PROGRAMMING

FOR PROBLEM SOLVING

{ C LANGUAGE }

Module :- 5

Basic Algorithms

start

Module 5 :- Basic Algorithms

- Searching (linear search, binary search etc..)
- Basic sorting algorithms (bubble, insertion and selection)
- Finding roots of equation
- Notion of order of complexity through example programs (no formal definition required)

Searching

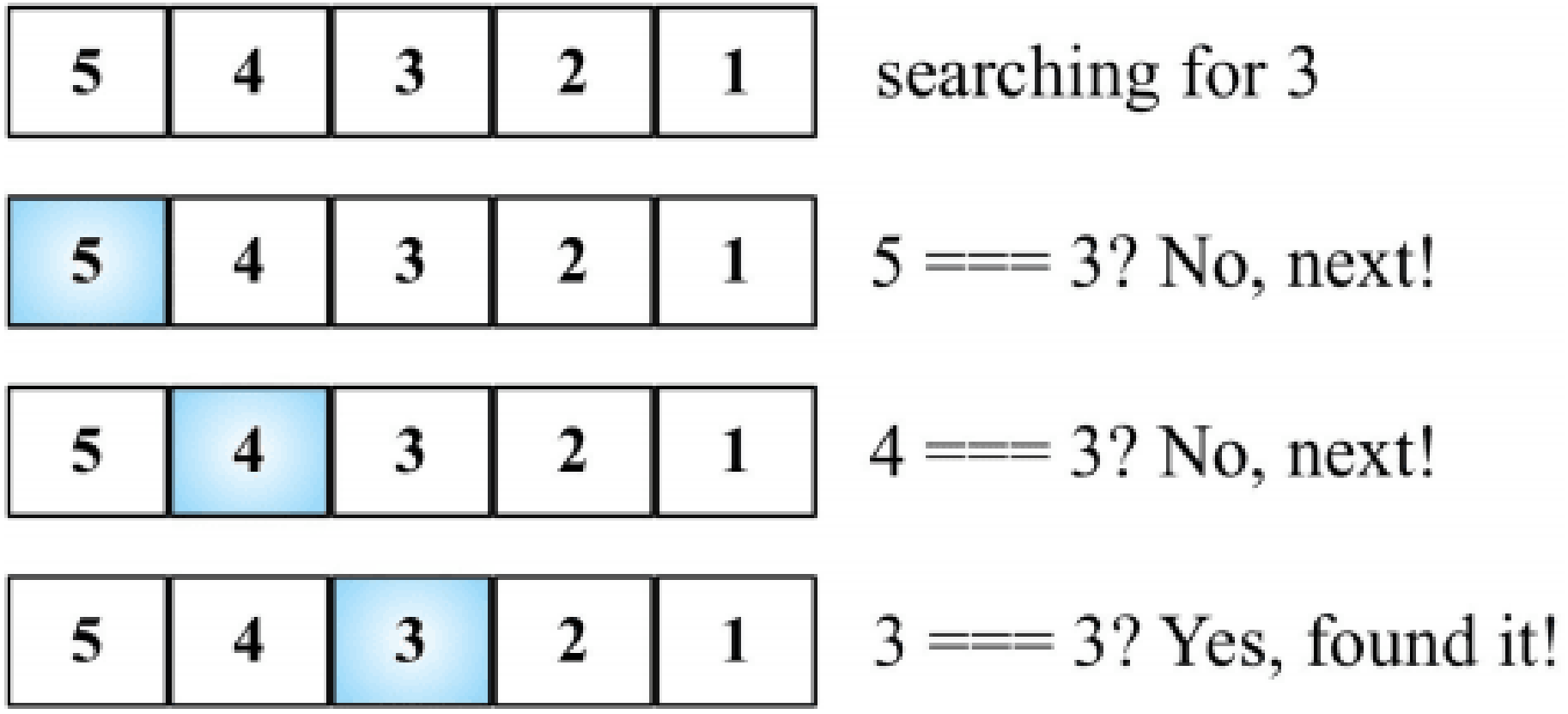
- Finding a specific element or value within a collection of data

Different Types of Searching algorithm

- ✓ Linear search
- ✓ Binary search

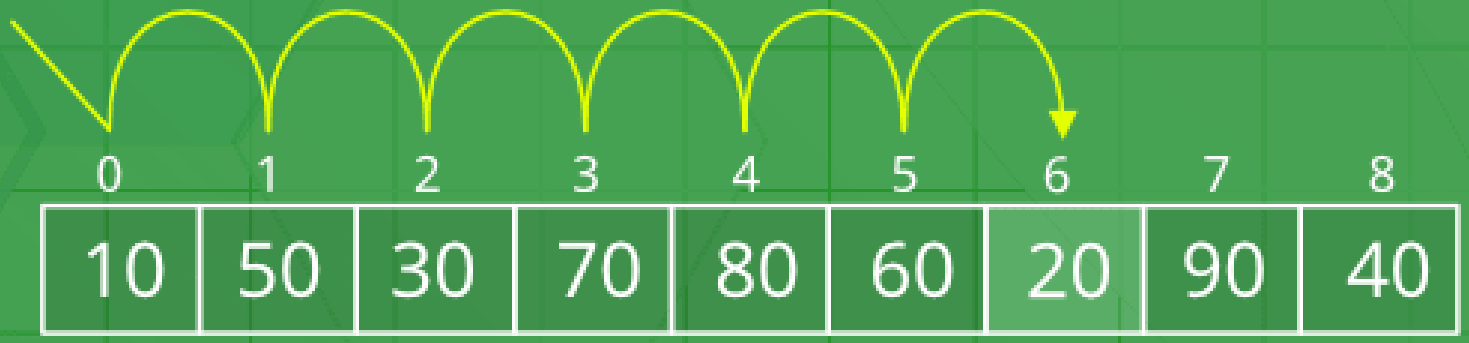
Linear search

- Linear search is a sequential searching algorithm that is used to find an element in a list
- Linear search compares each element of the list with the key till the element is fount or we reach the end of the list
- The linear search algorithm works by sequentially iteration through the whole array from one end until the specified element is fount
- **Best use case :** this search is best used when the list of elements is unsorted and the search is to be performed only once.
- It is also preferred for list to be small, as the time taken grows with the size of the data.


5 4 3 2 1
searching for 3
5 4 3 2 1
5 === 3? No, next!
5 4 3 2 1
4 === 3? No, next!
5 4 3 2 1
3 === 3? Yes, found it!

Linear Search

Find '20'



ial search to find the index of the target T in L.

