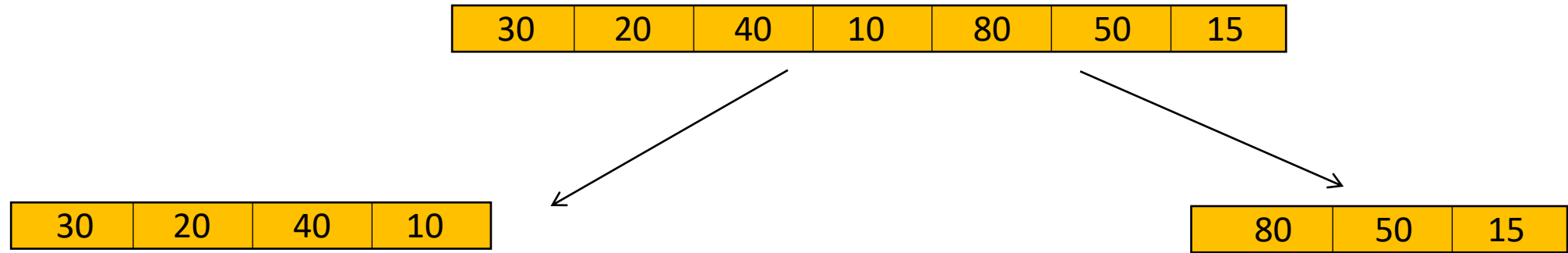


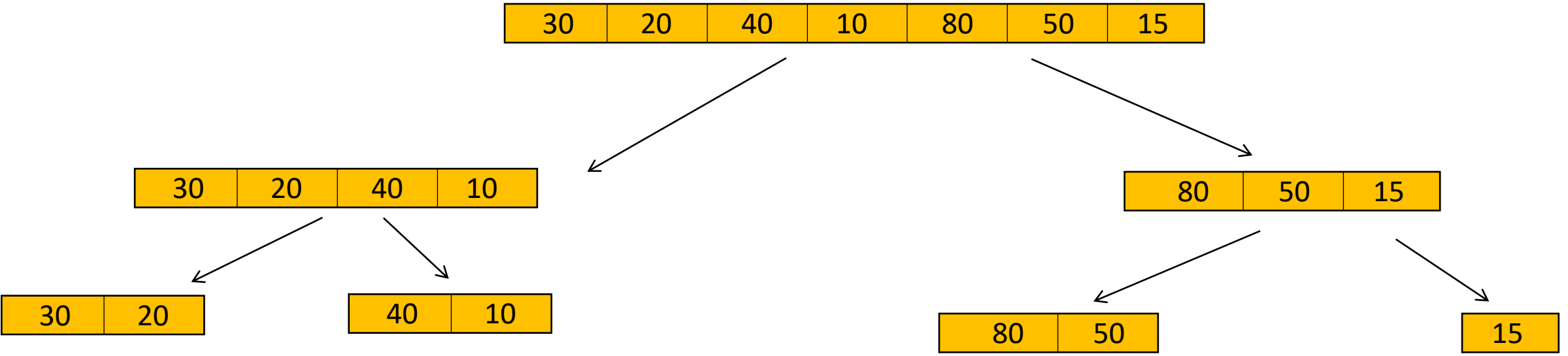
Merge Sort

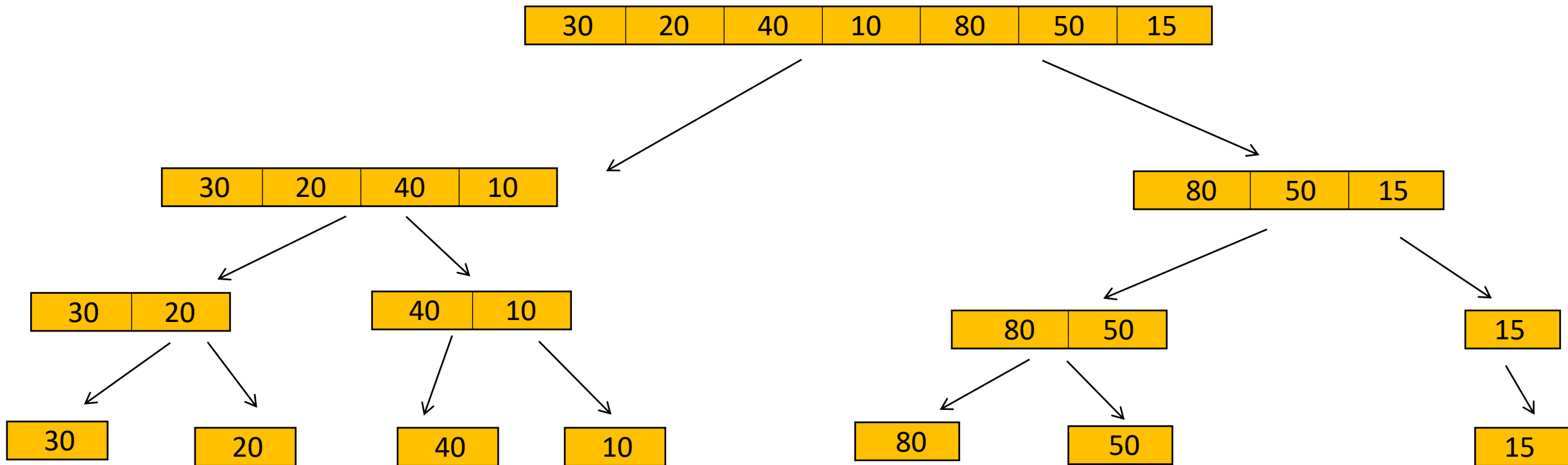
Merge sort

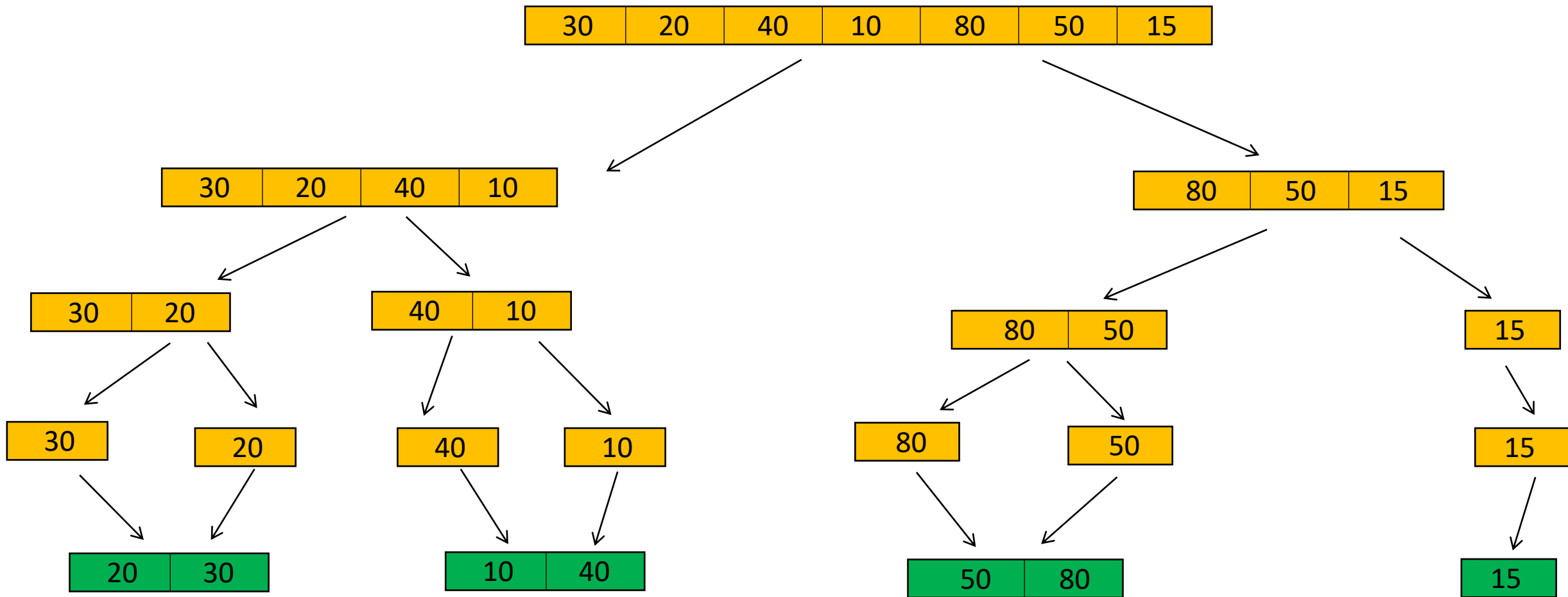
- Merge Sort is a Divide and Conquer algorithm
- It divides input array into two halves, keeps breaking those 2 halves recursively until they become too small to be broken further.
- Then each of the broken pieces are merged together to inch towards final answer

