Data Structure and Algorithms

Session-20

Dr. Subhra Rani Patra
SCOPE, VIT Chennai

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 not empty.

30	10	50	20	60	40

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
```

30	10	50	20	60	40

Time & Space Complexity of Insertion Sort Algorithm:

InsertionSort(A):

loop: i = 1 to n -- $O(n)$

tmp = A[i], j=i -- $O(1)$

while (A[j-1]>tmp && J>0) --------------------- $O(n)$ ------ $O(n)$ -------------- $O(n^2)$

A[j]=A[j-1] ------------------------------------- $O(1)$

j-- --- $O(1)$

A[j] = currentNumber ------------------------- $O(1)$

Time Complexity - $O(n^2)$

Space Complexity – $O(1)$

When to Use/Avoid Insertion Sort:

- ***When to use:***
  - No extra space
  - Simple implementation
  - Best when we have continuous inflow of numbers and we want to keep the list shorted

- ***When not to use:***
  - Average case is bad

Bucket Sort:

✓Bucket sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets.

✓Each bucket is then sorted individually.

40	10	30	80	70	20	60	50	100

40	10	30	80	70	20	60	50	100

- ✓Create Number of buckets = ceil/floor (squareroot of total number of items)
- ✓Iterate through each number and place it in appropriate bucket
- ✓Appropriate bucket = Ceil ((Value * number of buckets) / max value in array)
- ✓Sort all the buckets
- ✓Merge all the buckets

Bucket Sort Algorithm:

BucketSort(A):

find no. of buckets to be created and create those buckets B1[ ], B2[ ]…

find divisor value

loop: i = 0 to n-1

insert A[i] into array B[ ] using divisor and bucket#

sort B[ ] with insertion sort (or any sort as per comfort)

concatenate B1[ ], B2[ ],B3[ ]…

Time & Space complexity of Bucket Sort Algorithm:

BucketSort(A):

find no. of buckets to be created and create those buckets B1[ ], B2[ ]… ------------- O(1)

find divisor value --- O(1)

loop: i = 0 to n-1 --- O(n)

insert A[i] into array B[ ] using divisor and bucket# -- O(1)

sort B[ ] --- O(n logn)

concatenate B1[ ], B2[ ],B3[ ]… --- O(n)

Time Complexity *= O(n) + O(n log n) + O(n)*

= O(n logn)

Space Complexity *– O(n)*

When to Use/Avoid Bucket Sort:

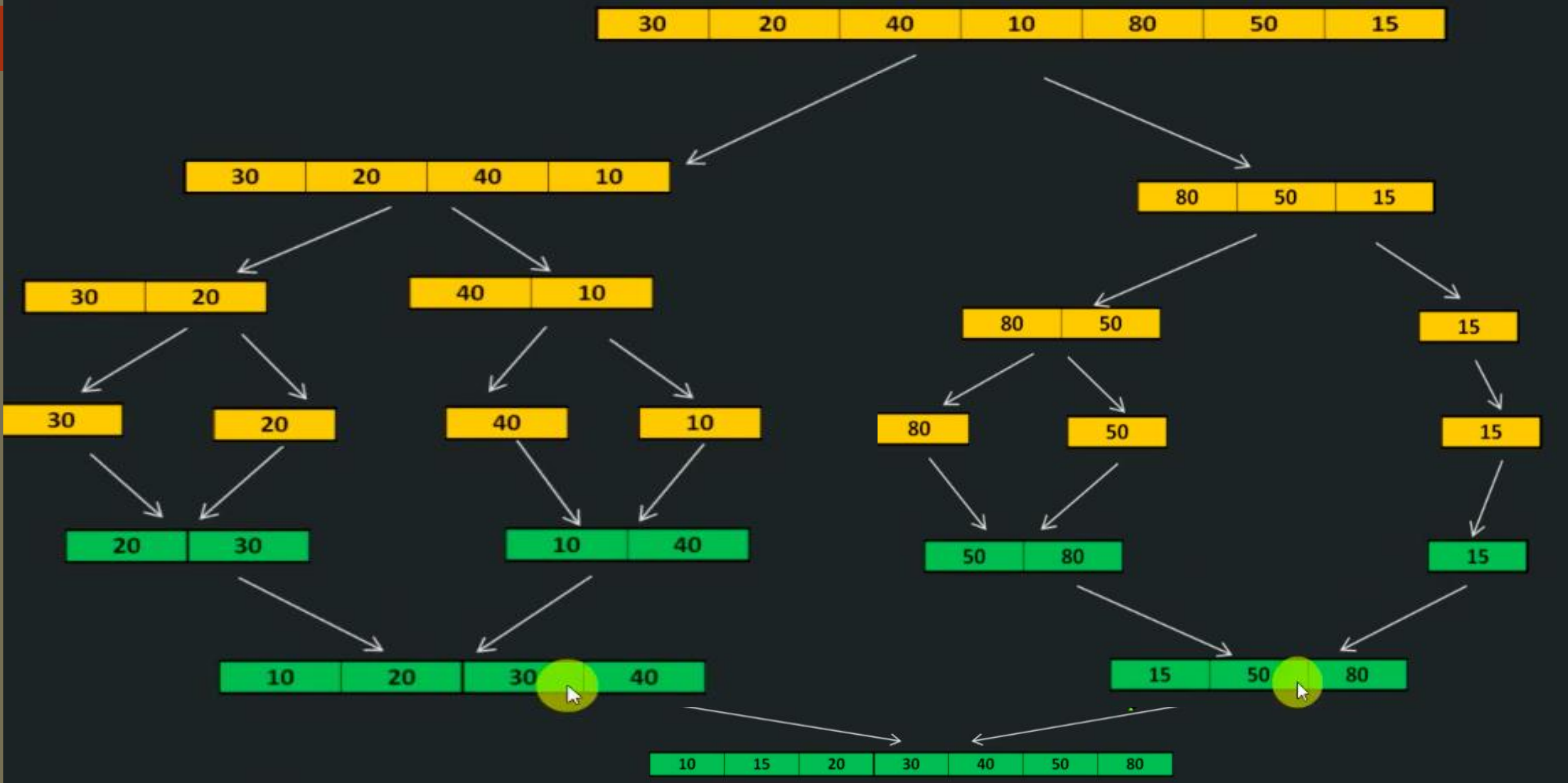
40	10	30	80	70	20	60	50	100

- ✓ ***When to use:***
  - ✓ *When input is uniformly distributed over a range*
- ✓ ***When not to use:***
  - ✓ *When space is a concern*

Merge Sort:

- ✓ Merge Sort is a Divide and Conquer algorithm.
- ✓ It divides input array in 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.

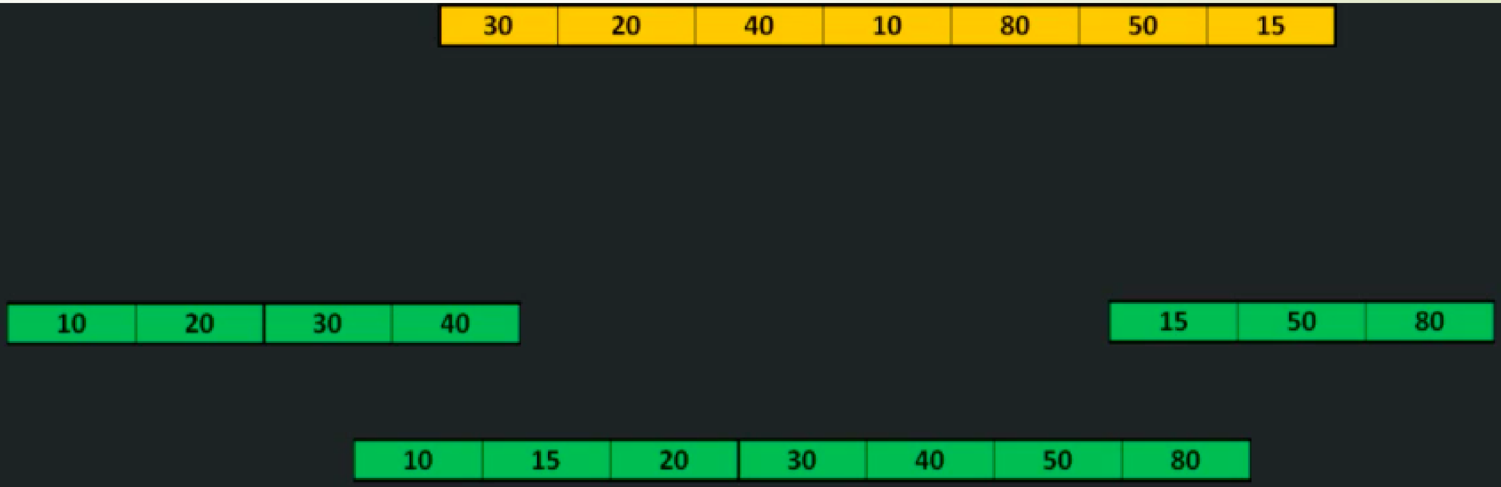
Merge sort



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

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 arrays L & R and 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]; i++
      else
          A[k] = R[j]; j++
```