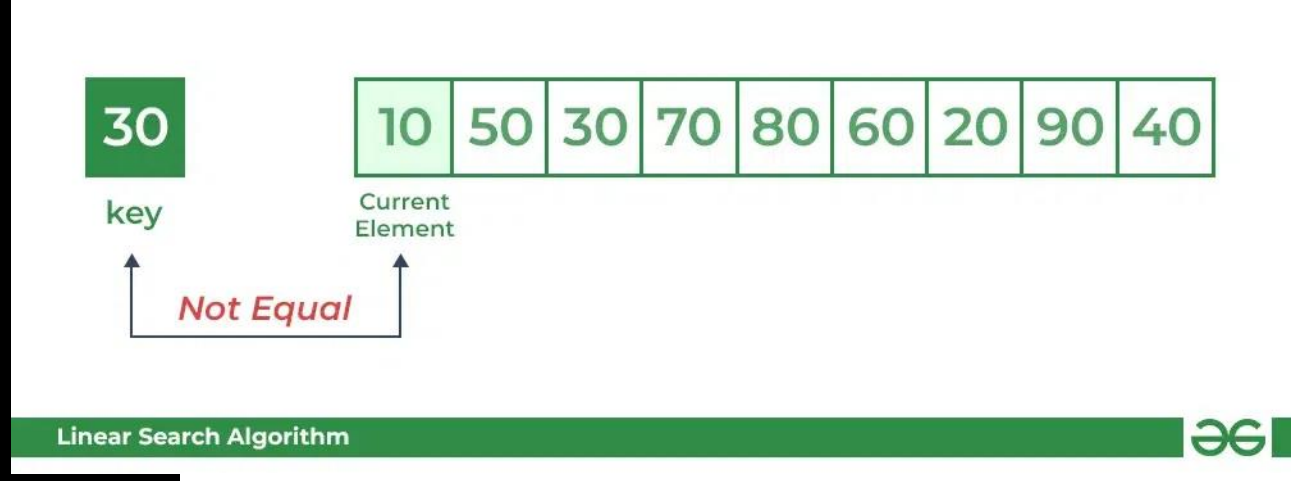
1. Set i to 0.
2. If $L_i = T$, the search terminates successfully; return i.
3. Increase i by 1.
4. If $i < n$, go to step 2. Otherwise, the search terminates unsuccessfully.

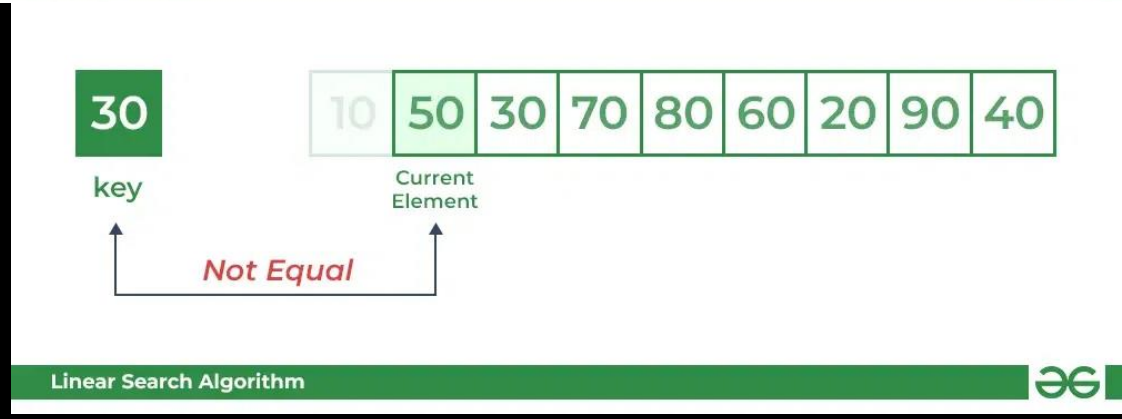
Step 1: Start from the first element (index 0) and compare **key** with each element (arr[i]).

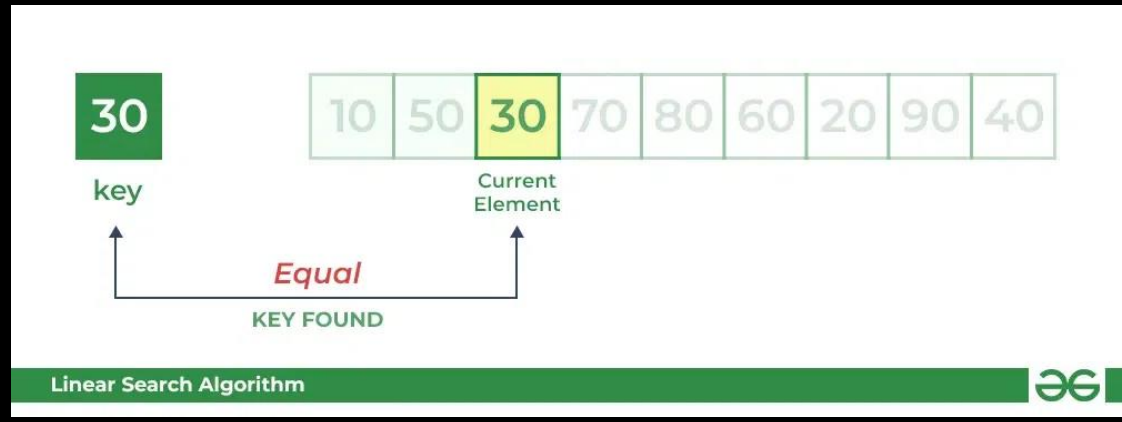
Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.

Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.

Step 2: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).







Algorithm

Linear Search (Array Arr, Value a) // Arr is the name of the array, and a is the searched element.

Step 1: Set i to 0 // i is the index of an array which starts from 0
Step 2: if i > n then go to step 7 // n is the number of elements in array
Step 3: if Arr[i] = a then go to step 6
Step 4: Set i to i + 1
Step 5: Goto step 2
Step 6: Print element a found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Pseudocode of Linear Search Algorithm

```
Start
linear_search ( Array , value)
For each element in the array
    If (searched element == value)
        Return's the searched element location
    end if
end for
end
```

Code

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- **Average Case:** O(N)

Auxiliary Space:

O(1) as except for the variable to iterate through the list, no other variable is used.

Advantages of Linear Search:

•Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
•Does not require any additional memory.
•It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

•Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
•Not suitable for large arrays.