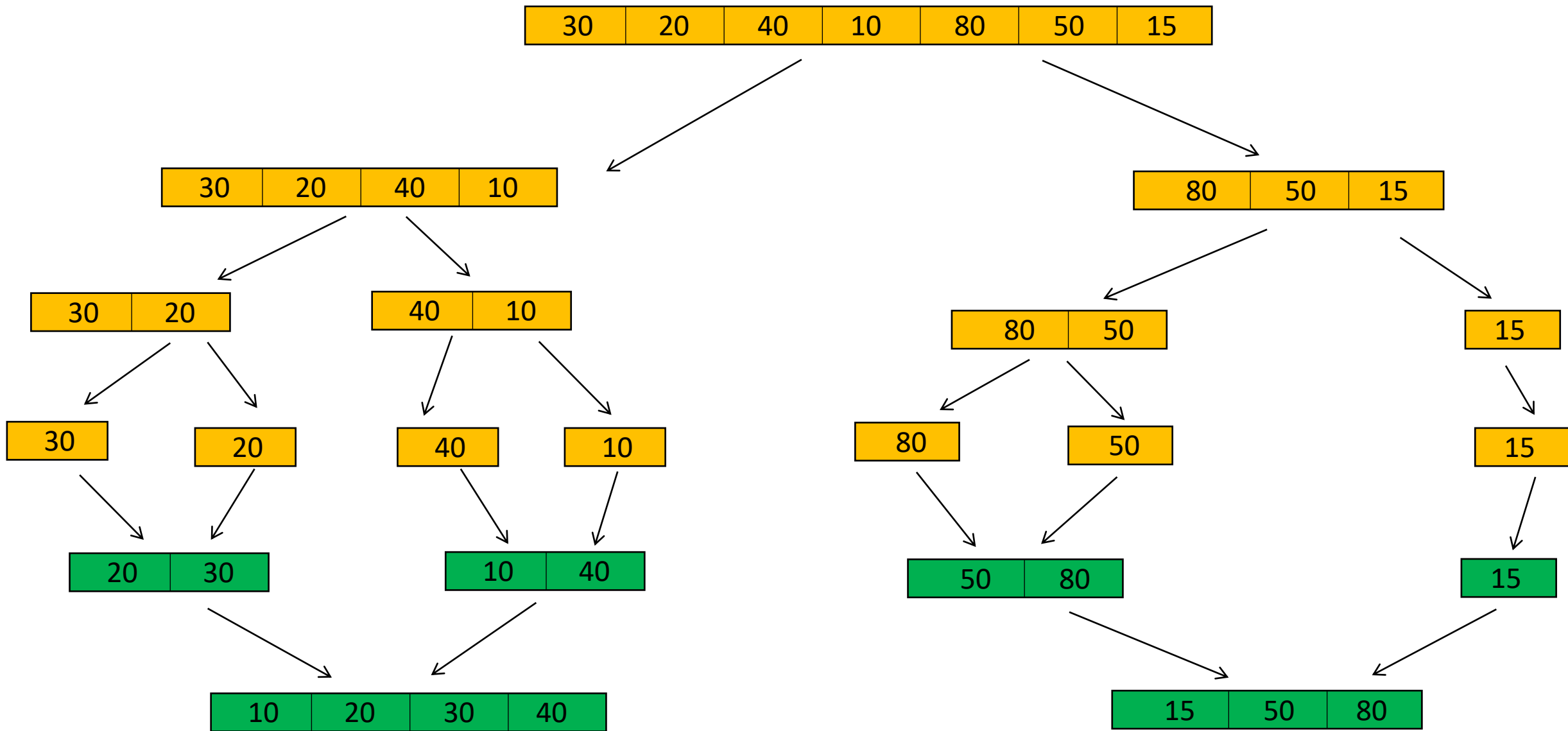
30	20	40	10	80	50	15
----	----	----	----	----	----	----

