GRAPHS

{ UNIT-4

INTRODUCTION

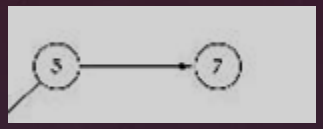
- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices
- A graph *G* is defined as follows:

$$G=(V,E)$$

V(G): a finite, nonempty set of vertices

E(G): a set of edges e (pairs of vertices) denoted by e=[u,v]

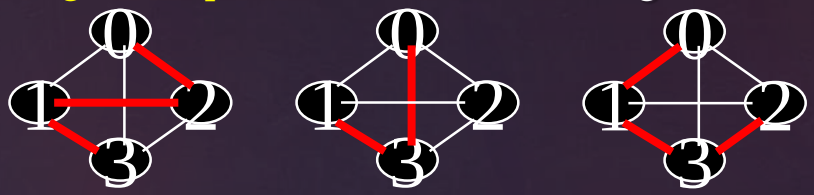
- Adjacent nodes: two nodes are adjacent if they are connected by an edge



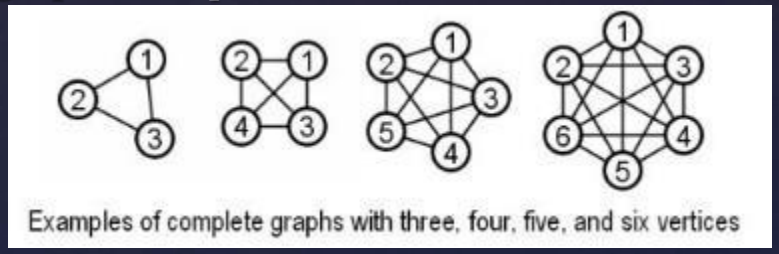
5 is adjacent to 7
7 is adjacent from 5

If ($v0$, $v1$) is an edge in an undirected graph,
$v0$ and $v1$ are adjacent
The edge ($v0$, $v1$) is incident on vertices $v0$ and $v1$

- Path: a sequence of vertices that connect two nodes in a graph
- The length of a path is the number of edges on it.



- **simple path (simple directed path)**: a path in which all vertices, except possibly the first and the last, are distinct.
- A cycle is a simple path in which the first and the last vertices are the same.
- **Simple cycle**: cycle that does not pass through other vertices more than once
- **Complete graph**: each pair of distinct vertices has an edge between them



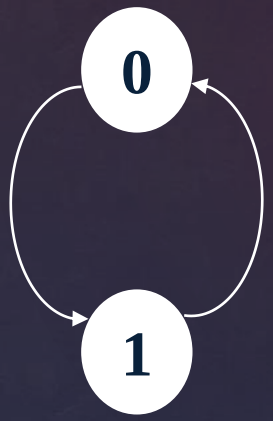
Examples of complete graphs with three, four, five, and six vertices

GRAPH THEORY TERMINOLOGY

GRAPH THEORY TERMINOLOGY

- Connected graph
  - In an undirected graph G, two vertices, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
  - An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j
- Strongly Connected Component
  - A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i

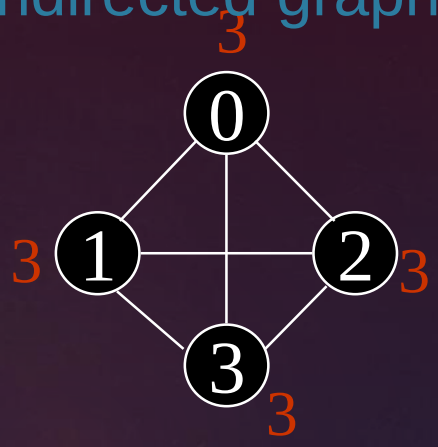
strongly connected component



GRAPH THEORY TERMINOLOGY

- Degree
  - The degree of a vertex is the number of edges incident to that vertex.
- For directed graph
  - in-degree (v) : the number of edges that have v as the head
  - out-degree (v) : the number of edges that have v as the tail

undirected graph



G_1

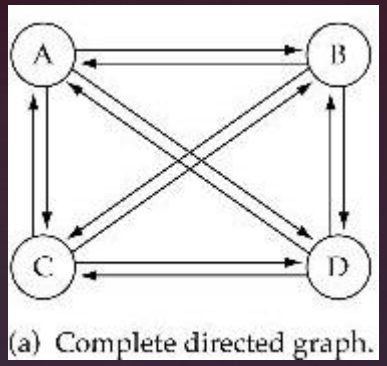
directed graph

in:1, out: 1

in: 1, out: 2

in: 1, out: 0

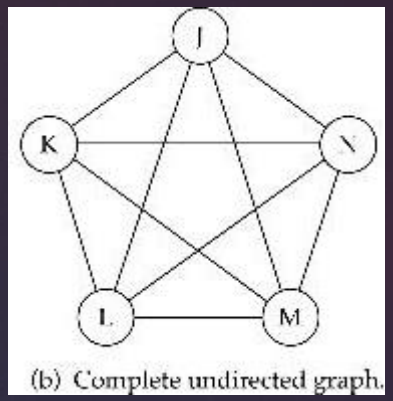
G_3



(a) Complete directed graph.

- What is the number of edges in a complete directed graph with N vertices?

  *N * (N-1)*



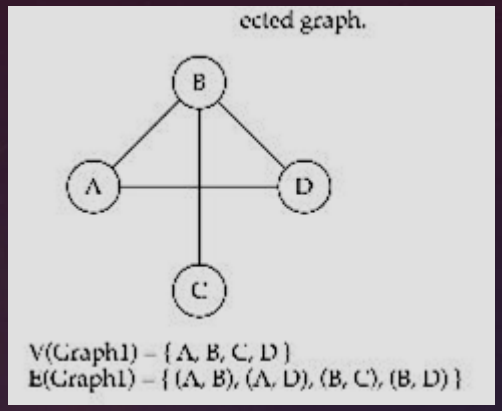
(b) Complete undirected graph.

- What is the number of edges in a complete undirected graph with N vertices?

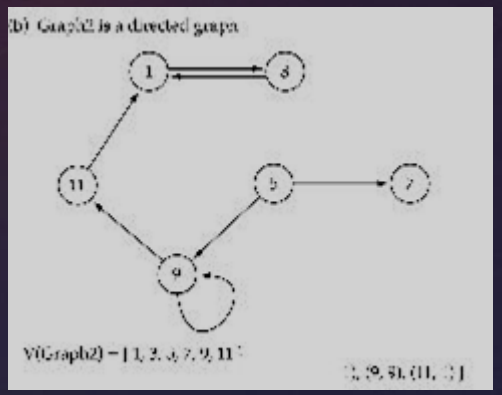
  *N * (N-1) / 2*

GRAPH THEORY TERMINOLOGY

- When the edges in a graph have no direction, the graph is called *undirected*

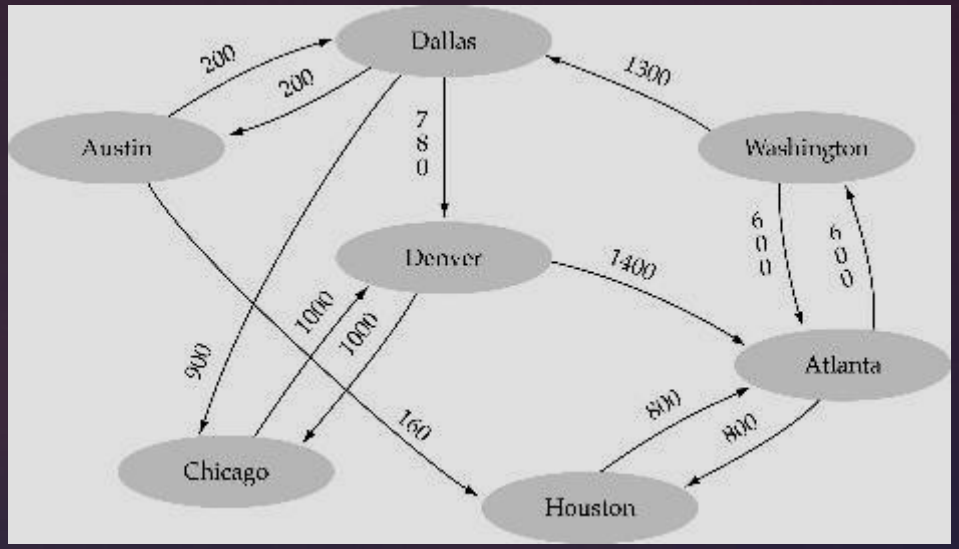


- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

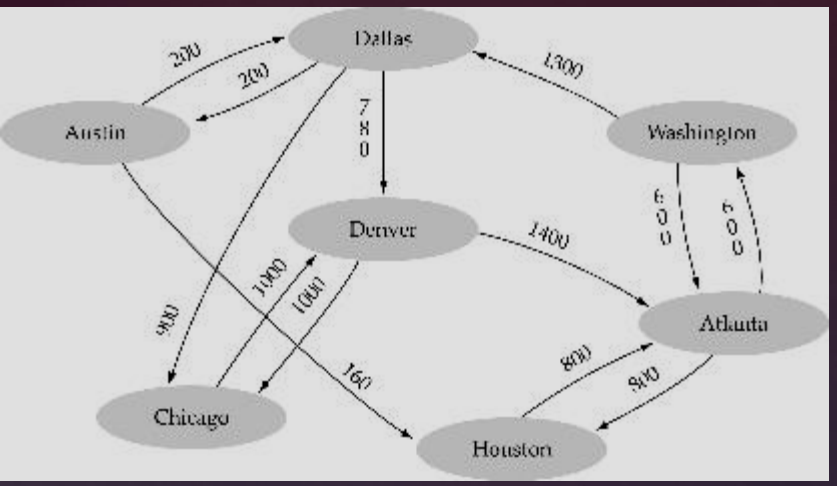


DIRECTED AND UNDIRECTED GRAPH

- Weighted graph: a graph in which each edge carries a value



WEIGHTED GRAPH



- A 1D array is used to represent the vertices
- A 2D array (adjacency matrix) is used to represent the edges

.vertices

[0]	"Atlanta"
[1]	"Austin"
[2]	"Chicago"
[3]	"Dallas"
[4]	"Denver"
[5]	"Houston"
[6]	"Washington"

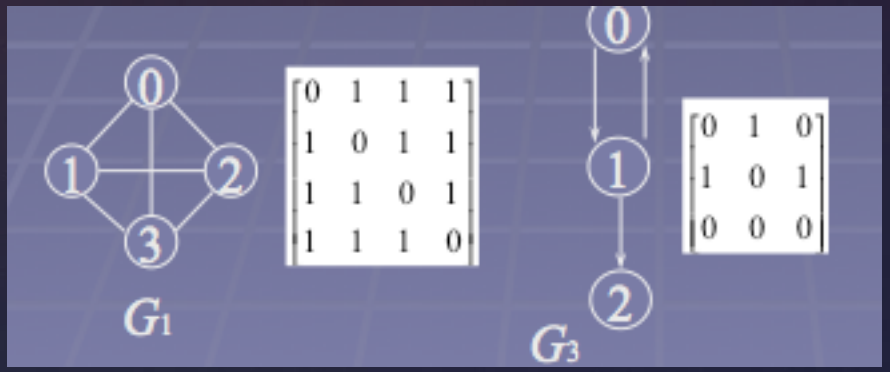
.edges

[0]	0	0	0	0	0	800	600
[1]	0	0	0	200	0	160	0
[2]	0	0	0	0	1000	0	0
[3]	0	200	900	0	780	0	0
[4]	1400	0	1000	0	0	0	0
[5]	800	0	0	0	0	0	0
[6]	600	0	0	1300	0	0	0

ARRAY BASED REPRESENTATION OF GRAPHS

SEQUENTIAL REPRESENTATION OF GRAPHS

- Adjacency Matrix
  - Let $G = (V,E)$ be a graph with n vertices.
  - The adjacency matrix of G is a two-dimensional n x n array, say $A=(a_{ij})$
  - If the edge (v_i, v_j) is(not) in $E(G)$, a_{ij}=1(0)
  - Matrix A is called BIT MATRIX



- Merits of Adjacency Matrix
  - For an undirected graph, the degree of any vertex, *i*, is its row sum.
  - For a directed graph, the row sum is the out-degree, while the column sum is the in-degree.