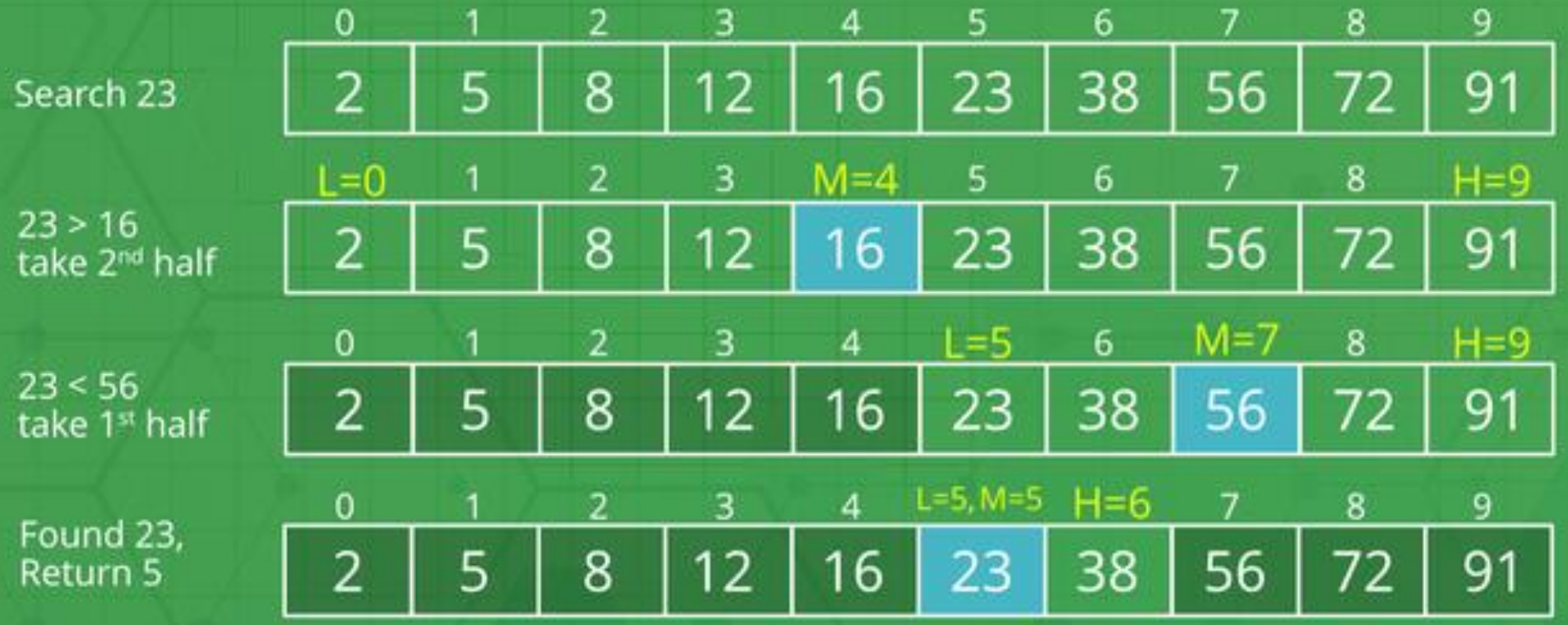
When to use Linear Search?

•When we are dealing with a small dataset.
•When you are searching for a dataset stored in contiguous memory.

Binary search

- Binary search works by halving the number of elements to look through in each iteration, hence reducing the number of elements to be searched. This makes it more efficient than the linear search
- The only condition for this search to work is that the array must be sorted.
- **Best use case :** this search is best used when the list of elements is sorted (not always feasible, specially when new elements are to be inserted)
- The list to be searched can be very large without much decrease in searching time, due to the logarithmic nature of the algorithm.

Binary Search



Conditions for when to apply Binary Search in a Data Structure:

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

In this algorithm,

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
  - If the key is smaller than the middle element, then the left side is used for next search.
  - If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

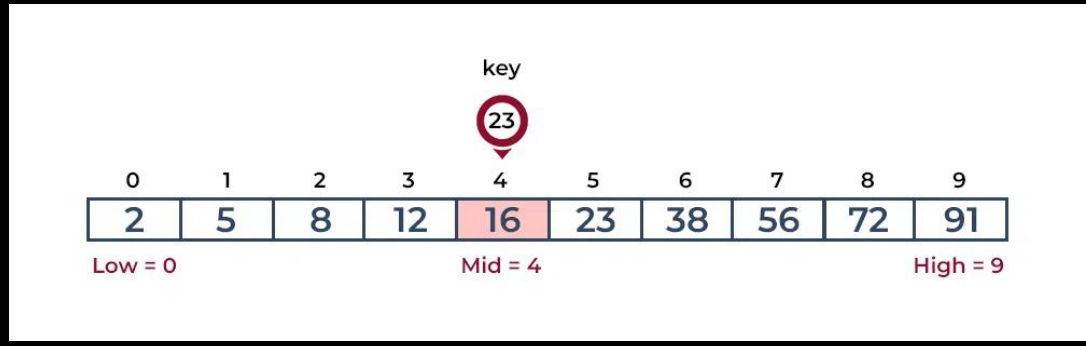
Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.

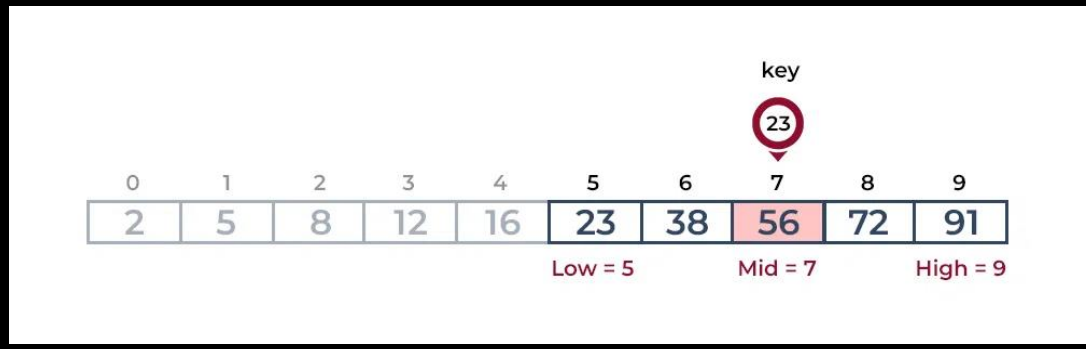
First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

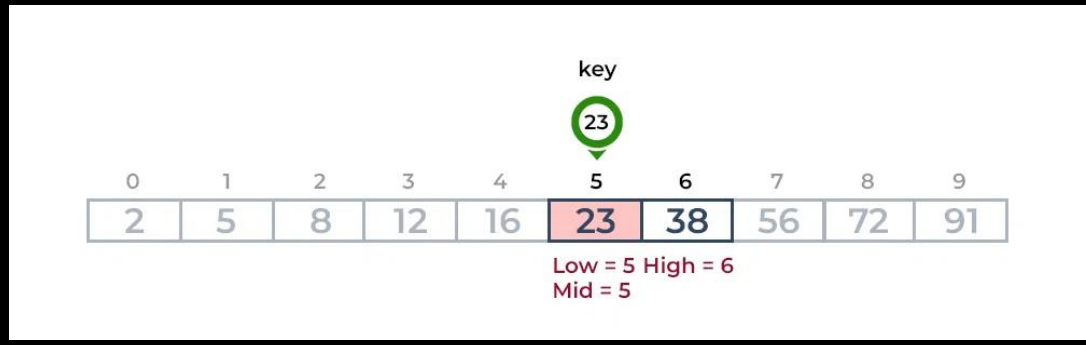
•Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.