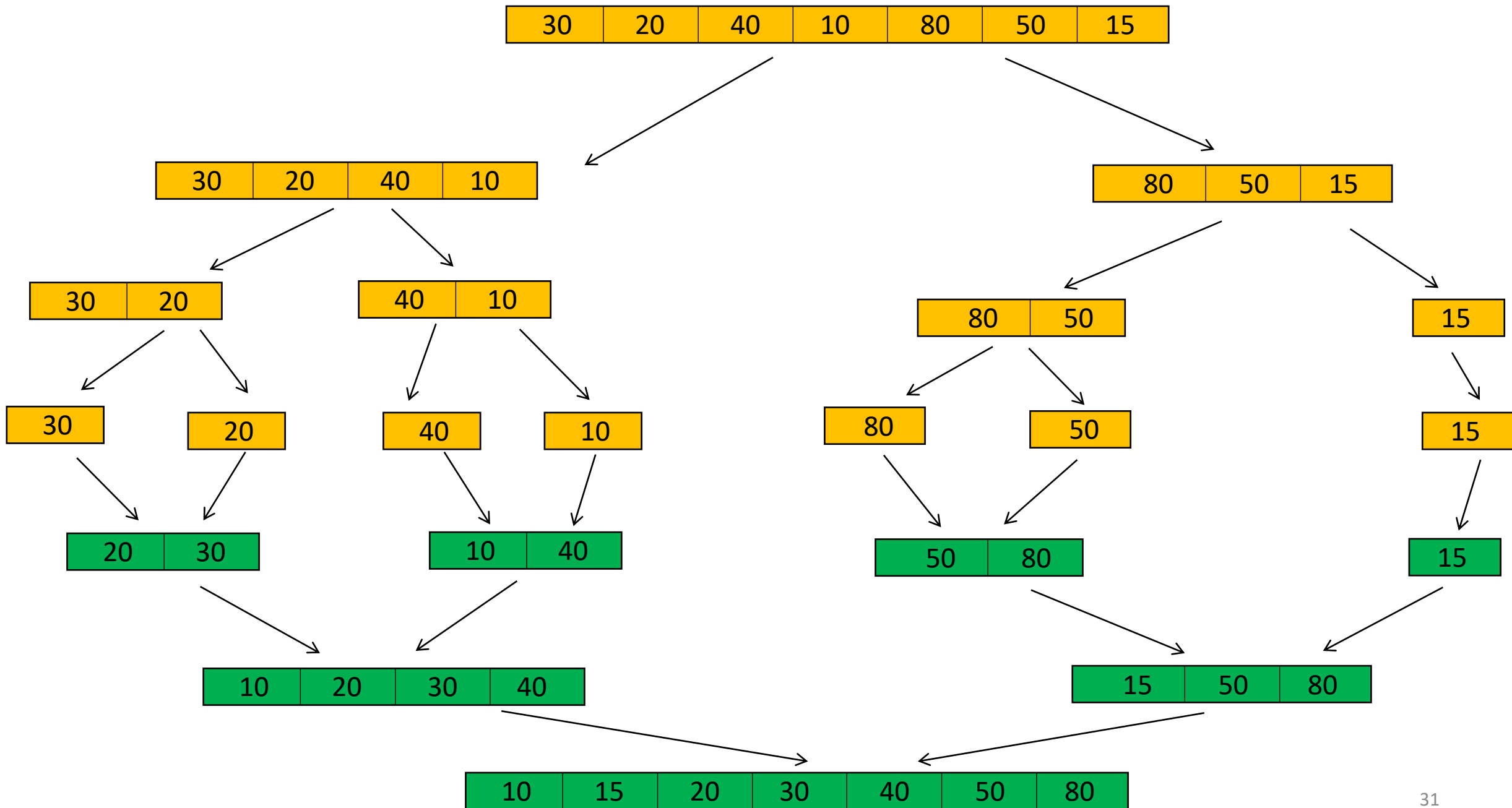












merge sort algorithm

```
mergeSort(A,l,r)
    if r > l
        middle m = (l+r)/2
        mergeSort(A,l,m)
        mergeSort(A,m+1,r)
        merge(A,l,m,r)
```

```
merge(A,p,m,r)
    create tmp array L & R
    copy A,p,m into L & A m+1,r into R
    i=j=0
    loop k = p to r
        if L[i] < R[j]
            A[k] = L[i++];
        else
            A[k] = R[j++];
```