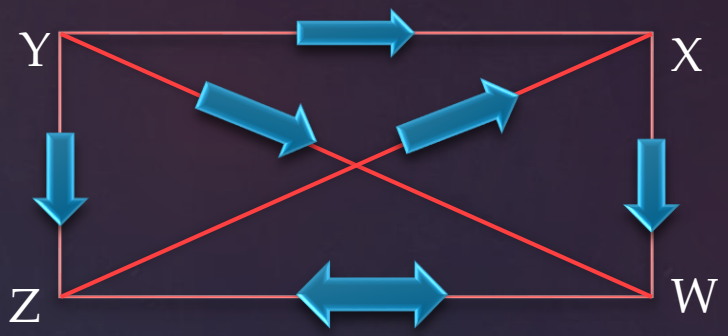
SEQUENTIAL REPRESENTATION OF GRAPHS

- Path Matrix
- Let G be a simple directed graph with m nodes, v_1, v_2-------v_m. The path matrix is m-square matrix $P=(p_{ij})$ defined as follows:

$P_{ij} = 1$; if there is a path from v_i to v_j

0 ; otherwise

Consider the following graph:



The adjacency matrix A is:

X	**0**	**0**	**0**	**1**
Y	1	0	1	1
Z	1	0	0	1
W	0	0	1	0
	X	**Y**	**Z**	**W**

SEQUENTIAL REPRESENTATION OF GRAPHS

- The powers A^2,A^3 and A^4 for the matrix A are:
- A^2 considers paths of length 2
- A^3 considers paths of length 3
- A^4 considers paths of length 4 and so on…

The matrix A^2 is

0	**0**	**1**	**0**
1	0	1	2
0	0	1	1
1	0	0	1

The matrix A^3 is

1	**0**	**0**	**1**
1	0	2	2
1	0	1	1
0	0	1	1

The matrix A^4 is

0	**0**	**1**	**1**
2	0	2	3
1	0	1	2
1	0	1	1

SEQUENTIAL REPRESENTATION OF GRAPHS

- If A is the adjacency matrix and P be the path matrix, then P_{ij}=1 if and only if there is a non zero number in the ij entry of the matrix.

  $Bm = A+A^2+A^3------A^m$

- The graph G in the above eg. contains 4 nodes, Therefore,

We obtain the following matrix B_4:

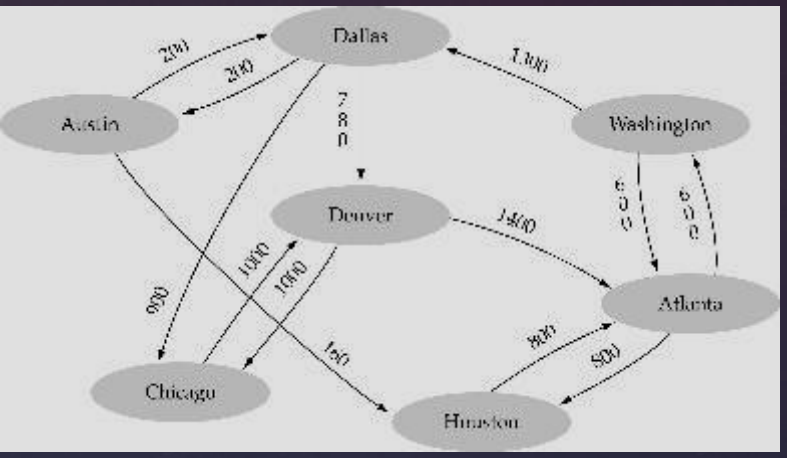
1	0	2	3
5	0	6	8
3	0	3	5
2	0	3	3

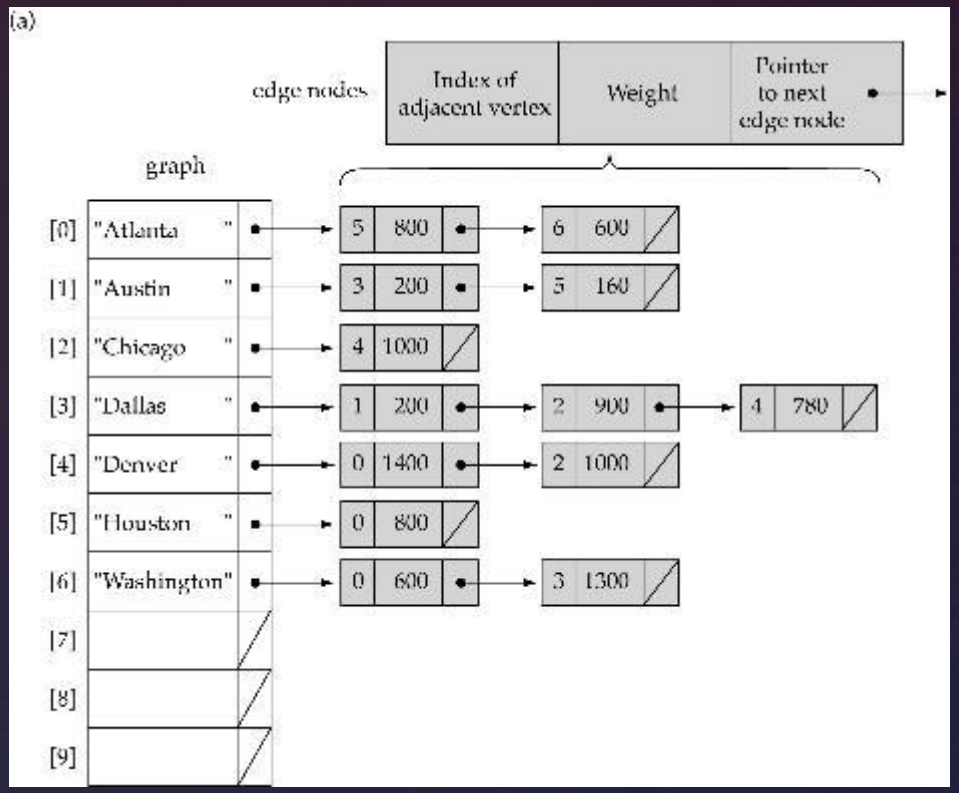
Replacing non zero entities in B_4, we get path matrix P:

1	0	1	1
1	0	1	1
1	0	1	1
1	0	1	1

SEQUENTIAL REPRESENTATION OF GRAPHS

- A 1D array is used to represent the vertices
- A list is used for each vertex v which contains the vertices which are adjacent from v (adjacency list)

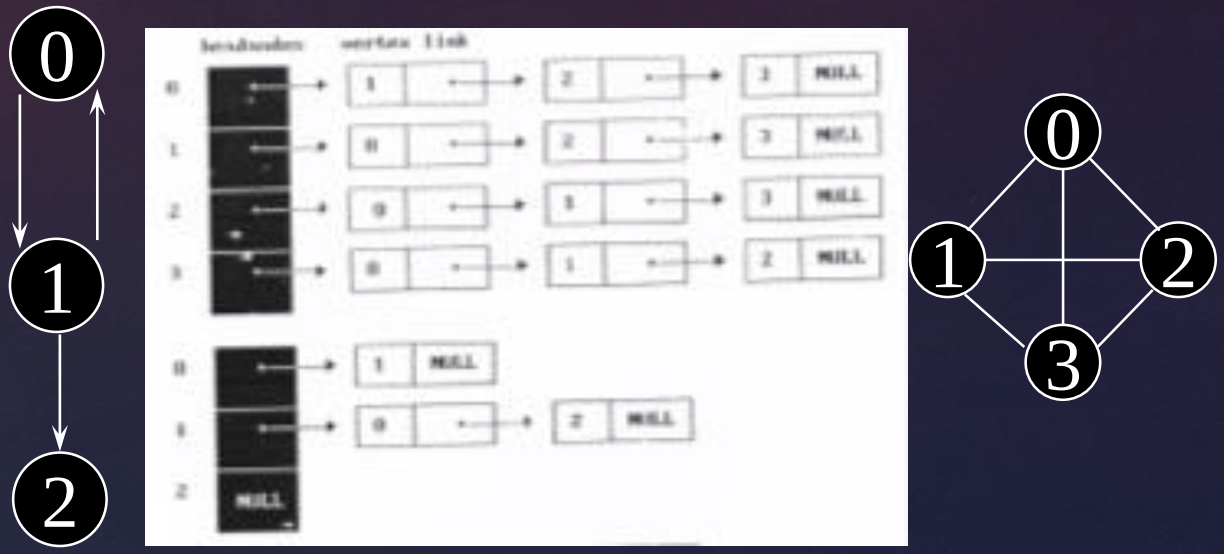




LINKED REPRESENTATION OF GRAPHS

- Adjacency lists (LINKED REPRESENTATION)
  - There is one list for each vertex in *G*. The nodes in list *i* represent the vertices that are adjacent from vertex *i*.

degree of a vertex in an undirected graph
No. of nodes in adjacency list
out-degree of a vertex in a directed graph
No. of nodes in its adjacency list



ADJACENCY LIST

- QUESTION 8.36

LINKED REPRESENTATION OF GRAPHS

TRAVERSING A GRAPH

- Many graph algorithms require a way to systematically examine the nodes and edges of a graph.
- Two algorithm that we will be discussing are :
  - BFS- uses a queue to hold nodes for future processing
  - DFS- uses a stack to hold nodes for future processing.