•Key is less than the current mid 56. The search space moves to the left.

•**Second Step:** If the key matches the value of the mid element, the element is found and stop search.







Algorithm

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is th e index of the first array element, 'upper_bound' is the index of the last array element, 'val' is t he value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
set pos = mid
print pos
go to step 6
else if a[mid] > val
set end = mid - 1
else
set beg = mid + 1
[end of if]
[end of loop]
Step 5: if pos = -1
print "value is not present in the array"
[end of if]
Step 6: exit

Code

Complexity Analysis of Binary Search:

•Time Complexity:

- Best Case: O(1)
- Average Case: O(log N)
- Worst Case: O(log N)

•Auxiliary Space:

O(1), If the recursive call stack is considered then the auxiliary space will be O(logN).

Advantages of Binary Search:

•Binary search is faster than linear search, especially for large arrays.
•More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
•Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

•The array should be sorted.
•Binary search requires that the data structure being searched be stored in contiguous memory locations.
•Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

Applications of Binary Search:

•Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
•It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
•It can be used for searching a database.

Sorting

- Sorting is the process of arranging a list of elements in a particular order (ascending or descending)

Different Types of Sorting algorithm

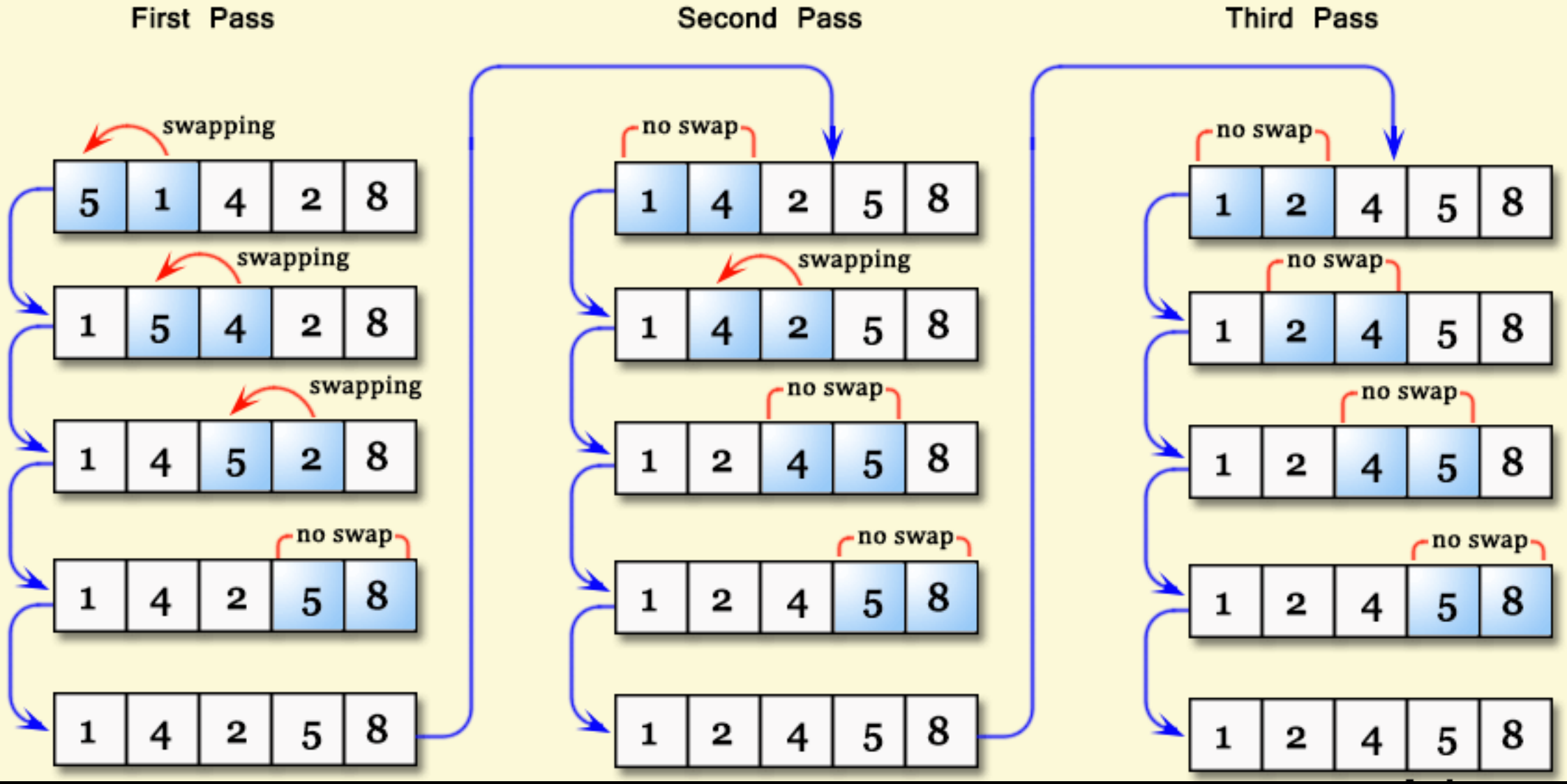
- ✓ Bubble sort
- ✓ Insertion sort
- ✓ Selection sort
- ✓ Merge sort
- ✓ Quick sort

Bubble sort

- Repeatedly swapping the adjacent elements if they are in wrong order.
- The pass through the list is repeated until the list is sorted.
- **Best time complexity** O(n) when elements are already
- **Worst time complexity :** O(n^2)
- **Average time complexity :** O(n^2)
- **Space complexity :** O(1)
- **Stability :** stable

- Repeatedly swap two adjacent elements if in wrong order

Bubble Sorting



How it works:

1.Go through the array, one value at a time.
2.For each value, compare the value with the next value.
3.If the value is higher than the next one, swap the values so that the highest value comes last.
4.Go through the array as many times as there are values in the array.

Algorithm

Step 1 − Check if the first element in the input array is greater than the next element in the array.
Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.
Step 3 − Repeat Step 2 until we reach the end of the array.
Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.
Step 5 − The final output achieved is the sorted array.