Time complexity: $O(n \log n)$

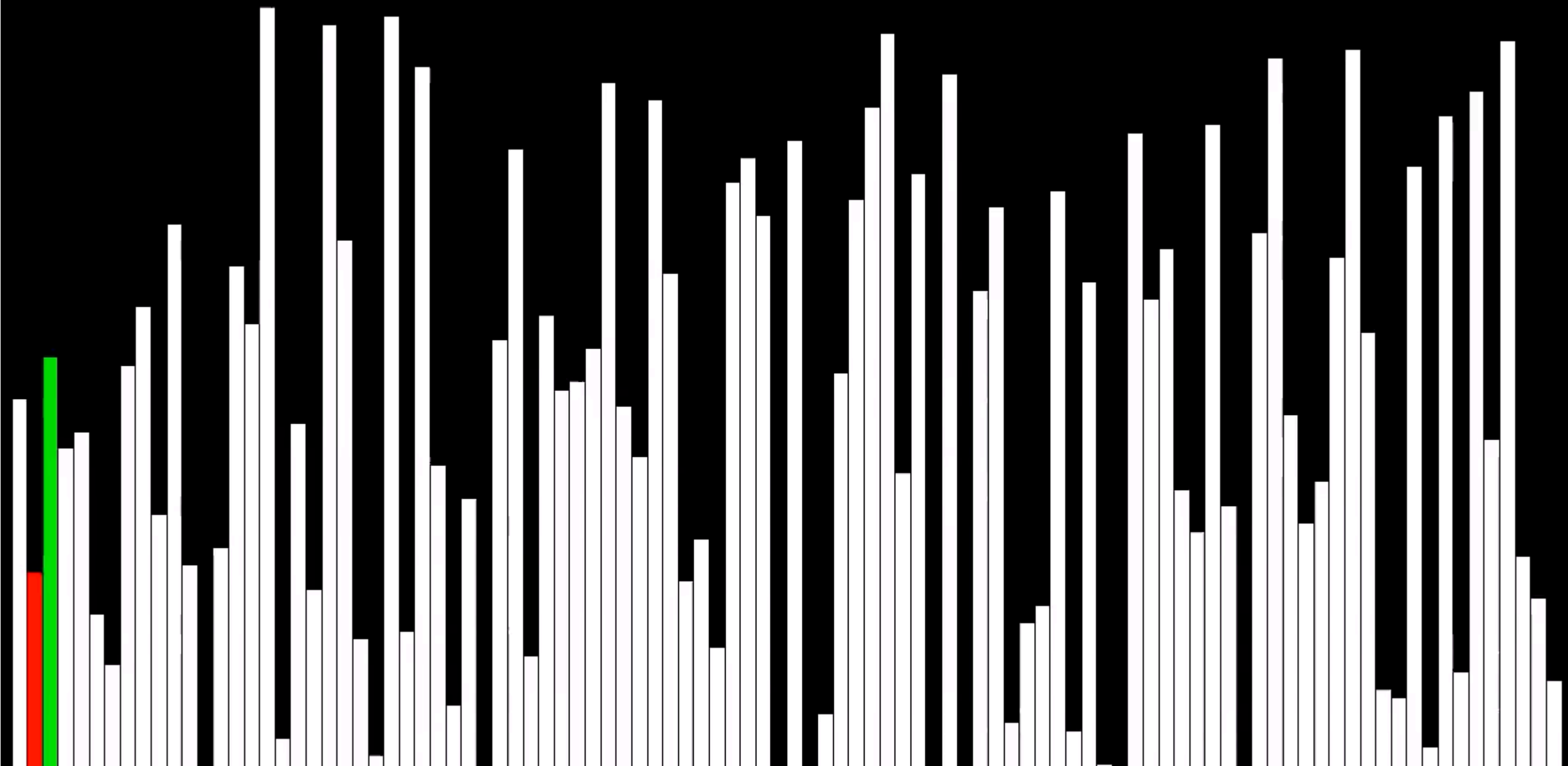
Time Complexity: $O(n \log n)$

Time Complexity

$$\begin{aligned}T(n) &= O(1) + O(1) + T(n/2) + T(n/2) + O(n) \\&= 2T(n/2) + O(n) \\&= 2(2T(n/4) + O(n/2)) + O(n) \\&= 4(T(n/4)) + 2*O(n) + O(n) \quad // O(n/2) \text{ can be represented as } O(n) \\&= 2^k T(n/2^k) + k*O(n) \quad // \text{ now replace } k \text{ with } \log n \text{ to meet the base condition} \\&= 2^{\log n} * 1 + \log n * O(n) \\&= O(n \log n)\end{aligned}$$

When to use / avoid Merge sort

- When to use
 - When you need a stable sort
 - When average expected time is $O(n \log n)$
- When not to use
 - When space is a concern like embedded systems
 - Ps. Java 6 and earlier versions use merge sort as the sorting algorithms



merge sort

<https://www.youtube.com/watch?v=ZRPoEKHXTJg&list=RDCMU9C9m2XDfCnrS4QTzVBTrD89w&index=4>

Quick Sort

Quick sort

- Quick Sort is a Divide and Conquer algorithm
- At each step it finds “Pivot” and then makes sure that all the smaller elements are left of “Pivot” and all bigger elements are Right of “Pivot”.
- It does this recursively until the entire array is sorted
- Unlike Merge Sort it does not requires any external space

9	4	6	3	7	1	2	11	5
---	---	---	---	---	---	---	----	---

4	3	1	2	5	6	9	11	7
---	---	---	---	---	---	---	----	---

1	2	4	3	5	6	9	11	7
---	---	---	---	---	---	---	----	---

1	2	3	4	5	6	9	11	7
---	---	---	---	---	---	---	----	---

1	2	3	4	5	6	7	11	9
---	---	---	---	---	---	---	----	---

1	2	3	4	5	6	7	11	9
---	---	---	---	---	---	---	----	---

1	2	3	4	5	6	7	11	9
---	---	---	---	---	---	---	----	---

1	2	3	4	5	6	7	9	11
---	---	---	---	---	---	---	---	----



quick sort algorithm

```
quickSort(A,p,q)
```

```
    if p < q
```

```
        r = partition (A,p,q)
```

```
        quickSort(A,p,r-1)
```

```
        quickSort(A,r+1,q)
```

```
Partition(A,p,q)
```

```
    pivot = q
```

```
    i = p-1
```

```
    loop j = p to q
```

```
        if A[j] <= A[pivot]
```

```
            i++
```

```
            swap (A[i], A[j])
```