During execution of the algorithm each node N will have one of the three states:

STATUS=1 ; Ready state

STATUS=2 ; waiting state (on queue or stack)

STATUS=3 ; processed state.

BREADTH-FIRST SEARCH

In this algorithm, we first examine the start node say A. then examine all the neighbours of A, then examine all the neighbours of neighbours of A and so on.

We need to keep track of the neighbours and ensure that no node is processed more than once.

This algorithm enable us to find the shortest path from source to destination.

BFS_ALGORITHM

1. Initialise all nodes to ready state i.e STATUS=1.
2. Put starting node A in queue and change its STATUS to waiting state i.e STATUS=2
3. Repeat steps 4 and 5 until queue is empty
    4. Remove the front node of Queue and process it and change its STATUS=3.
    5. add to queue all the neighbours of this node and change their status, STATUS=2.

End of step 3 loop

6. Exit

- Question

BREADTH-FIRST SEARCH

DEPTH-FIRST SEARCH

In this algorithm, we first examine the start node say A. Then we examine each node along the path that begins at A i.e. we process a neighbour of A, then process the neighbour of neighbour of A and so on.

Once we reach a dead end e back track on this path and continue on a different path.

This algorithm is very similar to inorder traversal of a binary tree..

DFS_ALGORITHM

1. Initialise all nodes to ready state i.e STATUS=1.
2. Put starting node A on Stack and change its STATUS to waiting state i.e. STATUS=2
3. Repeat steps 4 and 5 until Stack is empty
    4. Pop the top node of Stack and process it and change its STATUS=3.
    5. Push to Stack all the neighbours of this node and change their status, STATUS=2.

End of step 3 loop

6. Exit

BREADTH-FIRST SEARCH

- Question

SHORTEST PATH ALGORITHM- DIJKSTRA'S ALGORITHM

- This algorithm finds the length of shortest path between two vertices.
- Consider a weighted graph G=(V,E)
- There are three things associated with every vertex of a graph:
  - Parent of a vertex; P(V)
  - Distance of the vertex; d(V)= W(U,V)+ d(U)
    - Where U and V are two vertices
    - W denotes the weight on the edge connecting these two vertices.
  - Color of the vertex, C(V) which can be:
    - White: C(V)='W' ; when the vertex is not traversed.
    - Grey: C(V)='G' ; when vertex is being processed.
    - Black: C(V)='B' ; when all adjacent vertices are found.

SHORTEST PATH ALGORITHM-DIJKSTRA'S ALGORITHM

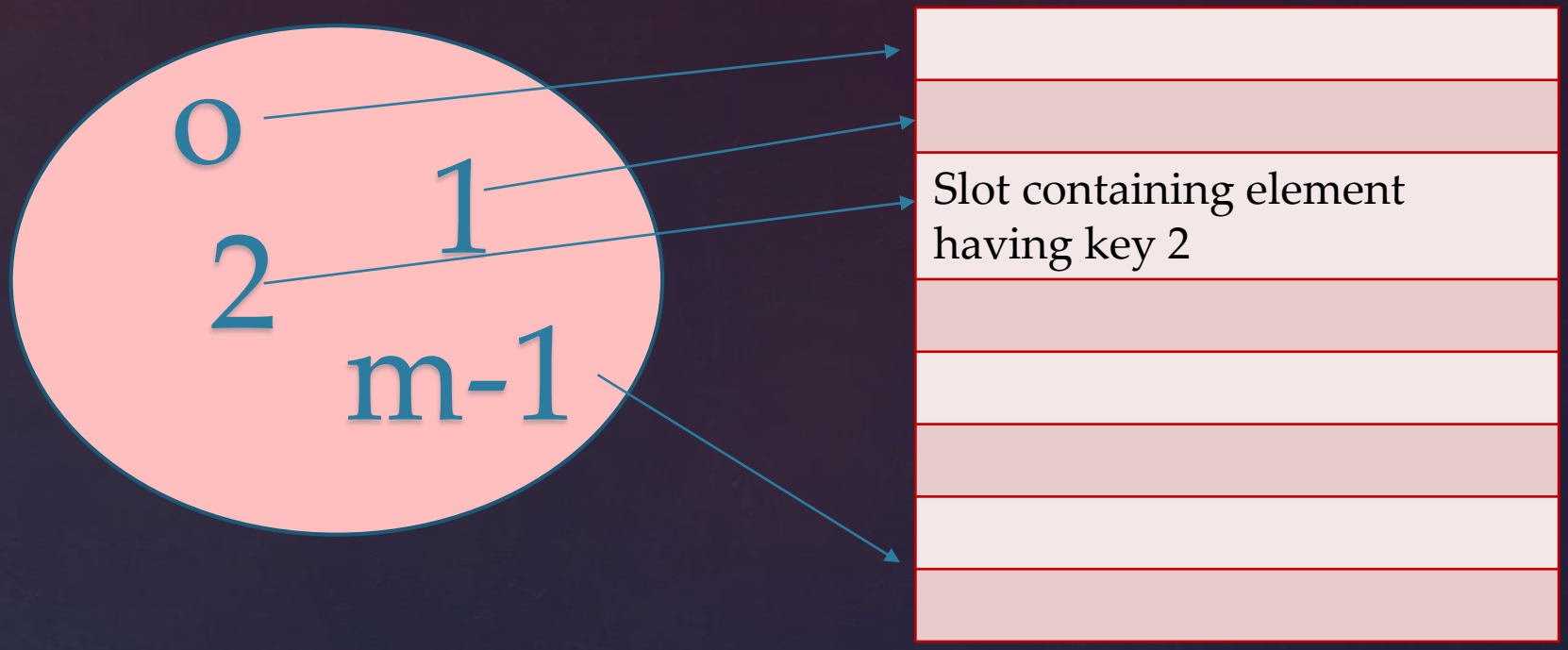
Ref: CORMEN

```
DIJKSTRA(G,S)
{
For all v belonging to V
        {
        C(v)='W'
        P(v)=nil
        d(v)=∞
        }
d(S)=0
Maintain a Queue Q and insert all the vertices in this queue
While(Q not empty)
{
u=extractmin(Q);
C(u)='G'
        for all (v that are adjacent to u)
        {
If (C(v)=='W')
{
If (d(v)>W(u,v)+d(u))
{
        d(v)=W(u,v)+d(u)
        P(v)=u
}}}
C(u)='B'
}
```

TABLES

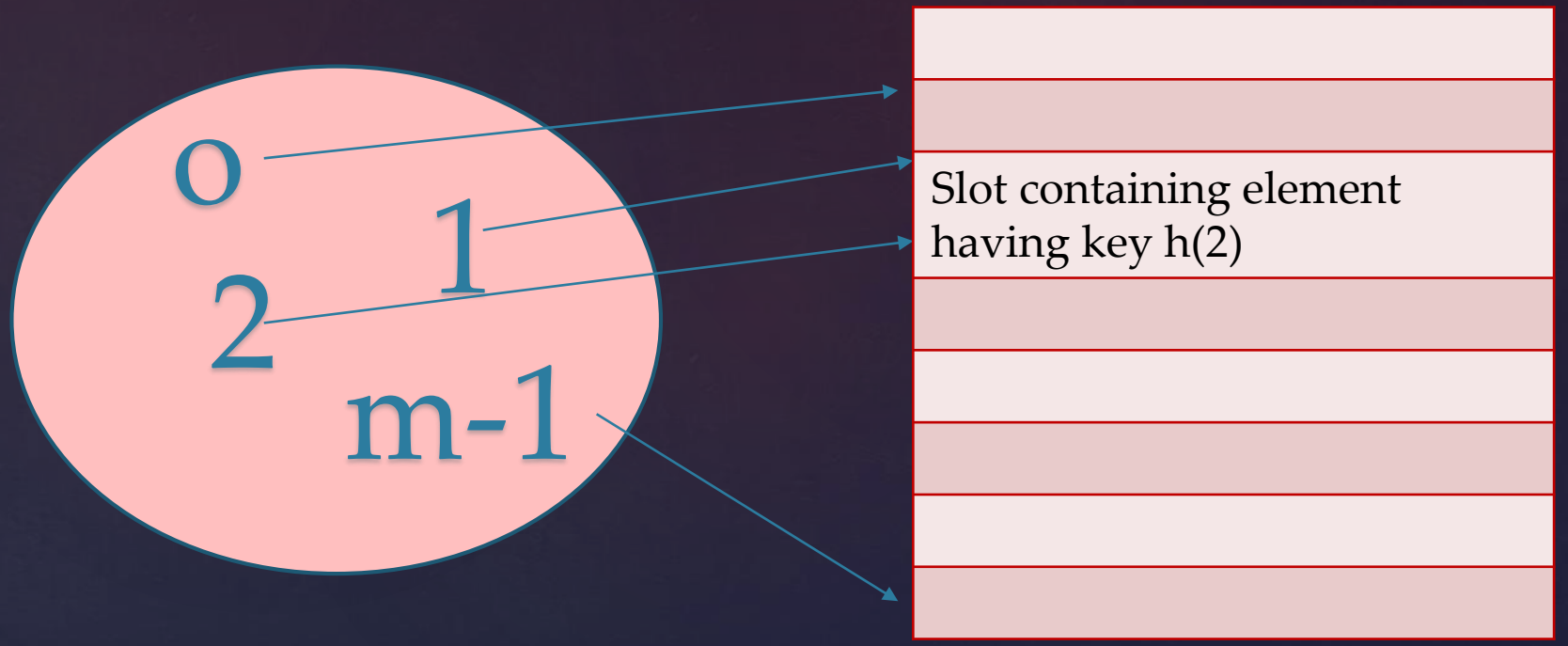
- TABLE: it is a data structure that is used to store information in tabular format.
- Two types of tables that we will be discussing are:
  - Direct address tables
  - Hash tables

- Consider a set of keys U={0,1,2,---------m-1}
- Consider a table T such that T[0,1,2--------m-1] and each position in Array T corresponds to a key in U.