Algorithm:

```
Sequential-Bubble-Sort (A)
for i ← 1 to length [A] do
for j ← length [A] down-to i +1 do
    if A[A] < A[j-1] then
       Exchange A[j] ↔ A[j-1]
```

Pseudo Code


```
voidbubbleSort(int numbers[], intarray_size){
   inti, j, temp;
   for (i = (array_size - 1); i>= 0; i--)
   for (j = 1; j <= i; j++)
   if (numbers[j-1] > numbers[j]){
      temp = numbers[j-1];
      numbers[j-1] = numbers[j];
      numbers[j] = temp;
   }
}
```

Code

Advantage of bubble sort

- Bubble sort is easy to understand and implement
- Id does not require any additional memory space
- It is stable sorting algorithm

✓ Bubble sort takes minimum time O(n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid O(n^2) time complexity

Disadvantage of bubble sort

- Bubble sort as a time complexity of $O(n^2)$ which makes it very slow for large data set
- Bubble sort is a comparison based sorting algorithm , which means that it required a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain case

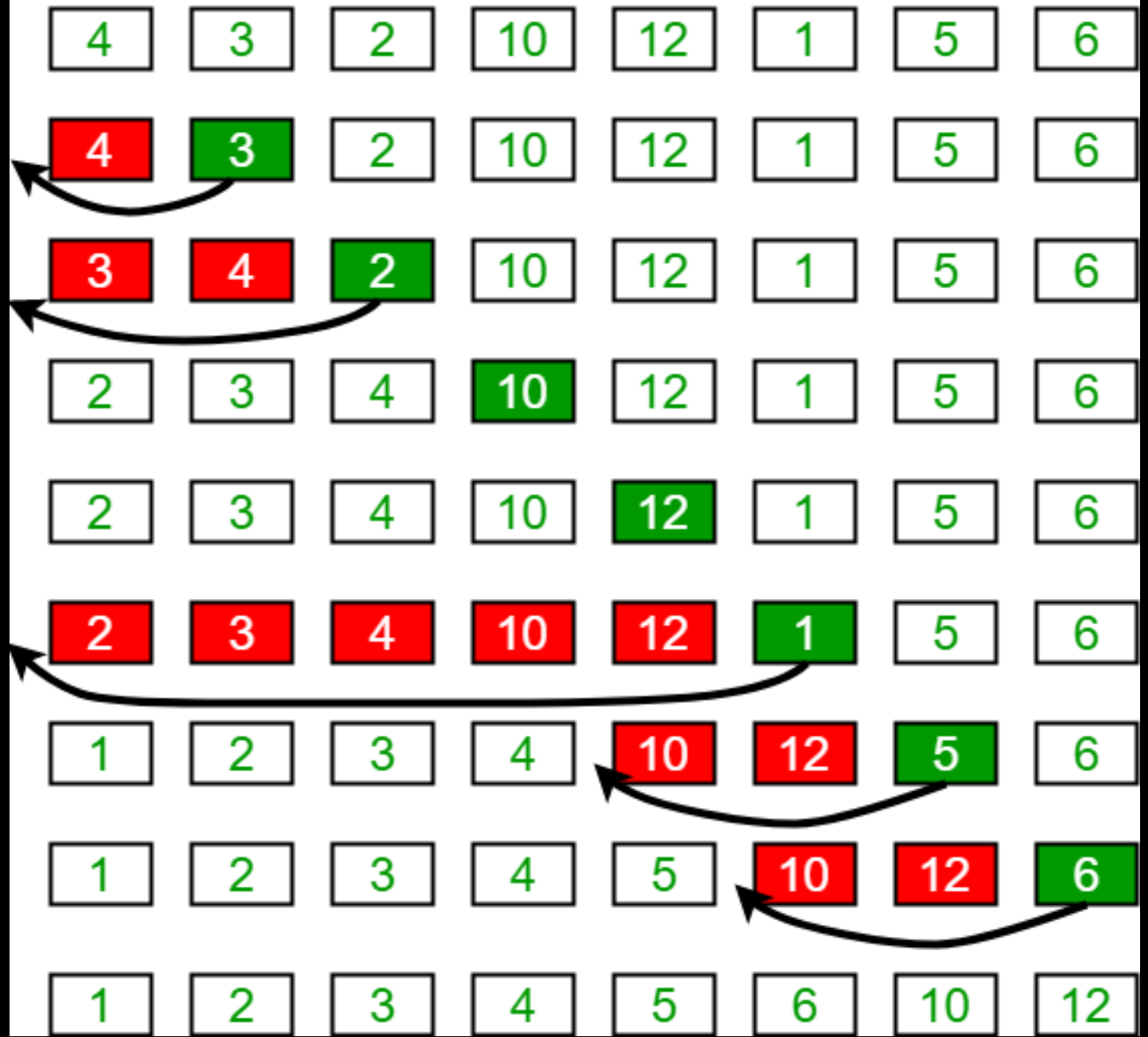
Insertion sort

- To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

- **Best case** is **O(n)**.
- Worst time complexity : O(n^2)
- Average time complexity : O(n^2)
- Space complexity : O(1)
- Stability : stable

- Repeatedly take elements from unsorted sub array and insert in sorted subarray

Insertion Sort Execution Example



Steps

- Step 1 : assume that first element in the list is in its sorted portion of the list and remaining all element are in unsorted portion
- Step 2 : take the first element from the unsorted list and insert that element into the sorted list in order specified
- Step 3 : repeat the above process until all the element from the unsorted list are moved into the sorted list

Insertion Sort Algorithm

Step 1 – If it is the first element, it is already sorted. return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 – Insert the value
Step 6 – Repeat until list is sorted

