DATA STRUCTURES AND ALGORITHM

Module

6

- To understand the difference sorting technique

SORTING

Sorting

- Arranging of names and numbers in meaningful ways.
- The process of placing elements from a collection in some kind of order.

SORTING

Temp = A
A = B
B = Temp

SORTING OPERATIONS

1	2	2
A	B	temp

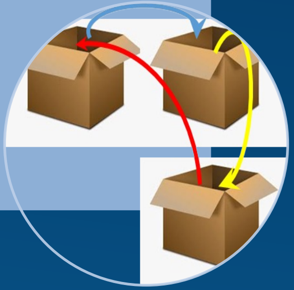
- Test whether $A_i < A_j$ or test $A_i > A_j$

Comparison

= < >
≠ ≤ ≥

- Switches the contents of A and B, B and C or C and A

Interchange



- Set A=B, then set B = C or set C = A

Assignments

X = Y

SORTING

Sorting techniques examples

- ✓ Bubble sort
- ✓ Insertion sort
- ✓ Selection sort
- ✓ Quick sort
- ✓ Merge sort
- ✓ Heap sort - max
- ✓ Binary sort
- ✓ Shell sort
- ✓ Radix sort

BUBBLE SORTING

- Makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order.
- Each pass through the list places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs.

BUBBLE SORTING 2 1

ALGORITHM (BUBBLE SORT)

- input n numbers of an array A
- initialise i equal to 0 and repeat through sub steps if i is less than n
  - ➢ initialise j equal to 0 and repeat through sub steps if j is less than n – 1
  - ➢ do sub steps if A[j] is greater than A[j+1]
    - ✓ assign **A[j]** to **swap** **A[0]** **swap 2**
    - ✓ assign **A[j+1]** to **A[j]** **A[1]** **A[0] 1**
    - ✓ assign **swap** to **A[j]** swap A[0] 2
- display the sorted numbers of array **A**
- exit

BUBBLE SORTING

```
void bubbleSort( A[1..n] ) {
  for i = 1 to n-1
     for j = n downto i+1
          if A[ j ] < A[ j–1 ]
               swap A[ j ] with A[ j–1 ]      }
```

BUBBLE SORTING

First pass →

54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

Bubble sort: The First Pass

BUBBLE SORTING

Given: A[ ] = {35, 10, 55, 20, 5}

35	10	55	20	5

FIRST PASS

10	35	20	5	55

SECOND PASS

10	20	5	35	55

THIRD PASS

10	5	20	35	55

FORTH PASS

BUBBLE SORTING

Given: A[ ] = { 35, 10, 55, 20, 5 }

35	**10**	55	20	5
10	**35**	**55**	20	5
10	35	**55**	**20**	5
10	35	20	**55**	**5**
10	35	20	5	55

FIRST PASS

10	**35**	20	5	55
10	**35**	**20**	5	55
10	30	**35**	**5**	55
10	30	5	**35**	**55**
10	30	5	35	55

SECOND PASS

10	**30**	5	35	55
10	**30**	**5**	35	55
10	5	**30**	**35**	55
10	5	30	**35**	**55**
10	5	30	35	55

THIRD PASS

10	**5**	30	35	55
5	**10**	**30**	35	55
5	10	**30**	**35**	55
5	10	30	**35**	**55**
5	10	30	35	55

FORTH PASS

SELECTION SORTING

- Improves on the bubble sort by making only one exchange for every pass through the list.
- Finds the smallest element of the array and interchange it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