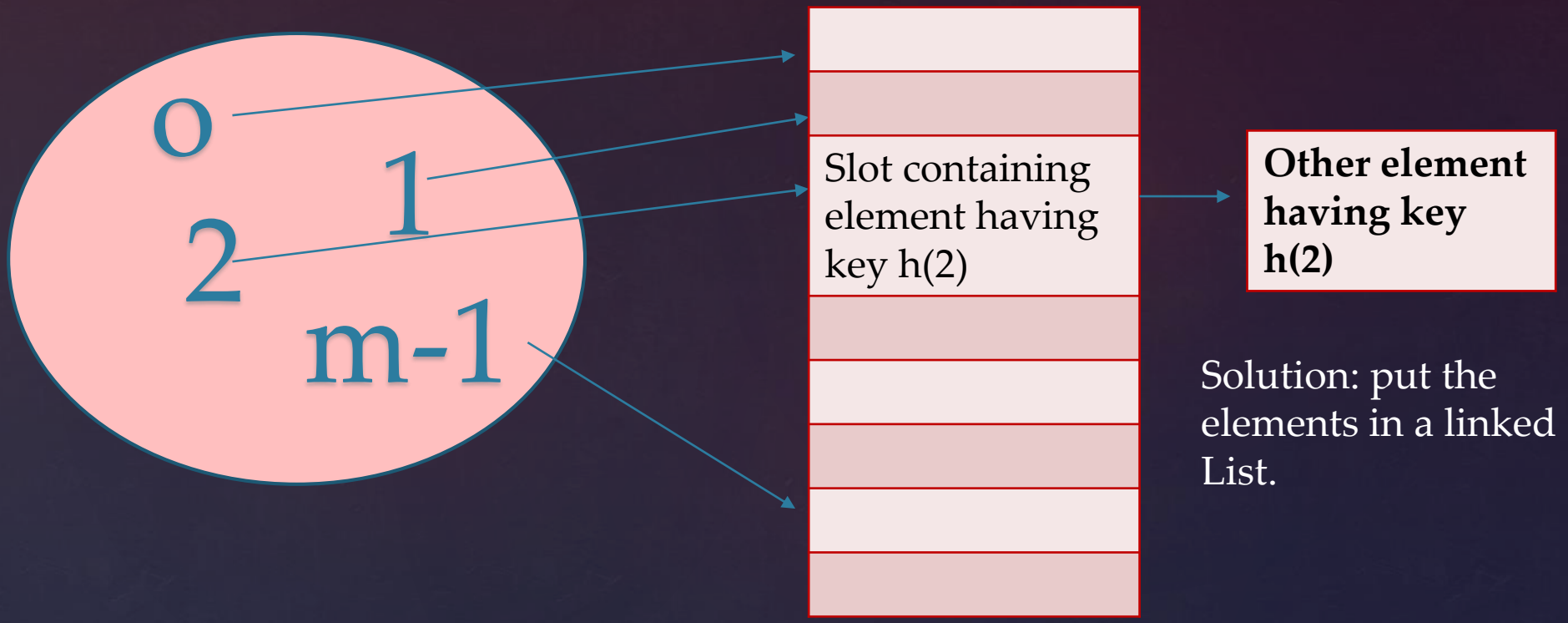
DIRECT ADDRESS TABLES

- Effective data structure for implementing dictionaries.
- Elements having key K is stored in slot h(k).



HASH TABLE

- Sometimes elements hash to the same slot. This causes collision.



COLLISION

SORTING ALGORITHMS

UNIT-6

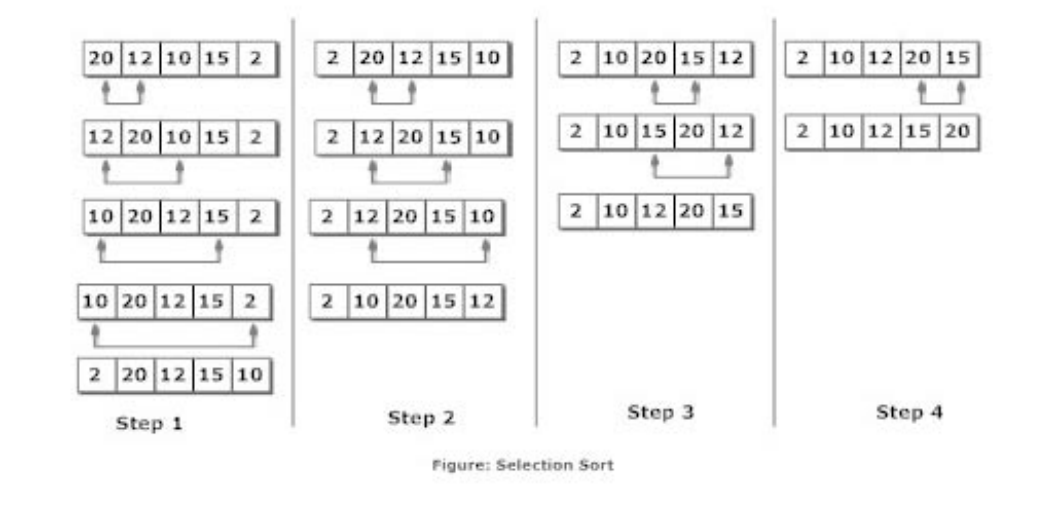


Figure: Selection Sort

Consider an array A of size N.
Let MIN store the minimum value in the array.
LOC stores the location of this element.

COMPLEXITY= $O(n^2)$

SELECTION(A,N)

Set MIN=A[0]
LOC=0

Repeat for j=1 to N-1
if MIN>A[j]
Then set MIN=A[j]
And LOC=j

Now interchange A[0] and A[LOC] by taking a temp variable i.e.

Temp=A[0]
A[0]=A[LOC]
A[LOC]=temp

exit

SELECTION SORT

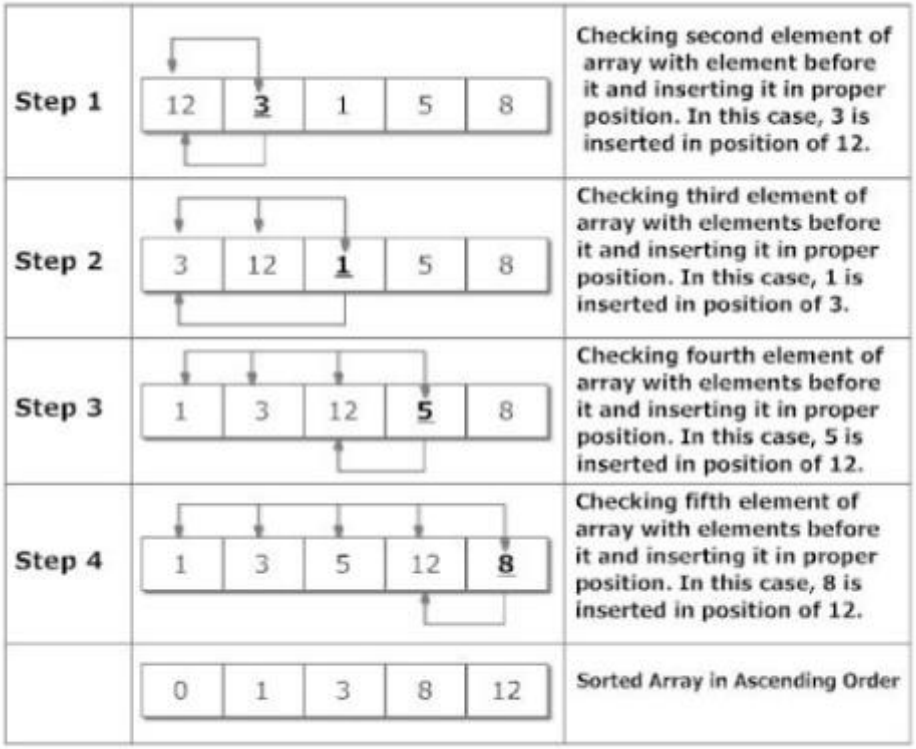


Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

INSERTION(A,N)

For (K=0 to N-1)
enter the elements of the array

For (i=1 to N-1)
temp=A[i]
PTR= i-1

While(temp<A[PTR])
A[PTR+1]=A[PTR]
PTR=PTR-1
End of while loop

Set A[PTR+1]=temp

exit

COMPLEXITY= $O(n^2)$

INSERTION SORT

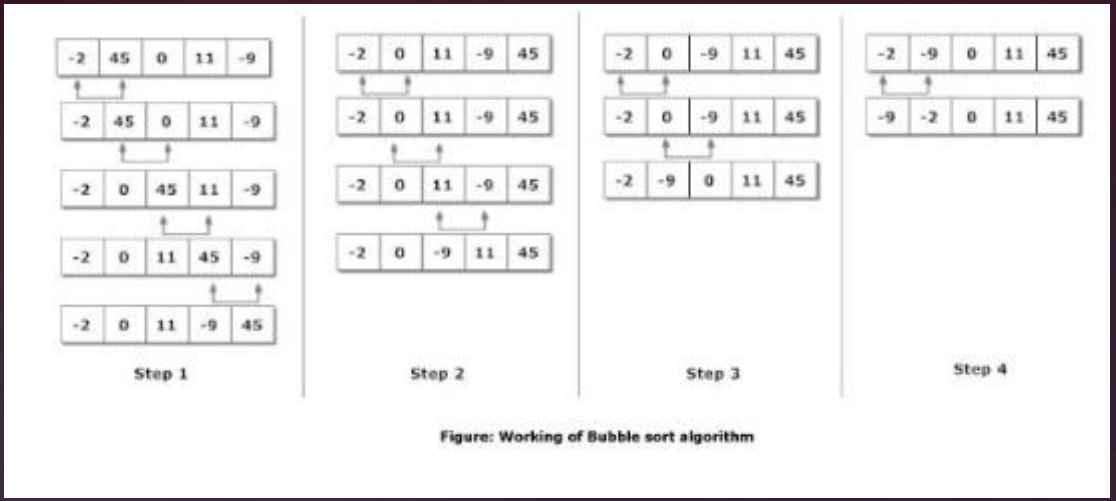


Figure: Working of Bubble sort algorithm

COMPLEXITY= $O(n^2)$

BUBBLESORT(A,N)