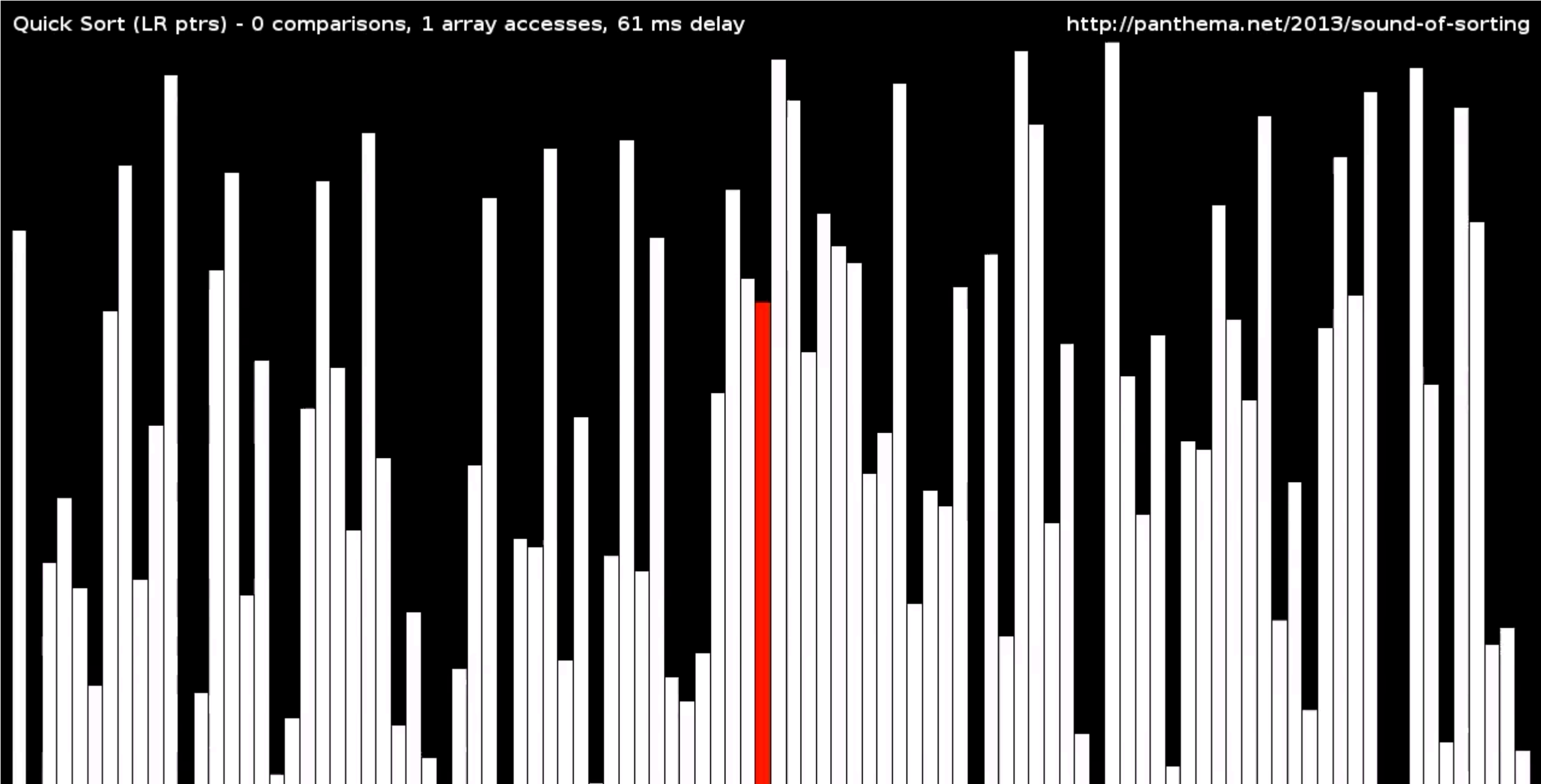
Time Complexity: $O(n \log n)$

Time Complexity

$$\begin{aligned}T(n) &= O(1) + O(1) + T(n/2) + T(n/2) + O(n) \\&= 2T(n/2) + O(n) \\&= 2(2T(n/4) + O(n/2)) + O(n) \\&= 4(T(n/4)) + 2*O(n) + O(n) \quad // O(n/2) \text{ can be represented as } O(n) \\&= 2^k T(n/2^k) + k*O(n) \quad // \text{ now replace } k \text{ with } \log n \text{ to meet the base condition} \\&= 2^{\log n} * 1 + \log n * O(n) \\&= O(n \log n)\end{aligned}$$

When to use / avoid Quick sort

- When to use
 - When average is desired to be $O(n \log n)$
- When not to use
 - When space is a concern
 - When stable sort is required



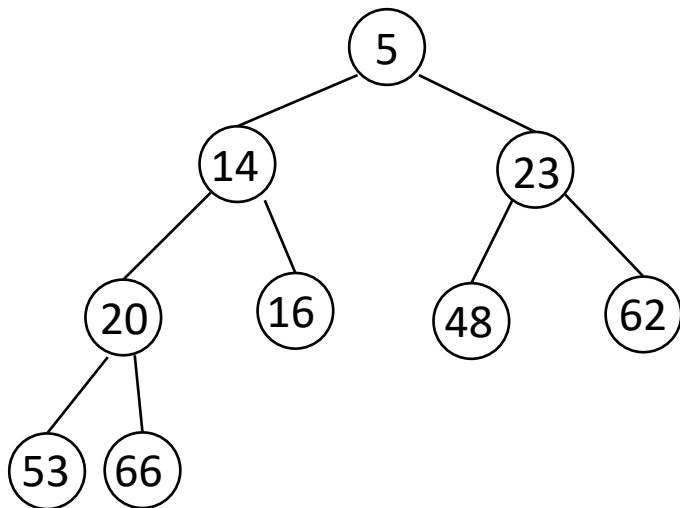
Heap Sort

Heap sort

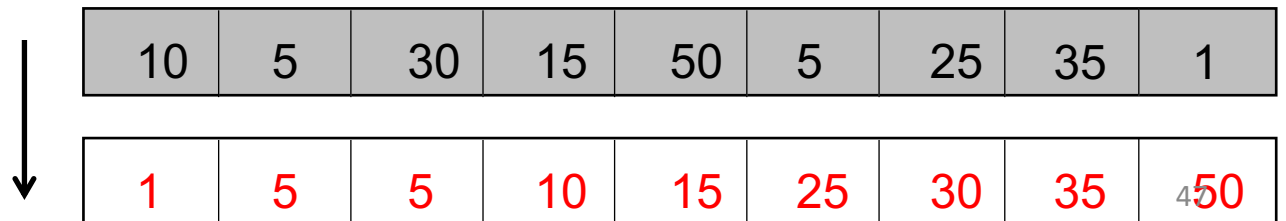
- Heap Sort works by first organizing the data to be sorted into a special type of binary tree called a heap
- It then removes the topmost item (the largest/smallest) and inserts it in current array. It keeps doing until binary heap is empty.
- It best suited with array. Does not works best with LinkedList