SELECTION SORTING

Given: A[ ] = { 35, 10, 55, 20, 5 }

35	10	55	20	5
5	10	55	20	35
5	10	55	20	35
5	10	20	55	35
5	10	20	35	55

SELECTION SORTING

ASCENDING ORDER

Given: A[ ] = { 35, 10, 55, 20, 5 }

35	10	55	20	5

First Pass

5	10	55	20	35

Second Pass

5	10	55	20	35

Third Pass

5	10	20	55	35

Forth Pass

5	10	20	35	55

Sorted Value

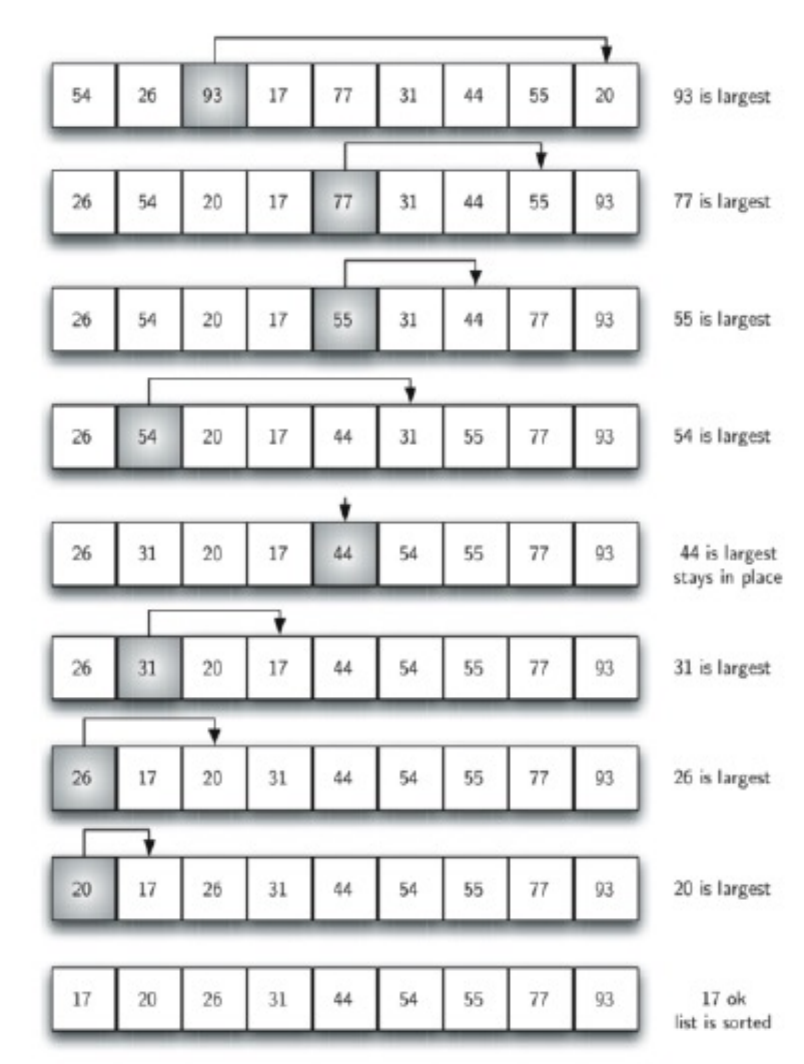
SELECTION SORTING

DESCENDING ORDER

- In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location.
- process continues and requires $n - 1$ passes to sort n items, since the final item must be in place after the $(n - 1)$st pass.

SELECTION SORTING

DESCENDING ORDER



INSERTION SORTING

- Sort a set of values by inserting values into an existing sorted file.
- Compare the second, place it before the first one. Otherwise place it just after the first one.
- Compare the third value with the second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place.
- And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place.

INSERTION SORTING

ALGORITHM (INSERTION SORT)

1. Let A be a linear array of n numbers, temp is temporary variable for swapping (or interchange) the position of the numbers.
2. Input n numbers of an array A
3. initialise i equal to 1 and repeat through sub steps if i is less than n
    - assign A[i] to temp
    - initialise j equal i minus 1 and repeat through sub steps,
      if j is greater than or equal to 0 and temp is less than A[j]
        assign A[j+1] to A[j]
        decrement the value j
      assign A[j+1] to A[j]
      increment the value of i
4. Display the sorted numbers of array A
5. Exit