Pseudocode

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
key = A[j]
i = j – 1
while i > 0 and A[i] > key
A[i + 1] = A[i]
 i = i -1
A[i + 1] = key
```

Code

Time Complexity: O(N^2)
Auxiliary Space: O(1)

Complexity Analysis of Insertion Sort:

Time Complexity of Insertion Sort

•The **worst-case** time complexity of the Insertion sort is **O(n^2)**
•The **average case** time complexity of the Insertion sort is **O(n^2)**
•The time complexity of the **best case** is **O(n)**.

Space Complexity of Insertion Sort

The auxiliary space complexity of Insertion Sort is **O(1)**

Characteristics of Insertion Sort

•This algorithm is one of the simplest algorithms with a simple implementation
•Basically, Insertion sort is efficient for small data values
•Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.

Q1. What are the Boundary Cases of the Insertion Sort algorithm?
Insertion sort takes the maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Q2. What is the Algorithmic Paradigm of the Insertion Sort algorithm?
The Insertion Sort algorithm follows an incremental approach.

Q3. Is Insertion Sort an in-place sorting algorithm?
Yes, insertion sort is an in-place sorting algorithm.

Q4. Is Insertion Sort a stable algorithm?
Yes, insertion sort is a stable sorting algorithm.

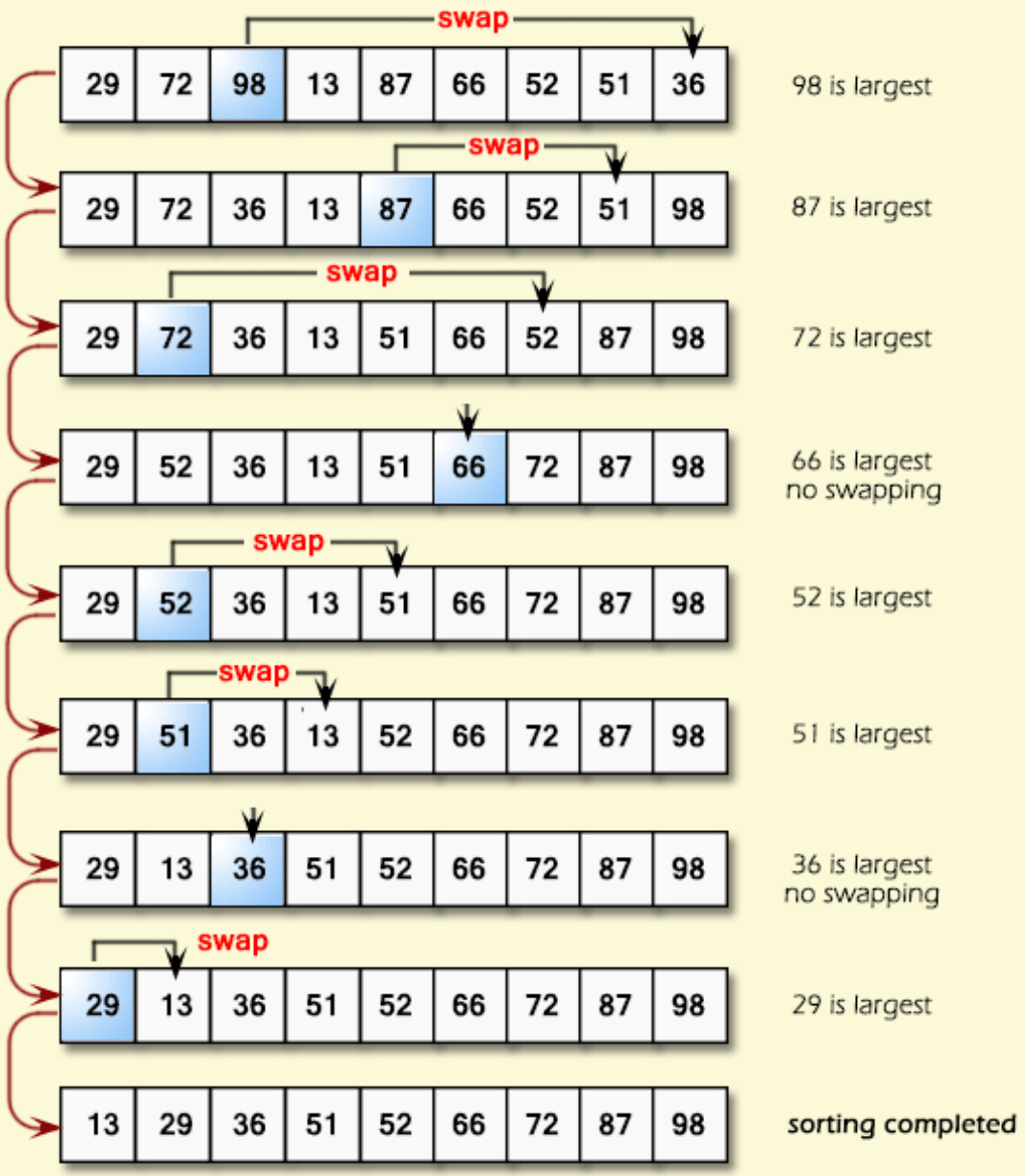
Q5. When is the Insertion Sort algorithm used?
Insertion sort is used when number of elements is small.
It can also be useful when the input array is almost sorted, and only a few elements are misplaced in a complete big array.

Selection sort

- **Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.
- Selection sort has quadratic time complexity even in the best case

- Repeatedly find minimum element in unsorted array and place it at beginning

Selection Sort



Here we find maximum elements in unsorted array and placed it at end of array

KKS Computer Science

Steps

- Step 1 : select the first element of the list
- Step 2 : compare the selected element with all other element in the list
- Step 3 : for every comparison if any element is smaller than selected element swap these two element
- Step 4 : repeat the same procedure with next position in the list

Algorithm

Step 1: Set Min to location 0 in Step 1.
Step 2: Look for the smallest element on the list.
Step 3: Replace the value at location Min with a different value.
Step 4: Increase Min to point to the next element
Step 5: Continue until the list is sorted.

Pseudo code