For (k=1 to N-1)

set PTR=0

while PTR<=N-k

if A[PTR]>A[PTR+1]

interchange them

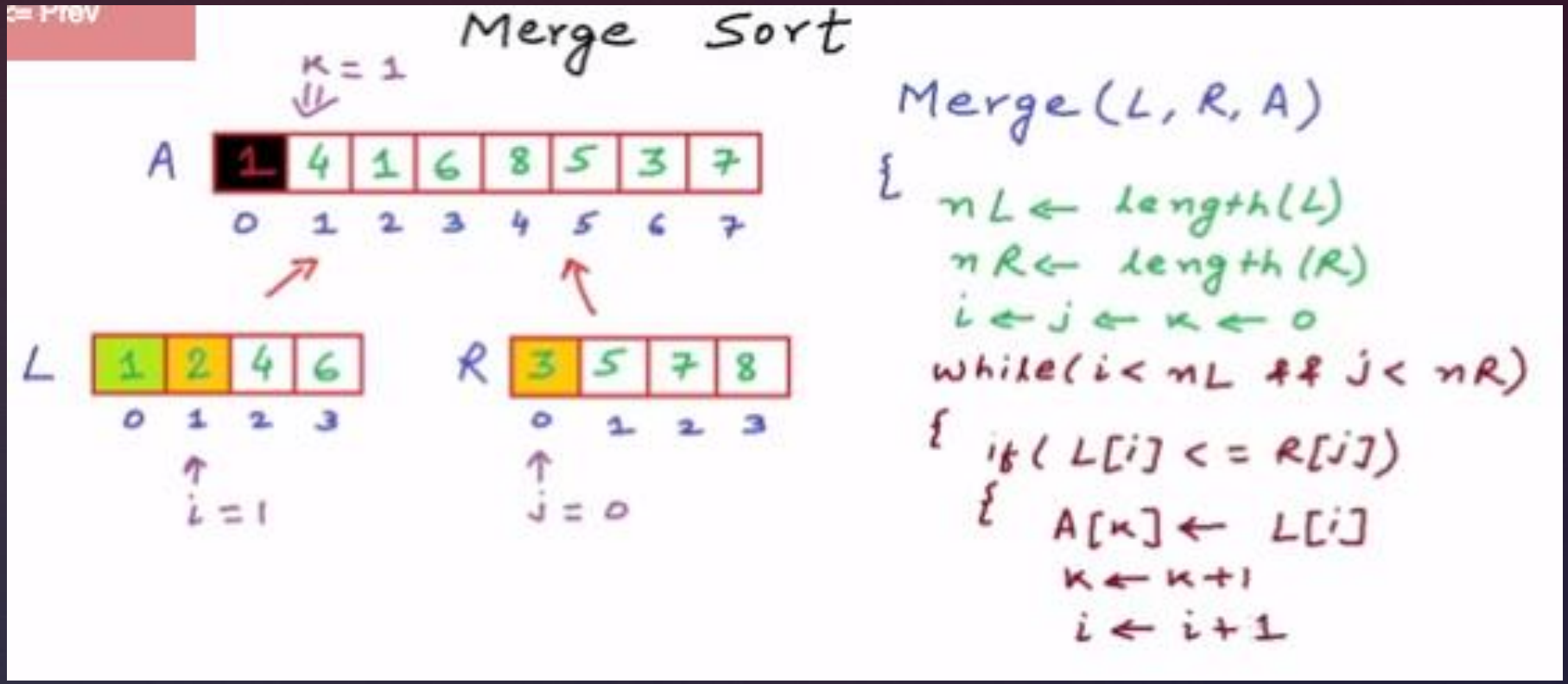
set PTR=PTR+1

end of while

End of for loop

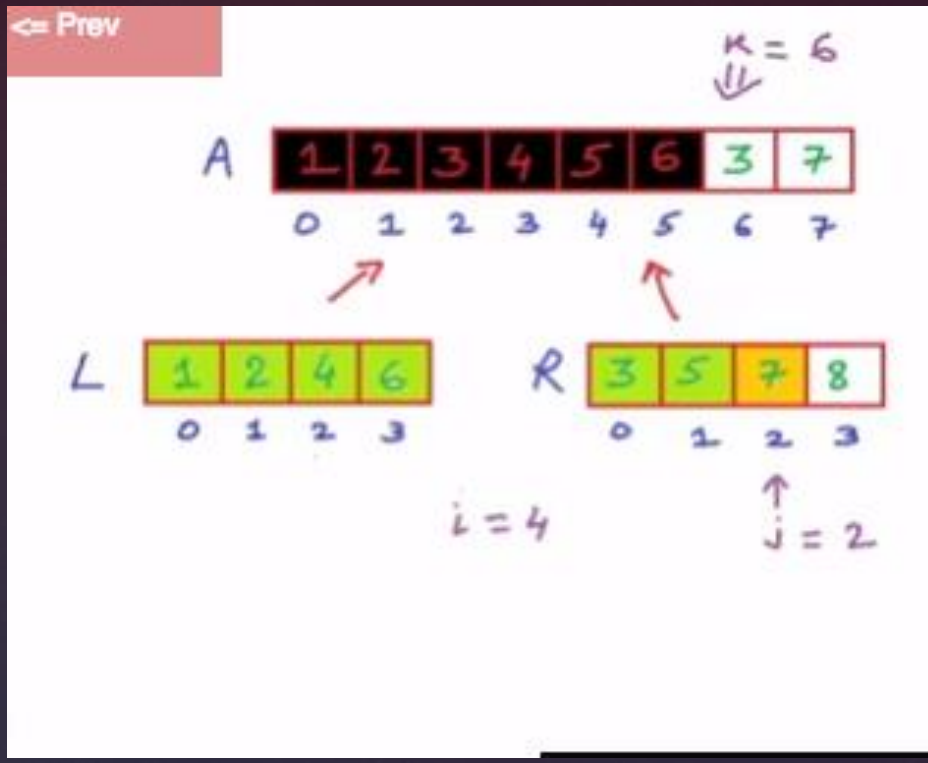
exit

BUBBLE SORT



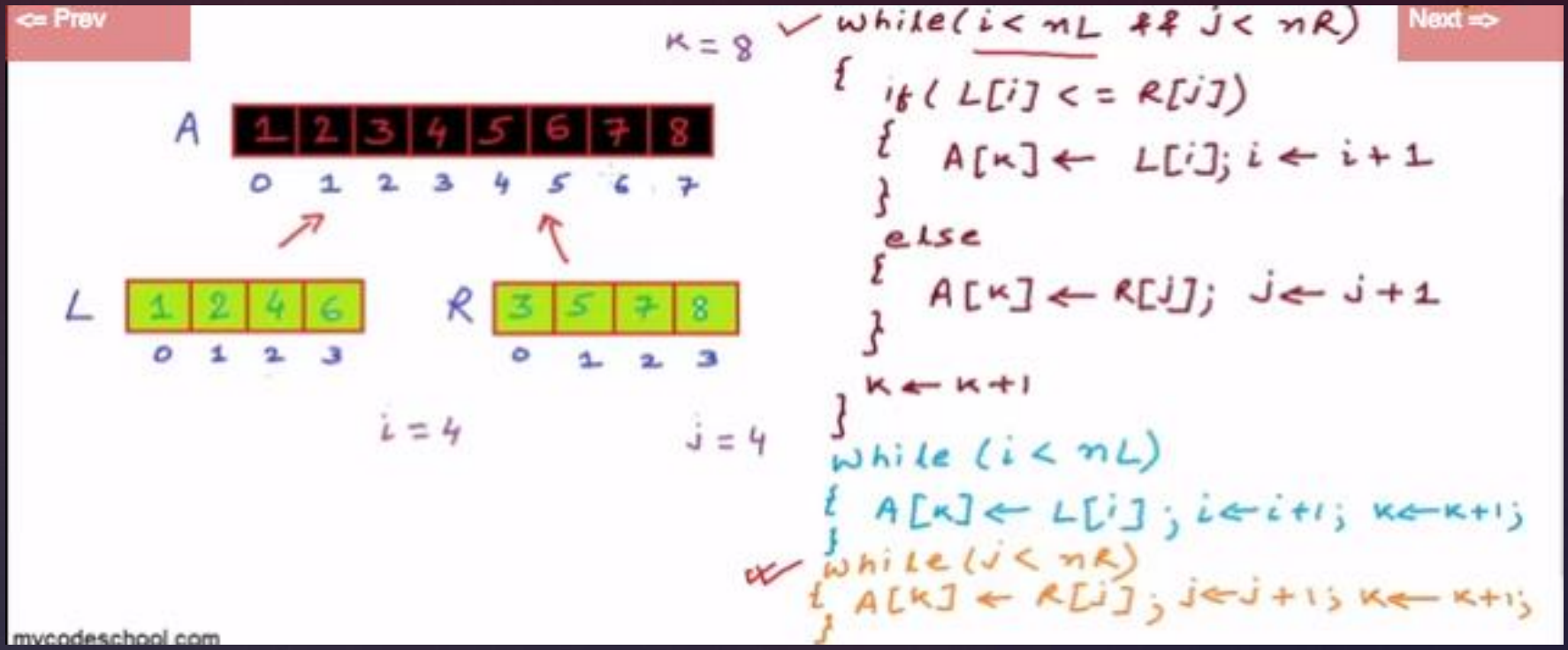
https://www.youtube.com/watch?v=TzeBrDU-JaY&noredirect=1

MERGE SORT

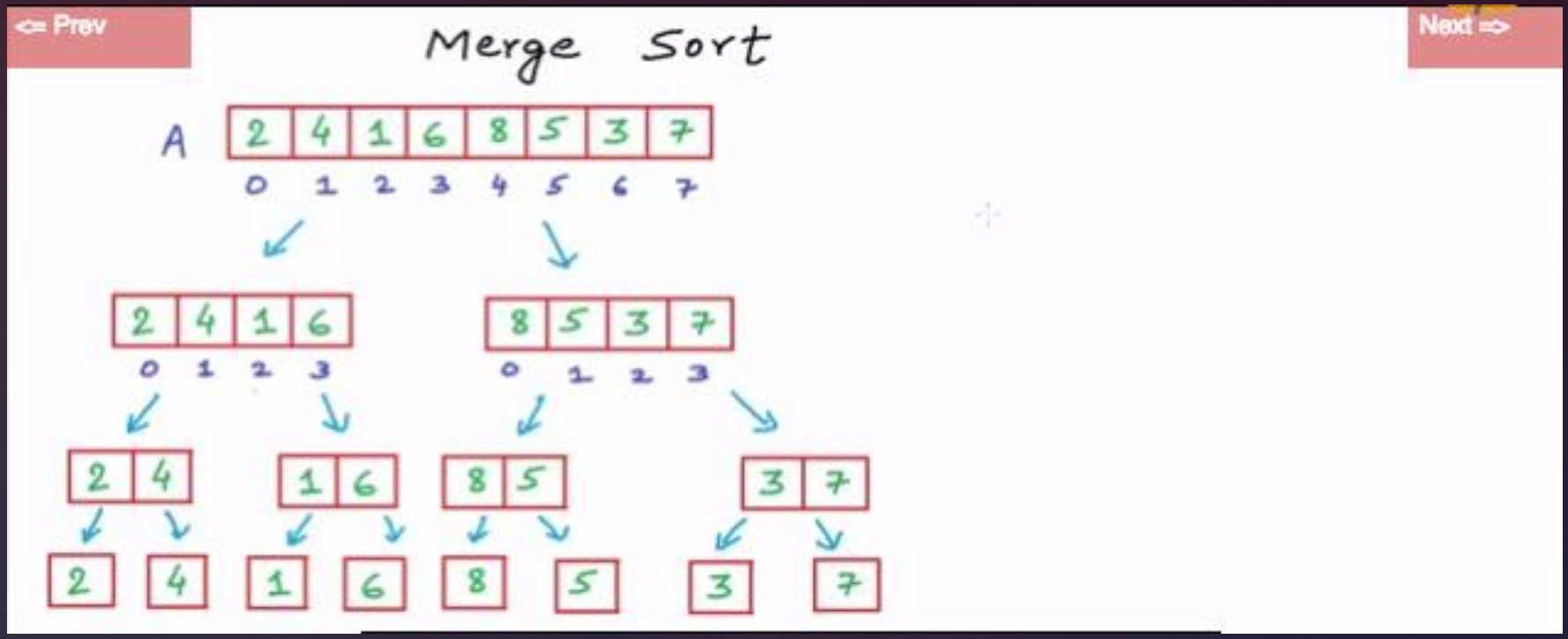



```
while(i < nL && j < nR)
{
  if( L[i] <= R[j])
  {
    A[k] ← L[i]; i ← i+1
  }
  else
  {
    A[k] ← R[j]; j ← j+1
  }
  k ← k+1
}
while (i < nL)
{ A[k] ← L[i]; i ← i+1; k ← k+1;
}
while(j < nR)
{ A[k] ← R[j]; j ← j+1; k ← k+1;
```