INSERTION SORTING

Given: A[ ] = { 35, 10, 55, 20, 5 } n - 1

35	10	55	20	5
10	35	55	20	5
10	35	55	20	5
10	20	35	55	5
5	10	20	35	55

INSERTION SORTING

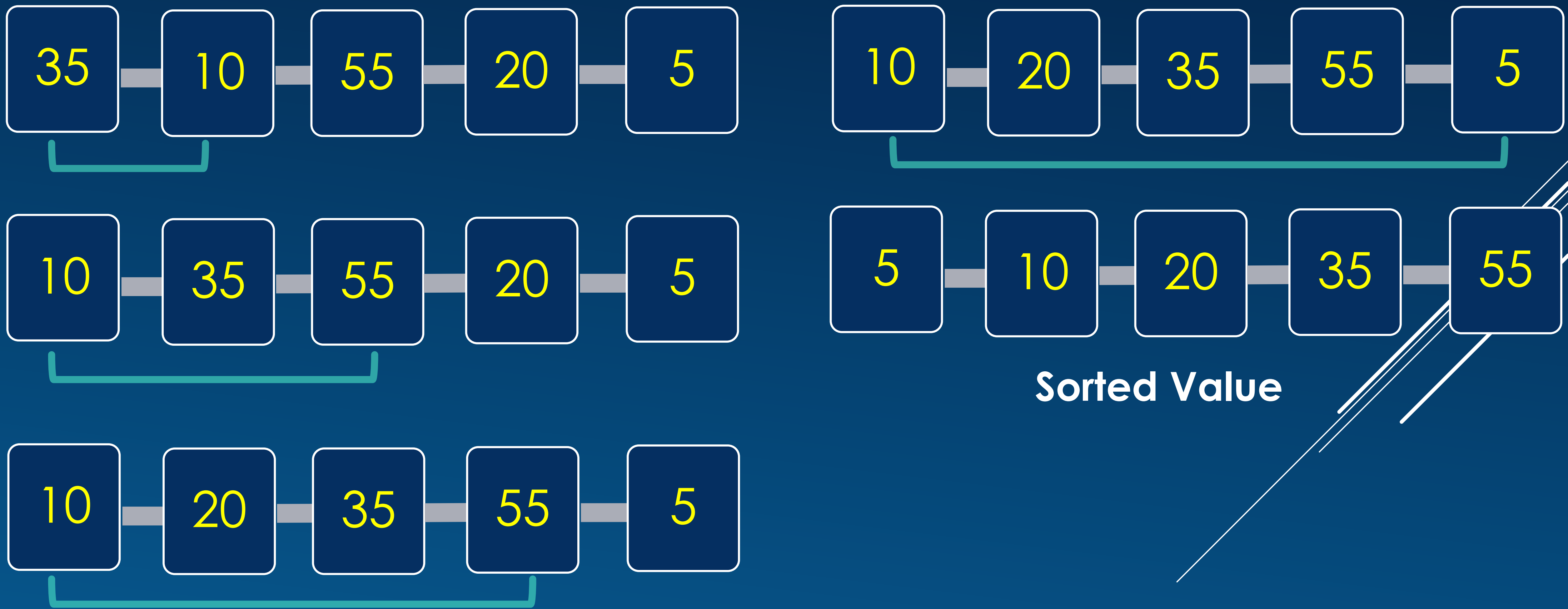
Given: A[ ] = { 5, 3, 4, 1, 3 } n - 1

5	3	4	1	3

INSERTION SORTING

ASCENDING ORDER

Given: A[ ] = { 35, 10, 55, 20, 5 }



SEATWORK

ASCENDING ORDER

BUBBLE | INSERTION | SELECTION

Given: A[ ]

-3	-5	-7	-1	-15	6

Filename : Section_Lastname_Sorting.jpg

Insertion

SEATWORK

ASCENDING ORDER

BUBBLE | INSERTION | SELECTION

Given: A[ ]

-3	-5	-7	-1	-15	6

SEATWORK

ASCENDING ORDER

BUBBLE | INSERTION | SELECTION

Given: A[ ] = 3, 6, 1, 2

Insertion

Selection

LONG QUIZ BOTH BSCS .1-.2

APRIL 22, 2022

- Long Quiz #3
  Linked List | Queues

APRIL 22, 2022

- Long Quiz #4
  Sorting (Bubble, Insertion, Merge Selection, Quicksort, Heapsort)

LABORATORY

APRIL 12, 2022
5:30PM

- Sorting

MERGE SORT

MERGE SORT n/2

- Merge sort is based on the divide-and-conquer paradigm.
- Deals with subproblems, we state each subproblem as sorting a subarray.
- These values change as we recursive through subproblems.