```
function selection sort
  array : array of items
  size : size of list
  for i = 1 to size - 1
    minimum = i // set current element as minimum
    for j = i+1 to n // check the element to be minimum
      if array[j] < array[minimum] then
        minimum = j;
      end if
    end for
    if indexofMinimum != i then //swap the minimum element with the current element
      swap array[minimum] and array[i]
    end if
  end for
end function
```

Code

Complexity Analysis of Selection Sort

Time Complexity:

The time complexity of Selection Sort is **$O(N^2)$** as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)
- Therefore overall complexity = O(N) * O(N) = O(N*N) = $O(N^2)$

Auxiliary Space:

O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

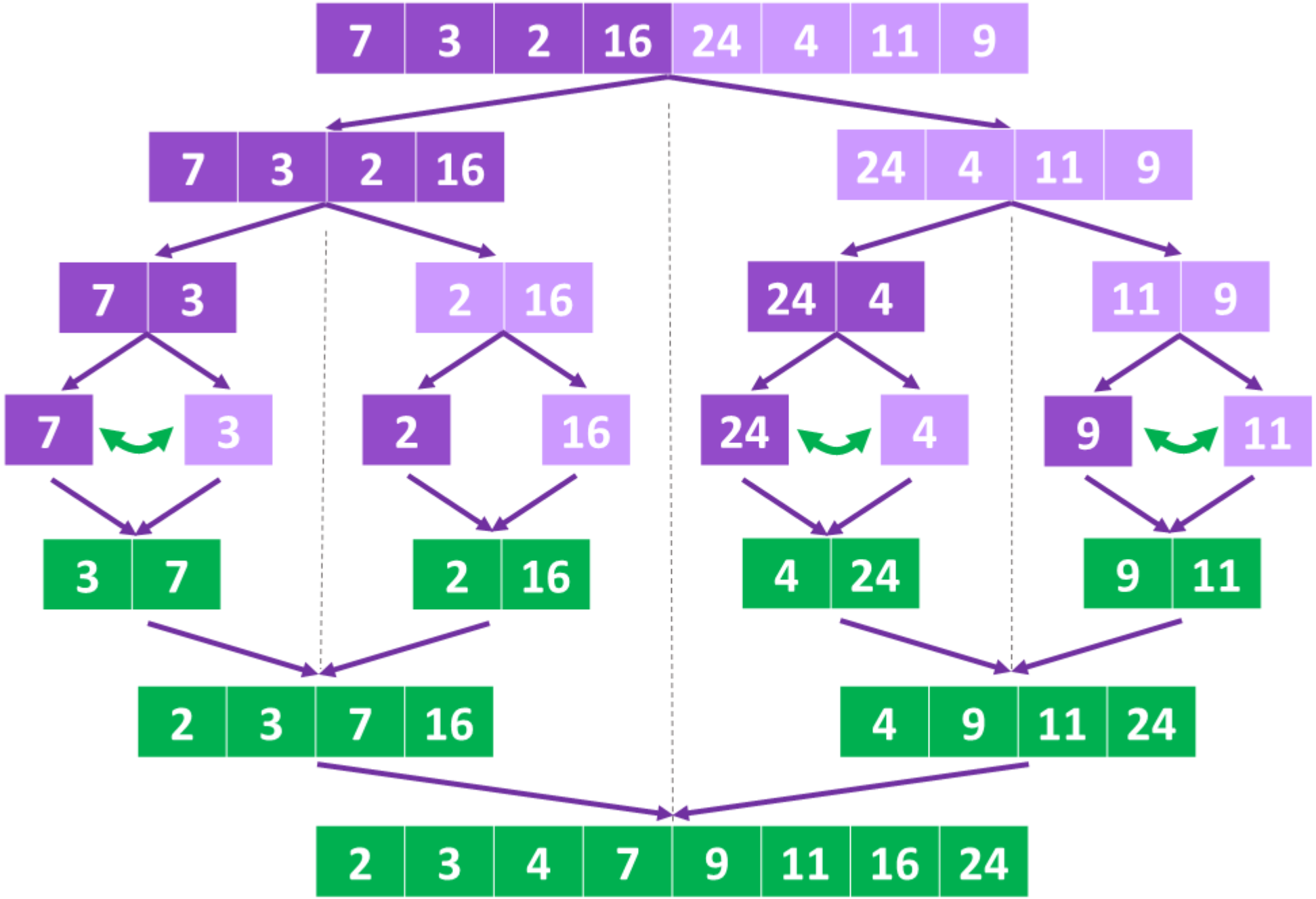
- Selection sort has a time complexity of $O(N^2)$ in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Merge sort

Merge sort is defined as a sorting algorithm that works by dividing an array into
smaller subarrays, sorting each subarray, and then merging the sorted subarrays
back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Merge Sort



Step 1:
Split sub-lists in two until you reach pair of values.

Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

How does Merge Sort work?

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Initially divide the array into two equal halves:

These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.

These sorted subarrays are merged together, and we get bigger sorted subarrays.

This merging process is continued until the sorted array is built from the smaller subarrays.

Algorithm

Step 1: If it is only one element in the list, consider it already sorted, so return.
Step 2: Divide the list recursively into two halves until it can no more be divided.
Step 3: Merge the smaller lists into new list in sorted order.

Pseudo code