What is binary Heap

- Definition: Binary Heap is a Binary tree with some special properties:
- Heap Properties:
 - Value of any given node must be \leq value of its children (Min-Heap)
 - Value of any given node must be \geq value of its children (Max-Heap)



STEP 1: CREATE
STEP 2: EXTRACT



heap sort algorithm

heapSort(A)

 for i=0 to A.length-1

$O(n)$

 insertInHeap(A[i])

$O(n \log n)$

 for i=0 to A.length-1

$O(n)$

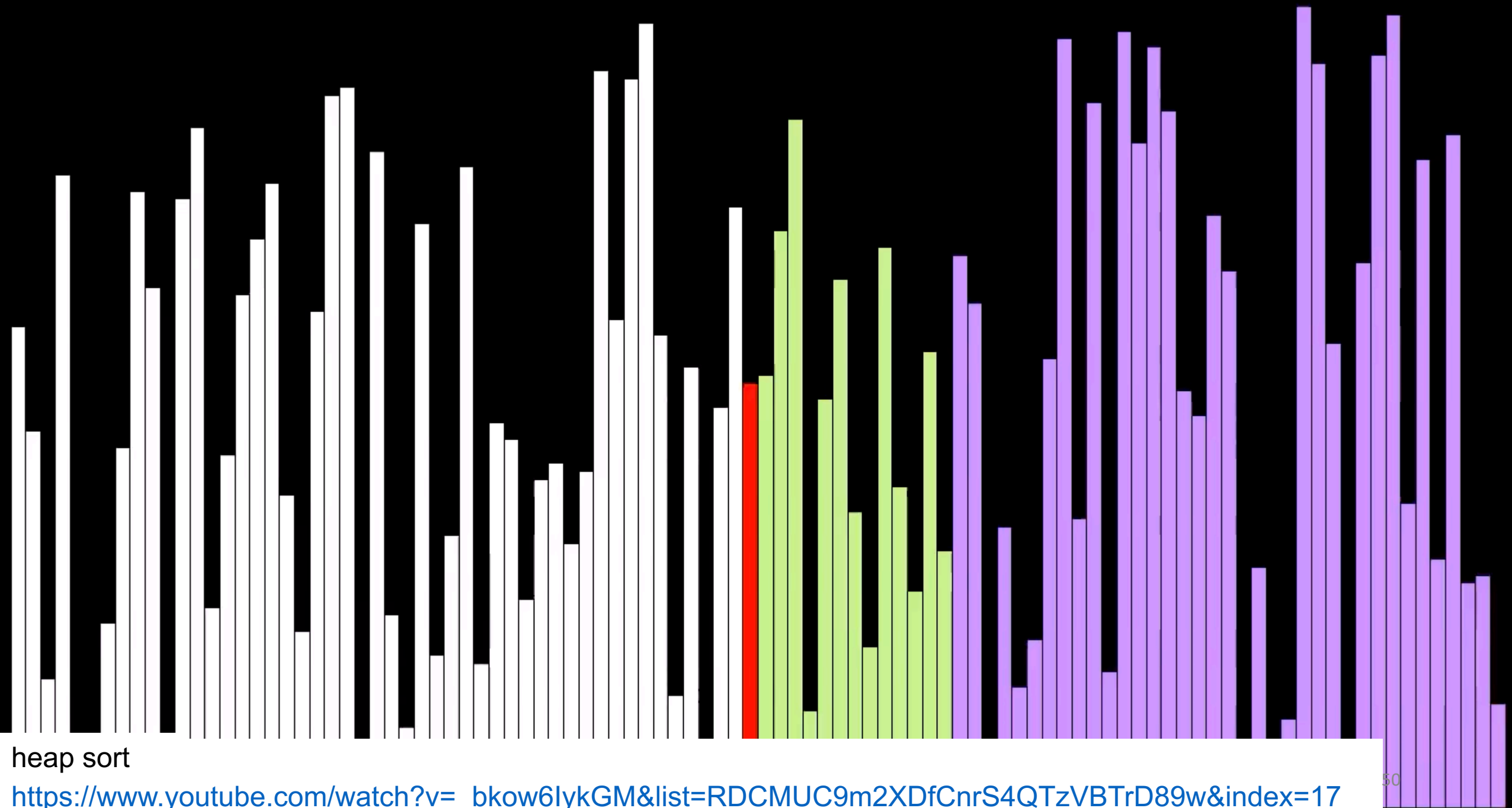
 extractFromHeap(A[i])