MERGE SORT right

- **ALGORITHM**

Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).

Repeatedly Merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

MERGE SORT

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

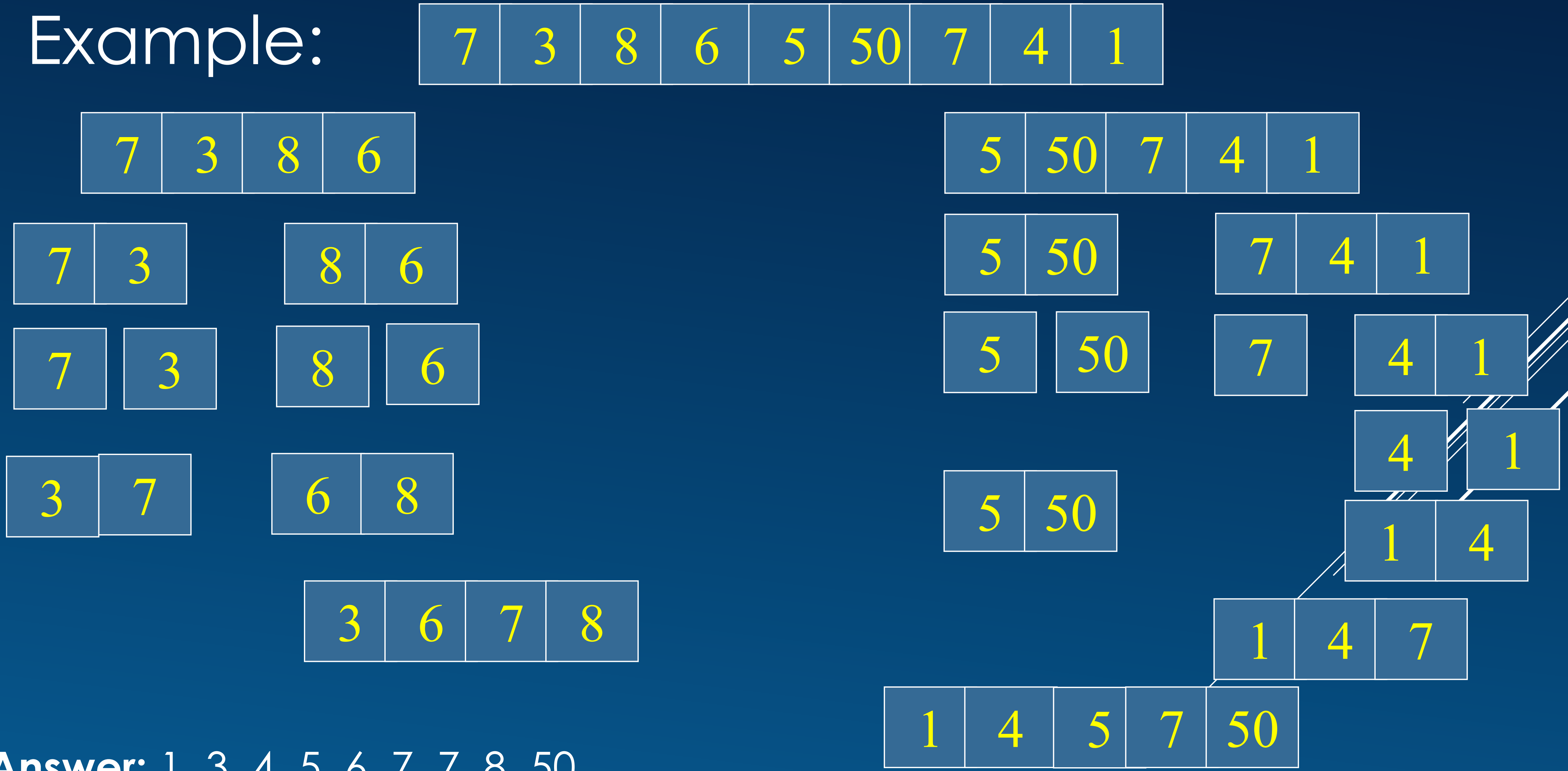
MERGE SORT

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \le C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