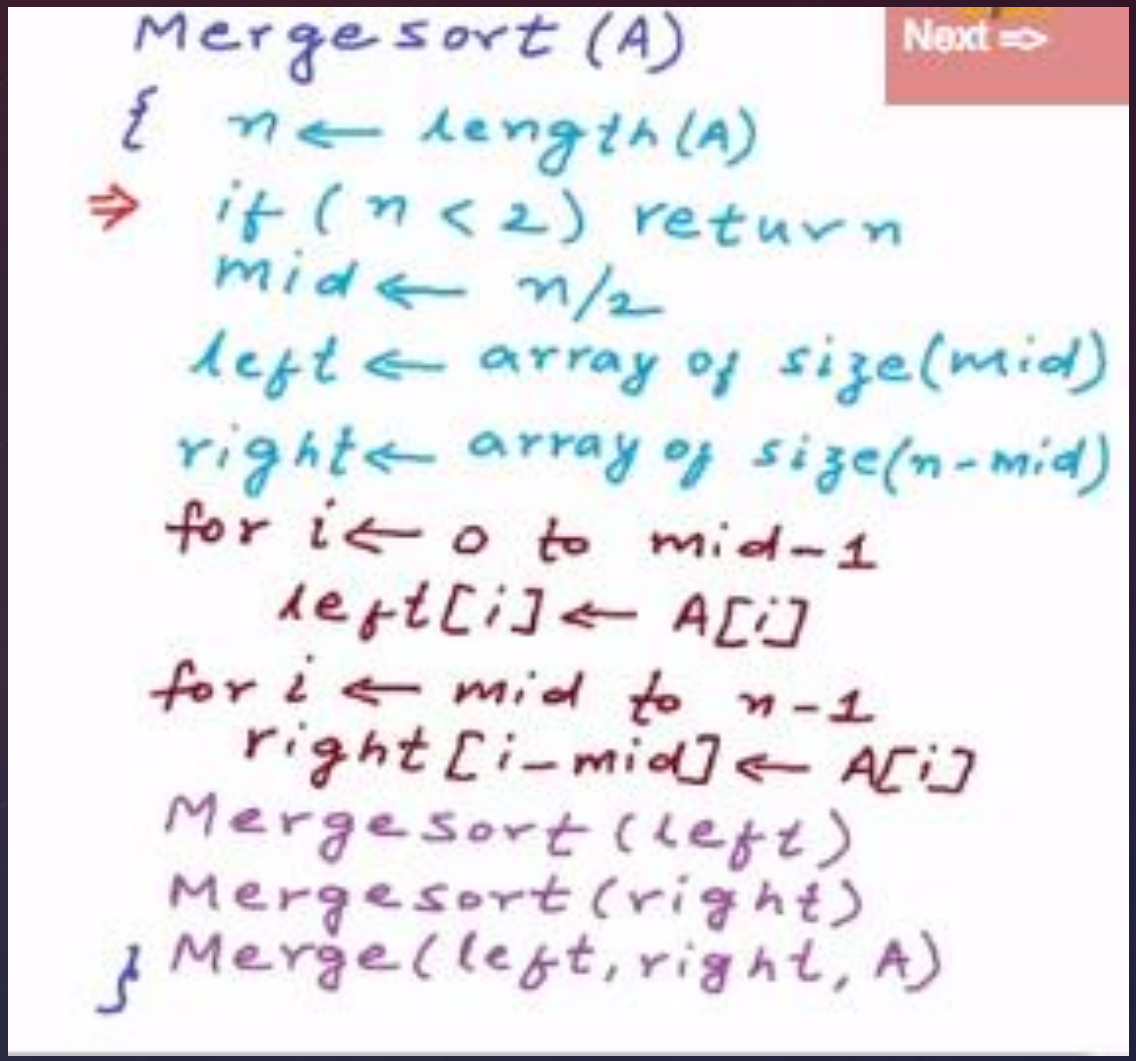
MERGE SORT


<= Prev
Next =>
k = 8
A
1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7
L
1 2 4 6
0 1 2 3
R
3 5 7 8
0 1 2 3
i = 4
j = 4
while(i < nL && j < nR)
{
if(L[i] <= R[j])
{ A[k] ← L[i]; i ← i+1
}
else
{ A[k] ← R[j]; j ← j+1
}
k ← k+1
}
while (i < nL)
{ A[k] ← L[i]; i ← i+1; k ← k+1;
}
while(j < nR)
{ A[k] ← R[j]; j ← j+1; k ← k+1;
}
mycodeschool.com

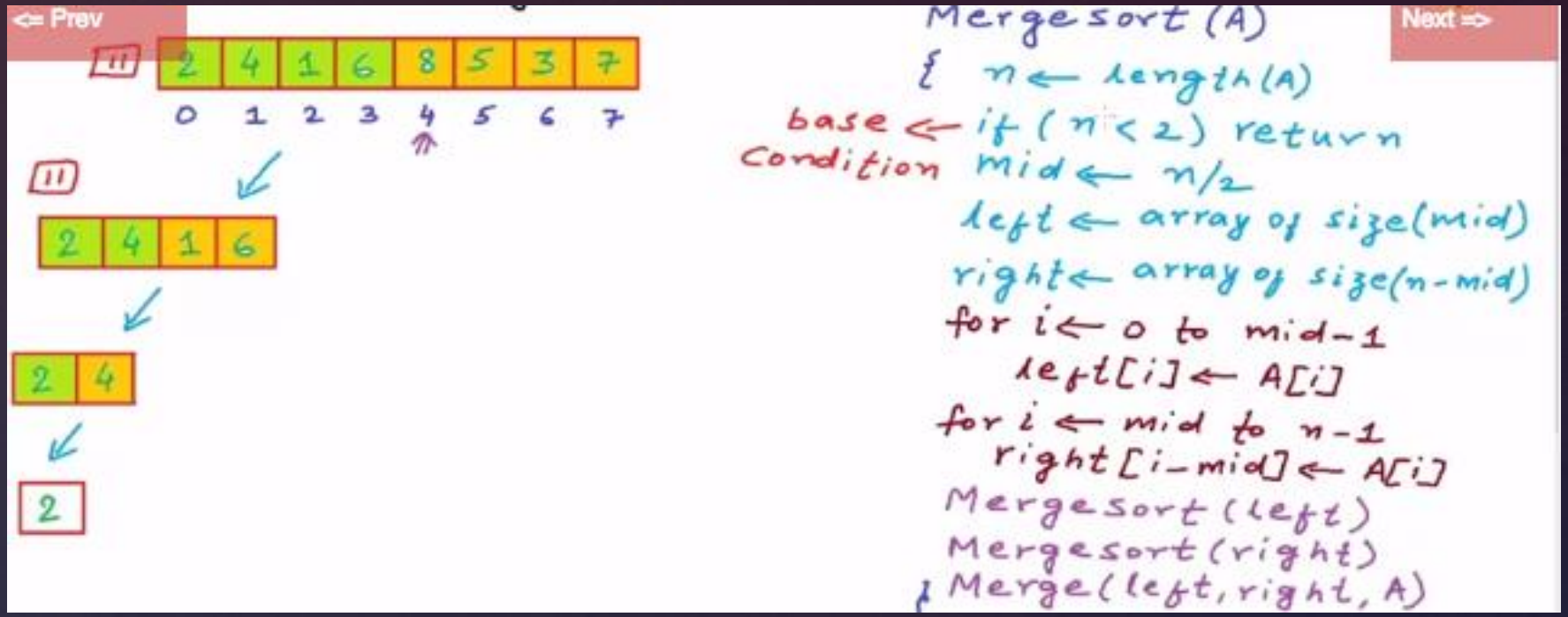
MERGE SORT


<= Prev
Merge Sort
Next =>
A
2 4 1 6 8 5 3 7
0 1 2 3 4 5 6 7
2 4 1 6
0 1 2 3
8 5 3 7
0 1 2 3
2 4
1 6
8 5
3 7
2
4
1
6
8
5
3
7