$O(n \log n)$

Time Complexity: $O(n \log n)$

When to use / avoid Heap sort

- When to use
 - When space is a concern
- When not to use
 - When stable sort is required



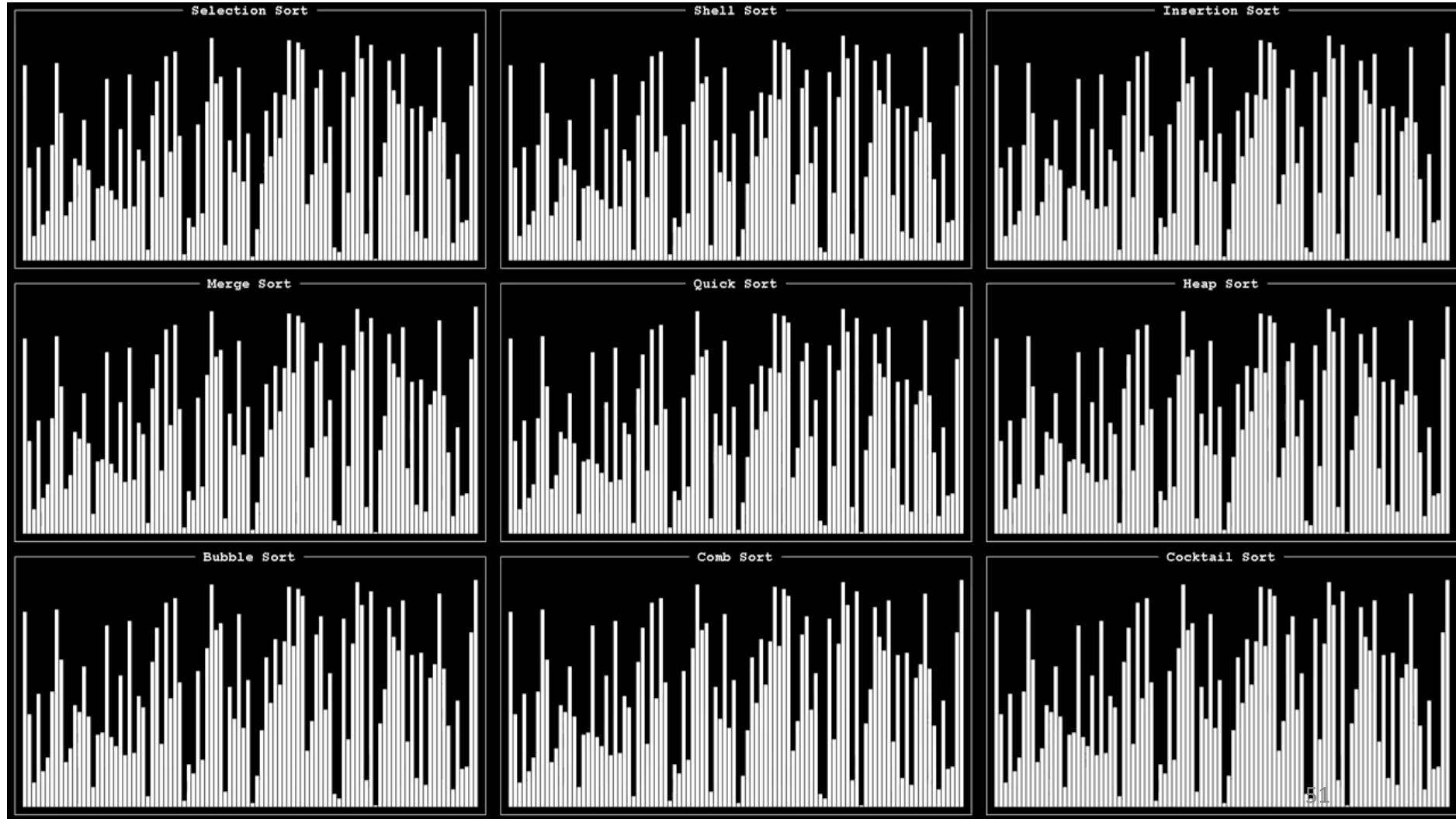
Visualization and comparison of 9 different sorting algorithms:

- selection sort
- shell sort
- insertion sort
- merge sort
- quick sort
- heap sort
- bubble sort
- comb sort
- cocktail sort

<https://www.youtube.com/watch?v=ZZuD6iUe3Pc>

With 4 types of input data:

- random [0:01](#)
- few unique [1:07](#)
- reversed [2:08](#)
- almost sorted [3:38](#)



Sorting Algorithms compared

Particulars	Time Complexity	Stable
Bubble Sort	$O(n^2)$	Yes
Selection Sort	$O(n^2)$	No
Insertion Sort	$O(n^2)$	Yes
Bucket Sort	$O(n \log n)$	Yes*
Merge Sort	$O(n \log n)$	Yes
Quick Sort	$O(n \log n)$	No
Heap Sort	$O(n \log n)$	No