PCET-NMVPM's
Nutan Maharashtra Institute of Engineering and Technology, Talegaon, Pune
DEPARTMENT OF CSE

AY-2025-26

Subject:Data Strucutres

Subject Code: PCC-201-COM

Subject Teacher: Prof. Sarita Charkha

Syllabus

- Overview of Array, Array as an Abstract Data Type, Operations on Array, Storage Representation, Multidimensional Arrays[2D, nD], Sparse matrix representation using 2D Searching: Sequential Search/Linear Search, Binary Search, Fibonacci Search, and Indexed Sequential Search. Sorting: Concepts- Stability, Efficiency, and Number of Passes, Internal and External Sorting, Bubble sort, Insertion Sort, Selection Sort , Quick Sort, Merge sort Case Study : Social Network Adjacency Matrix Representing friendship connections among millions of users.

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Abstract Data Type (ADT

An **Abstract Data Type (ADT)** is a conceptual model that defines a set of operations and behaviors for a data structure, **without specifying how these operations are implemented** or how data is organized in memory. The definition of ADT only mentions what **operations are to be performed** but not **how** these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. **It is called "abstract" because it provides an implementation-independent view.**

The process of providing only the essentials and hiding the details is known as abstraction.

Features of ADT

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Why Use ADTs?

The key reasons to use ADTs in Java are listed below:

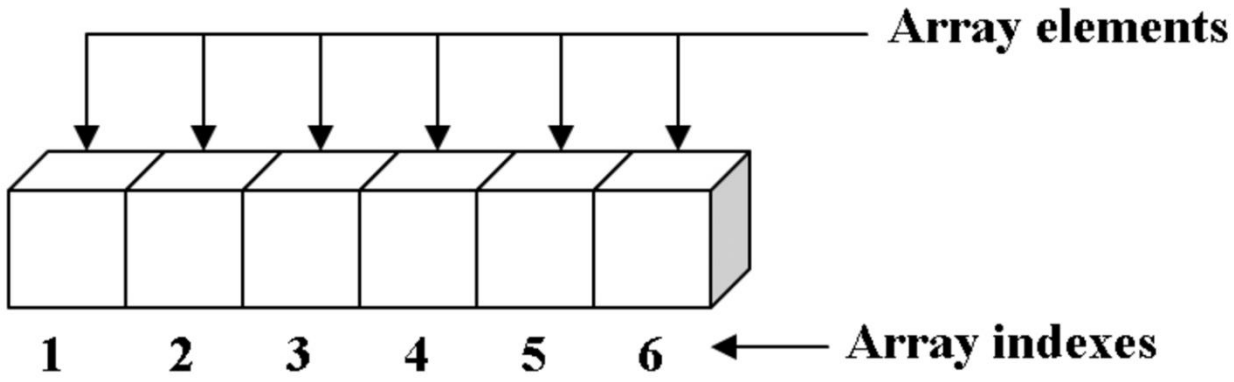
- **Encapsulation:** Hides complex implementation details behind a clean interface.
- **Reusability**: Allows different internal implementations (e.g., array or linked list) without changing external usage.
- **Modularity:** Simplifies maintenance and updates by separating logic.
- **Security:** Protects data by preventing direct access, minimizing bugs and unintended changes.

Arrays as an ADT

The **array** is a basic abstract data type that holds an ordered collection of items accessible by an integer index. These items can be anything from primitive types such as integers to more complex types like instances of classes. Since it's an ADT, it doesn't specify an implementation, but is almost always implemented by an array (data structure) or dynamic array.

Unless otherwise specified, for the remainder of this wiki the word "array" will refer to the abstract data type and not the data structure.

Arrays have one property: they store and retrieve items using an integer index. An item is stored in a given index and can be retrieved at a later time by specifying the same index. The way these indices work is specific to the implementation, but you can usually just think of them as the slot number in the array that the value occupies. Take a look at the image below:



One-dimensional array with six elements

In this visual representation of an array, you can see that it has a size of six. Once six values have been added, no more can be added until the array is resized or something is taken out. You can also see that each "slot" in the array has an associated number. By using these numbers, the programmer can directly index into the array to get specific values. Note that this image uses one-based numbering. Most languages use zero-based numbering where the first index in an array is 0, not 1.