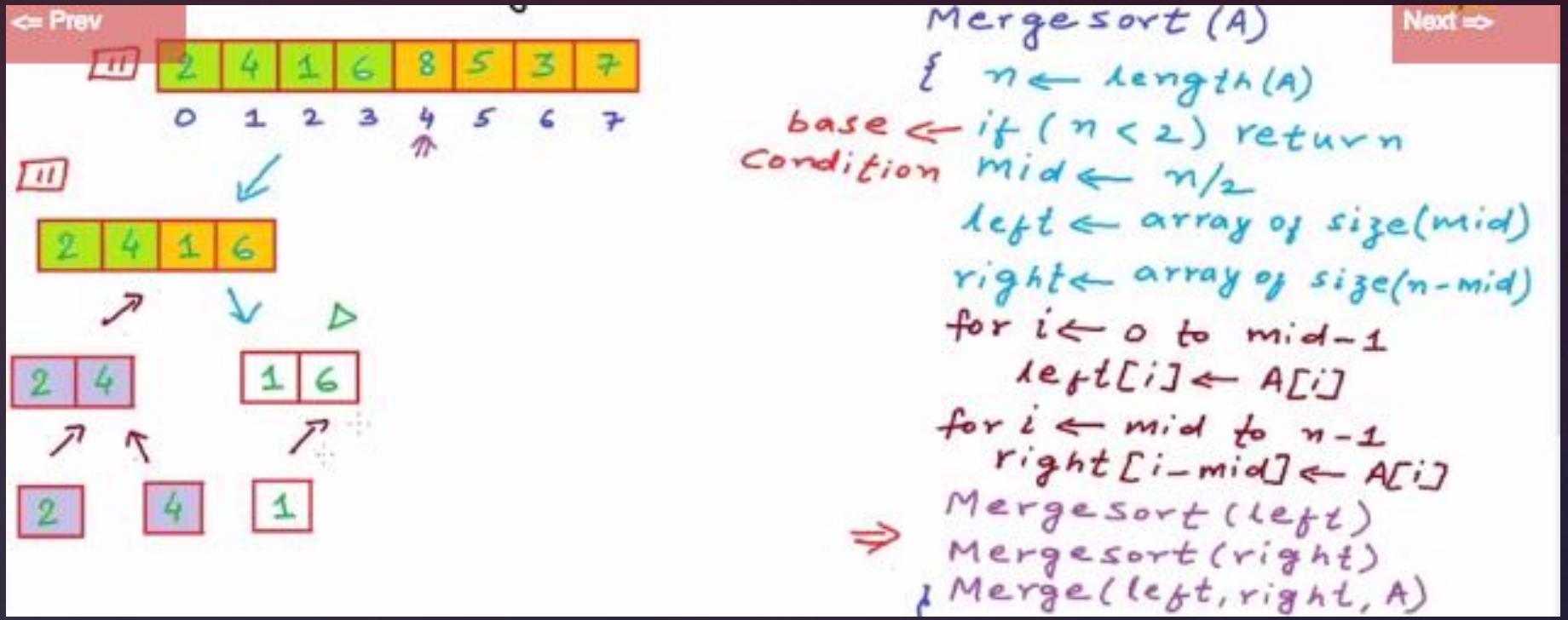
MERGE SORT


Next ⇨
Mergesort (A)
{ n ← length(A)
⇒ if (n < 2) return n
mid ← n/2
left ← array of size(mid)
right ← array of size(n-mid)
for i ← 0 to mid-1
left[i] ← A[i]
for i ← mid to n-1
right[i-mid] ← A[i]
Mergesort (left)
Mergesort (right)
} Merge (left, right, A)

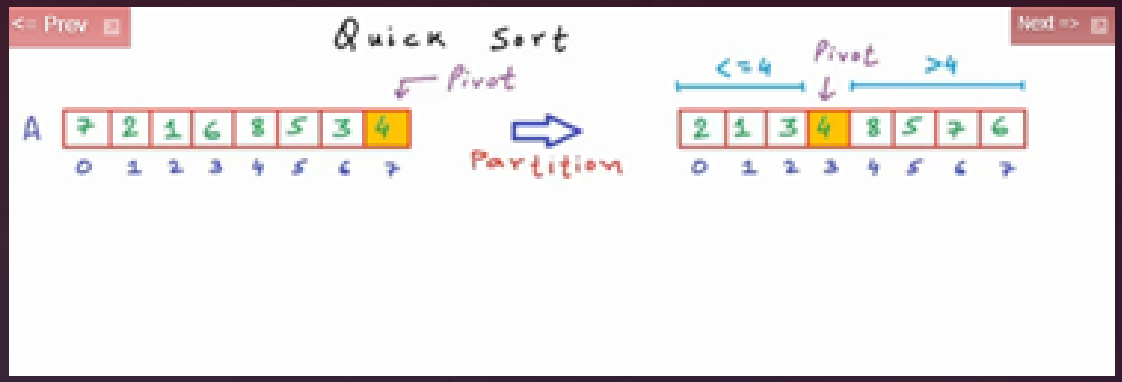
MERGE SORT

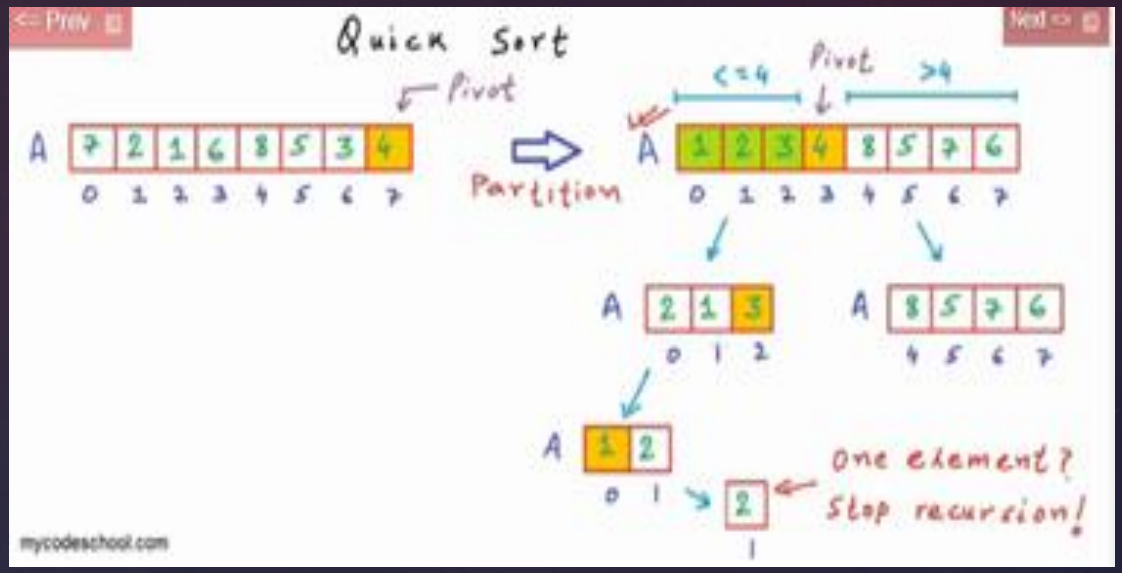

Prev
Next
2 4 1 6 8 5 3 7
0 1 2 3 4 5 6 7
2 4 1 6
2 4
2
Mergesort (A)
{ n ← length(A)
base Condition if (n < 2) return
Mid ← n/2
left ← array of size(mid)
right ← array of size(n-mid)
for i ← 0 to mid-1
left[i] ← A[i]
for i ← mid to n-1
right[i-mid] ← A[i]
Mergesort (left)
Mergesort (right)
Merge(left, right, A)

MERGE SORT

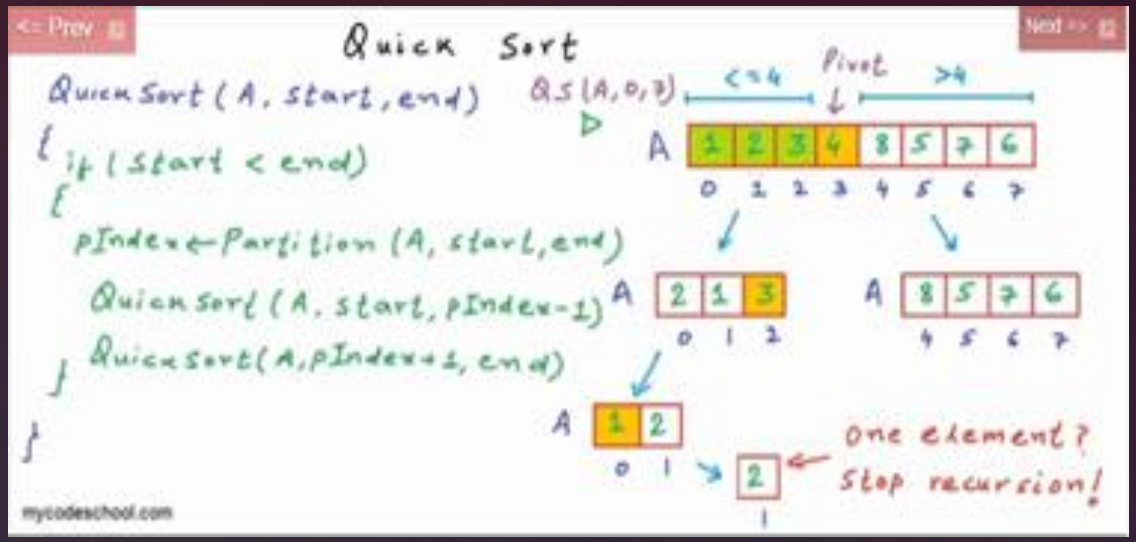

Prev
Next
2 4 1 6 8 5 3 7
0 1 2 3 4 5 6 7
2 4 1 6
2 4
1 6
2
4
1
Mergesort (A)
{ n ← length(A)
base condition if (n < 2) return n
Mid ← n/2
left ← array of size(mid)
right ← array of size(n-mid)
for i ← 0 to mid-1
left[i] ← A[i]
for i ← mid to n-1
right[i-mid] ← A[i]
Mergesort (left)
Mergesort (right)
Merge(left, right, A)

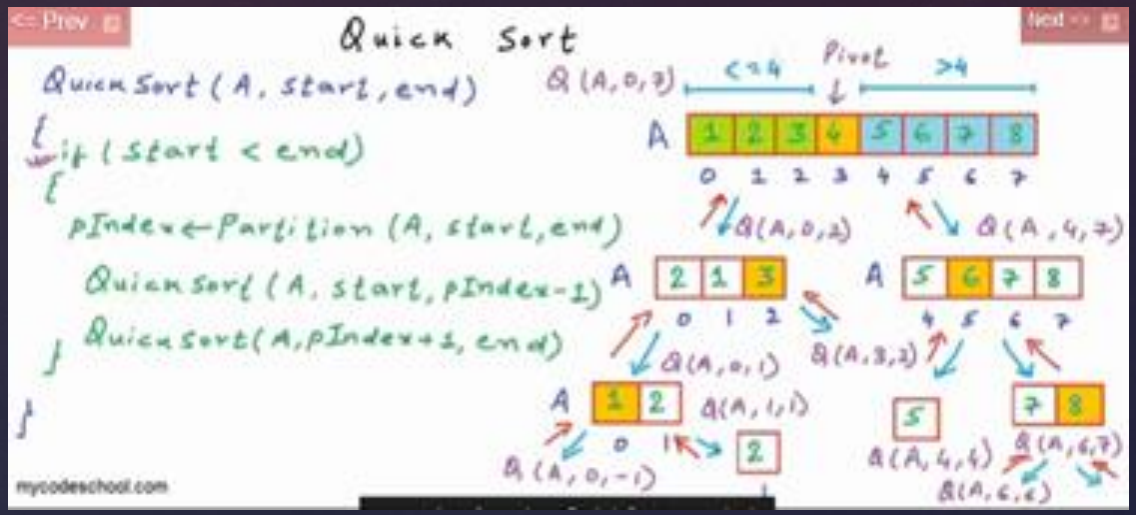
MERGE SORT


Quick Sort
Pivot
A
7 2 1 6 8 5 3 4
0 1 2 3 4 5 6 7
Partition
<=4
Pivot
>4
2 1 3 4 8 5 7 6
0 1 2 3 4 5 6 7