Time & Space complexity of Merge Sort Algorithm:

```
mergeSort(A,l,r): ------------------------------------ T(n)
    if r > l ------------------------------------------ O(1)
        middle m = (l+r)/2 ------------------------- O(1)
        mergeSort(A,l,m) --------------------------- T(n/2)
        mergeSort(A,m+1,r) ------------------------ T(n/2)
        merge(A,l,m,r) ------------------------------ O(n)

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

Time & Space complexity of Merge Sort Algorithm:

$T(n) = O(1) + O(1) + T(n/2) + T(n/2) + O(n)$

$= 2T(n/2) + O(n)$

$= 2(2T(n/4) + O(n/2)) + O(n)$

$= 4(T(n/4)) + 2*O(n) + O(n)$ // O(n/2) can be represented as O(n)

$= 2^k T(n/2^k) + k*O(n)$

$= 2^k T(n/2^k) + k*O(n)$ { now lets replace 'k' with logn to meet the base condition}

$= 2^{\log n} * 1 + \log n * O(n)$

$= o(n \log n)$

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \log \log n)$
   c. If $p < -1$, then $T(n) = \Theta(n^{\log_b^a})$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

Equation#1: $T(n) = 2T(n/2) + O(n)$

Equation#2: $T(n/2) = 2T(n/4) + O(n/2)$

Equation#3: $T(n/4) = 2T(n/8) + O(n/4)$

Base Condition: $T(1) = 1$

Space Complexity = $O(n)$

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

Thank you