MERGE SORT

Example:



Answer: 1, 3, 4, 5, 6, 7, 7, 8, 50

MERGE SORT

Example:

40	20	10	80	60	50	7	30

Answer:

QUICK SORT

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

Choose a pivot value.
We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.

QUICK SORT

Partition. Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.

Sort both parts. Apply quicksort algorithm recursively to the left and the right parts.

QUICK SORT

There are two indices i and j and at the very beginning of the partition algorithm i points to the first element in the array and j points to the last one.

Then algorithm moves i forward, until an element with value greater or equal to the pivot is found.

QUICK SORT

Index j is moved backward, until an element with value lesser or equal to the pivot is found.

If $i \leq j$ then they are swapped and i steps to the next position $(i + 1)$, j steps to the previous one $(j - 1)$.

QUICK SORT

Algorithm stops, when i becomes greater than j.

After partition, all values before i-th element are less or equal than the pivot and all values after j-th element are greater or equal to the pivot.

QUICK SORT

Example:

Given array of n integers to sort:

40	20	10	80	60	50	7	30	100
i								j

PARTITIONING ARRAY

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

QUICK SORT

Example: Given array of n integers to sort:

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
   Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

i

j

too_big_index

too_small_index

QUICK SORT

Example: Given array of n integers to sort:

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
   Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

40	50	60	80	100

i
j
too_big_index
too_small_index

7	20	10	30

7	10	20	30	40	50	60	80	100

QUICK SORT

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
   Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
i								j

too_big_index

too_small_index

HEAP SORT

Stage 1: Construct a heap for a given list of n keys

Stage 2: Repeat operation of root removal n-1 times:

Exchange keys in the root and in the last (rightmost) leaf

Decrease heap size by 1

If necessary, swap new root with larger child until the heap condition holds

HEAP SORT

Example: 2, 9, 7, 6, 5, 8

Bottom level - right side

9	6	8	2	5	7
7	6	8	2	5	9
8	6	7	2	5	9
7	6	5	2	8	9
2	6	5	7	8	9
6	2	5	7	8	9
5	2	6	7	8	9
2	5	6	7	8	9

5	6	7	8	9

HEAP SORT

Example: Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)	Stage 2 (root/max removal)
2 9 7 6 5 8	9 6 8 2 5 7
2 9 8 6 5 7	7 6 8 2 5 | 9
2 9 8 6 5 7	8 6 7 2 5 | 9
9 2 8 6 5 7	5 6 7 2 | 8 9
9 6 8 2 5 7	7 6 5 2 | 8 9
	2 6 5 | 7 8 9
	6 2 5 | 7 8 9
	5 2 | 6 7 8 9
	5 2 | 6 7 8 9
	2 | 5 6 7 8 9

HEAP SORT

Example: 10 3 6 5 8 10 11

Bottom level - right side

11	8	10	5	3	10	6
6	8	10	5	6	10	11
10	8	10	5	3	6	11
6	8	10	5	3	10	11
10	8	6	5	3	10	11
3	8	6	5	10	10	11
8	5	6	3	10	10	11
3	5	6	8	10	10	11
6	5	3	8	10	10	11
3	5	6	8	10	10	11
5	3	6	8	10	10	11
3	5	6	8	10	10	11

5	6	8	10	10	11

SEATWORK

ASCENDING ORDER

BUBBLE | INSERTION | SELECTION | MERGE | QUICKSORT | HEAPSORT

Given: A[ ] = {3, 5, 7, -1, 15, 6}

3	5	7	-1	15	6

Filename : Section_Lastname_Sorting.jpg