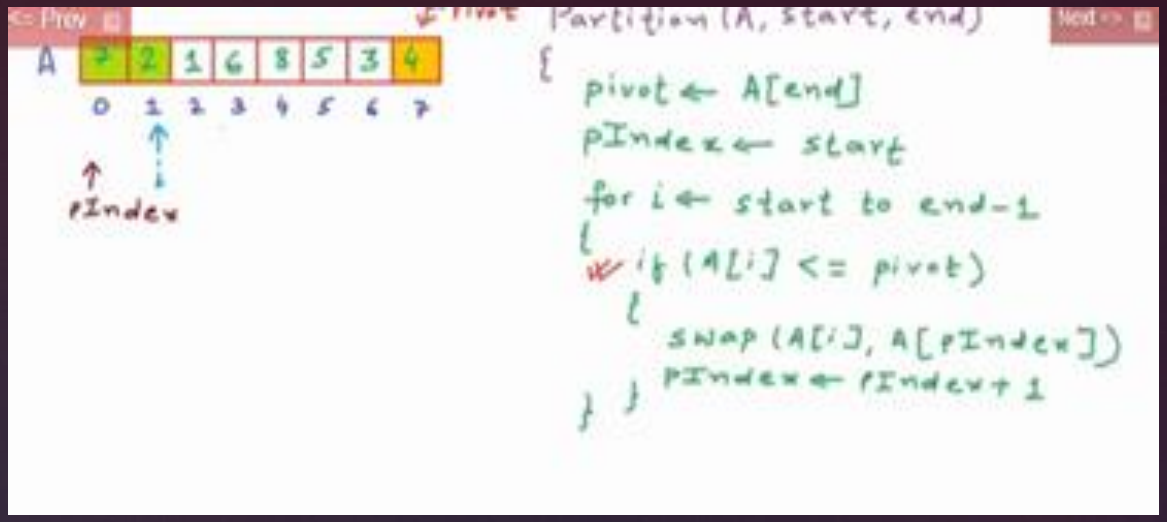

Quick Sort
Pivot
A
7 2 1 6 8 5 3 4
0 1 2 3 4 5 6 7
Partition
<=4
Pivot
>4
A
1 2 3 4 8 5 7 6
0 1 2 3 4 5 6 7
A
2 1 3
0 1 2
A
8 5 7 6
4 5 6 7
A
1 2
0 1
2
1
one element?
Stop recursion!
mycodeschool.com

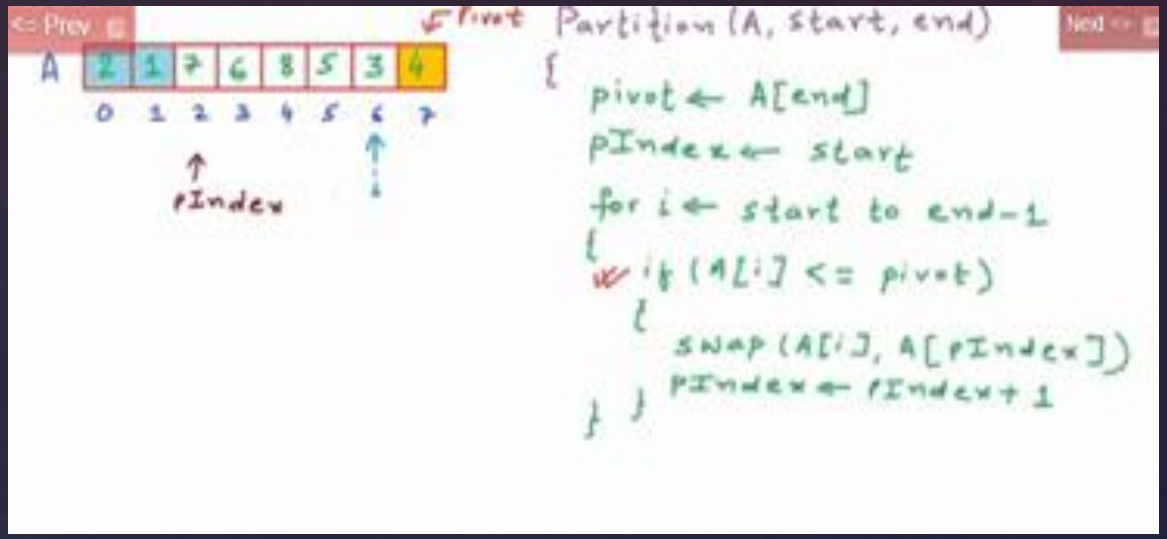
QUICK SORT


Quick Sort
QuickSort(A, start, end)
{
if (start < end)
{
pIndex ← Partition(A, start, end)
QuickSort(A, start, pIndex-1)
QuickSort(A, pIndex+1, end)
}
}
Q.S(A,0,7)
<=4
Pivot
>4
A
1 2 3 4 8 5 7 6
0 1 2 3 4 5 6 7
A
2 1 3
0 1 2
A
8 5 7 6
4 5 6 7
A
1 2
0 1
2
one element?
Stop recursion!
mycodeschool.com

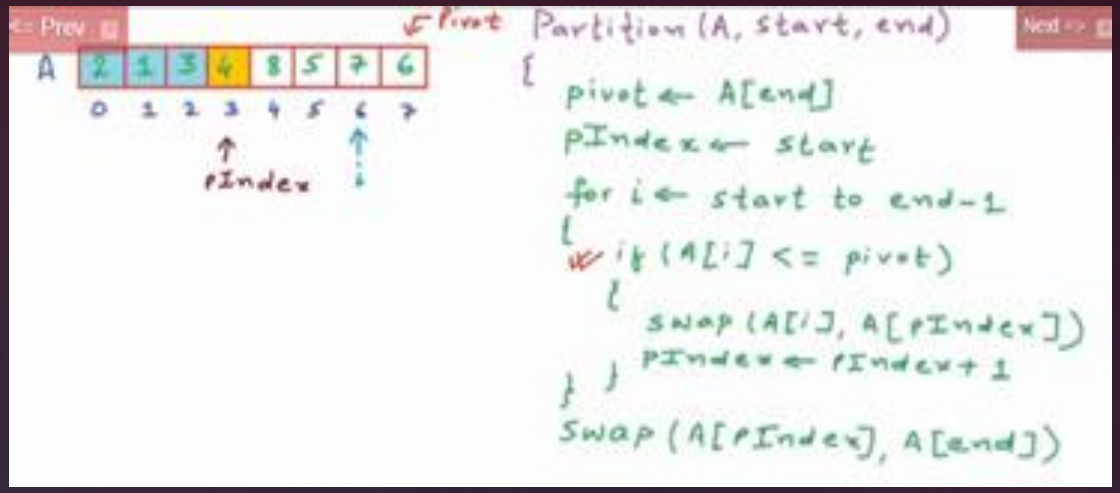

Quick Sort
QuickSort(A, start, end)
{
if (start < end)
{
pIndex ← Partition(A, start, end)
QuickSort(A, start, pIndex-1)
QuickSort(A, pIndex+1, end)
}
}
Q(A,0,7)
<=4
Pivot
>4
A
1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7
Q(A,0,2)
Q(A,4,7)
A
2 1 3
0 1 2
A
5 6 7 8
4 5 6 7
Q(A,0,1)
Q(A,3,2)
Q(A,6,7)
A
1 2
0 1
Q(A,1,1)
2
5
7 8
Q(A,0,-1)
Q(A,4,4)
Q(A,6,6)
mycodeschool.com

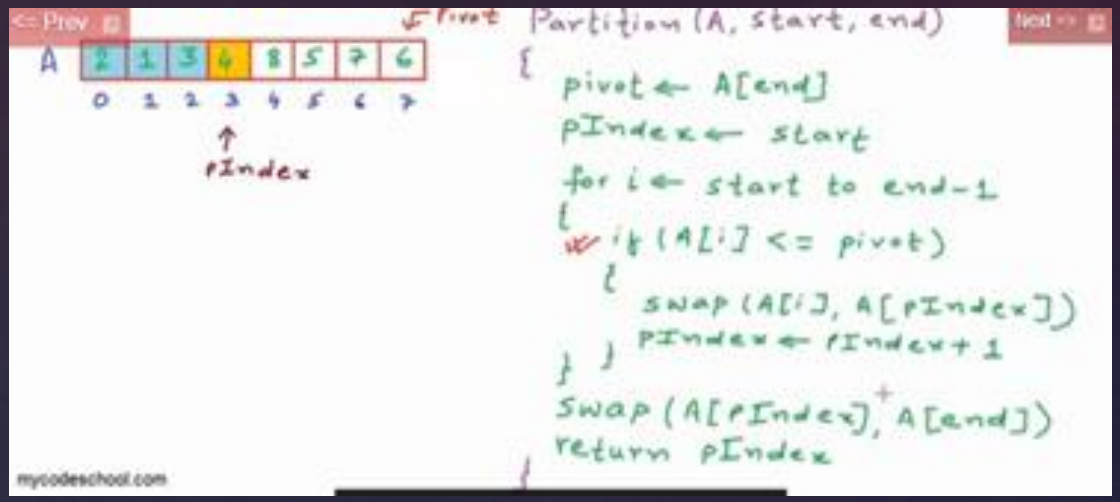
QUICK SORT


<= Prev
Next =>
A
7 2 1 6 8 5 3 4
0 1 2 3 4 5 6 7
pIndex
Partition (A, start, end)
{
pivot ← A[end]
pIndex ← start
for i ← start to end-1
{
if (A[i] <= pivot)
{
swap (A[i], A[pIndex])
pIndex ← pIndex + 1
} }


<= Prev
Pivot
Next =>
A
2 1 7 6 8 5 3 4
0 1 2 3 4 5 6 7
pIndex
Partition (A, start, end)
{
pivot ← A[end]
pIndex ← start
for i ← start to end-1
{
if (A[i] <= pivot)
{
swap (A[i], A[pIndex])
pIndex ← pIndex + 1
} }

QUICK SORT


<= Prev
Next =>
Pivot
A
2 1 3 4 8 5 7 6
0 1 2 3 4 5 6 7
pIndex
i
Partition (A, start, end)
{
pivot ← A[end]
pIndex ← start
for i ← start to end-1
{
if (A[i] <= pivot)
{
swap (A[i], A[pIndex])
pIndex ← pIndex + 1
}
}
Swap (A[pIndex], A[end])


<= Prev
Next =>
Pivot
A
2 1 3 4 8 5 7 6
0 1 2 3 4 5 6 7
pIndex
Partition (A, start, end)
{
pivot ← A[end]
pIndex ← start
for i ← start to end-1
{
if (A[i] <= pivot)
{
swap (A[i], A[pIndex])
pIndex ← pIndex + 1
}
}
Swap (A[pIndex], A[end])
return pIndex
mycodeschool.com

QUICK SORT