```
procedure mergesort( var a as array )
    if ( n == 1 ) return a
      var l1 as array = a[0] ... a[n/2]
      var l2 as array = a[n/2+1] ... a[n]
       l1 = mergesort( l1 )
       l2 = mergesort( l2 )
       return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
     var c as array
     while ( a and b have elements )
          if ( a[0] > b[0] )
             add b[0] to the end of c
             remove b[0] from b
          else
              add a[0] to the end of c
              remove a[0] from a
          end if
      end while
      while ( a has elements )
           add a[0] to the end of c
           remove a[0] from a
      end while
      while ( b has elements )
           add b[0] to the end of c
           remove b[0] from b
       end while
     return c
end procedure
```

Code

Complexity Analysis of Merge Sort

Time Complexity:

O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is $\theta(N\log(N))$. The time complexity of Merge Sort is $\theta(N\log(N))$ in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space:

O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem

Advantages of Merge Sort:

- **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

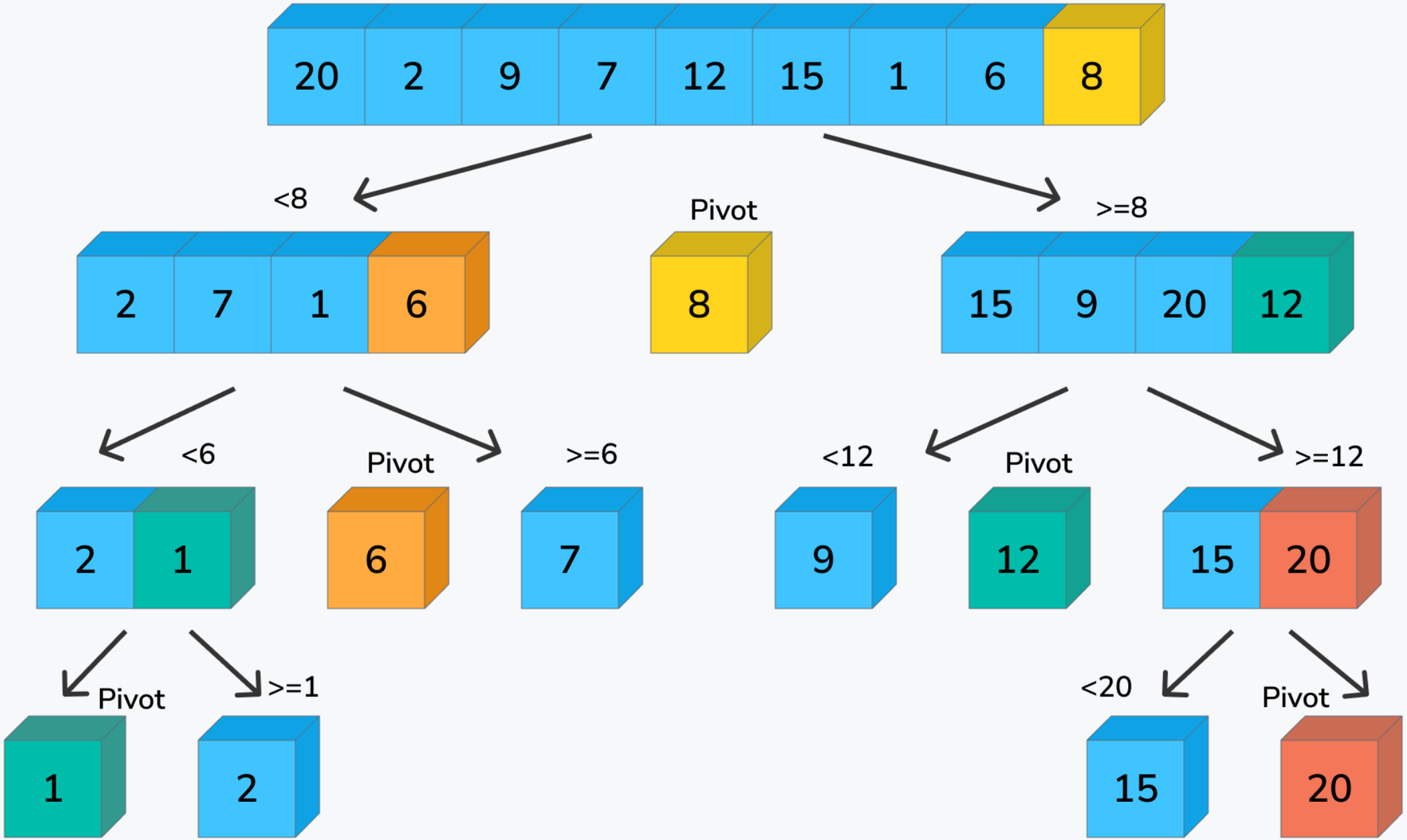
Drawbacks of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Quick sort

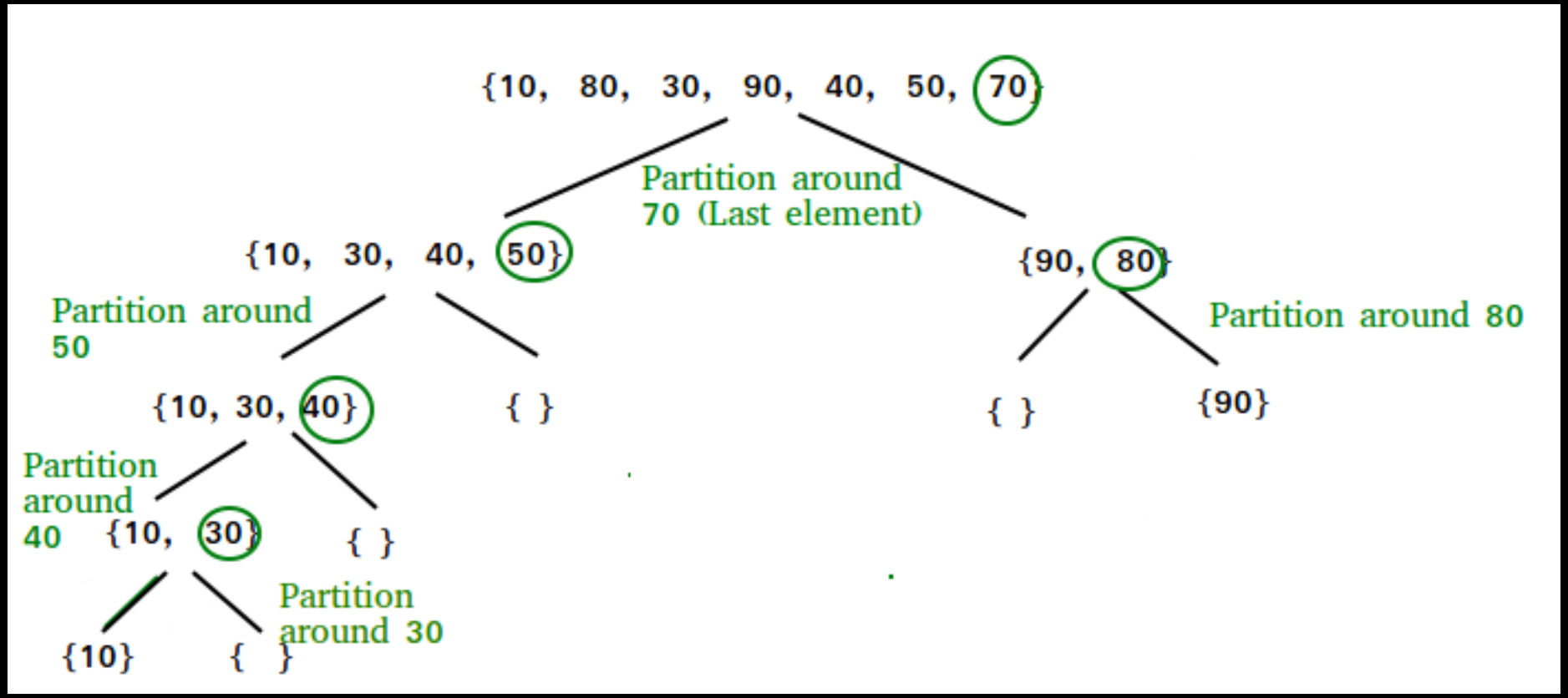
✓ **QuickSort** is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

- Unstable


20 2 9 7 12 15 1 6 8
<8
Pivot
>=8
2 7 1 6
8
15 9 20 12
<6
Pivot
>=6
<12
Pivot
>=12
2 1
6
7
9
12
15 20
Pivot
>=1
<20
Pivot
1
2
15
20

How does QuickSort work?

The key process in **quickSort** is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



Choice of Pivot:

- Always pick the first element as a pivot.
- Always pick the last element as a pivot
- Pick a random element as a pivot.
- Pick the middle as the pivot.

Partition Algorithm:

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i**. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

Algorithm

1.Make the right-most index value pivot
2.Partition the array using pivot value
3.Quicksort left partition recursively
4.Quicksort right partition recursively

Pseudo code

```
procedure quickSort(left, right)
   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if
end procedure
```

Code

Complexity Analysis of Quick Sort:

Time Complexity:

•**Best Case**: Ω (N log (N))

The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.

In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

•**Average Case:** **θ (N log (N))**

Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

•**Worst Case:** **O(N^2)**

The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

•**Auxiliary Space:** O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make $O(N)$.

Advantages of Quick Sort:

•It is a divide-and-conquer algorithm that makes it easier to solve problems.
•It is efficient on large data sets.
•It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

•It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
•It is not a good choice for small data sets.
•It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Thank You !!

Dhanybad !!

Shukriya !!