Python's list built-in provides array functionality. C, C++, C#, Java, and a few other languages all have similar syntax for working with arrays. Here's how you would initialize an array in Java and then assign a value to an index. The following code creates a new array with size 5 and sets index 2 equal to 4.

Examples:

1. **int[] arrA = new int[1];**

- ADT: Array of integers
- Domain: elements are integers (int)
- Size: 1 (fixed)
- Operations allowed:
  - arrA[0] — access first (and only) integer
  - arrA[0] = 42; — modify the integer
  - arrA.length — will return 1

It is a **1-element ordered collection of integers**.

2. **String[] arrB = new String[1];**

- ADT: Array of strings
- Domain: elements are strings (String)
- Size: 1
- Operations:
    - arrB[0] — get the string
    - arrB[0] = "Hello"; — store a string
    - arrB.length — returns 1

3. **Person[] arrC = new Person[3];**

- ADT: Array of Person objects
- Domain: elements are of type Person (user-defined class)
- Size: 3
- Operations:
    - arrC[0] — get a reference to the first Person
    - arrC[1] = new Person(...); — store a new Person at index 1
    - arrC.length — returns 3

This is an **Array ADT specialized for Person references.**

Why is Array an ADT?

Because when you use an array, you care about **what you can do (operations)**, not **how it's implemented.**

- You assume that indexing and assignment work.
- You assume constant-time access O(1) per index.
- You don't worry about the low-level memory layout.

Advantages of Array as ADT

- Random Access
- Simplicity
- Memory Locality
- Efficient Iteration

Disadvantages of Array as ADT

- Fixed Size
- Costly Insertions/Deletions

- Wasted Memory
- Inefficient for Dynamic Operations

Operations on Arrays

1. Traversal
    - Visiting each element of the array from start to end
    - Used for displaying, summing, or processing all elements
    - Time Complexity: O(n)
2. Insertion
    - Adding an element at a specific index
    - Requires shifting elements to the right
    - Time Complexity:
        - Best Case (end insertion): O(1)
        - Worst Case (front insertion): O(n)
3. Deletion
    1. Removing an element from a specific index
    2. Requires shifting elements to the left to fill the gap
    3. Time Complexity: O(n)
4. Updating
    - Replacing an element at a given index with a new value
    - Time Complexity: O(1)
5. Searching
    - Finding the position of an element
    - Types: Linear Search O(n), Binary Search O(log n) (if sorted)

Sparse Matrix

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero

elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:

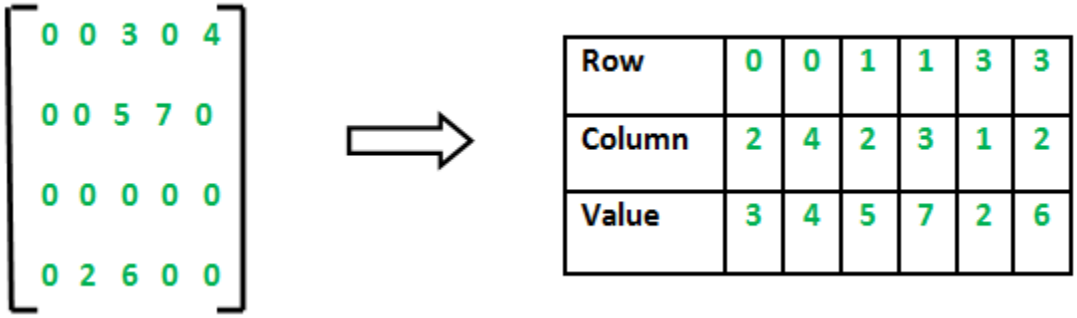
1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index - (row,column)

Example:



What is Triplet Representation of a Sparse Matrix?

- When a matrix is sparse (i.e., most elements are zero), it wastes memory to store all the zeros. The **triplet representation** (or 3-tuple representation) stores only the **non-zero elements** along with their **row and column indices**.
- What is Triplet Representation of a Sparse Matrix?
- When a matrix is sparse (i.e., most elements are zero), it wastes memory to store all the zeros.
- The triplet representation (or 3-tuple representation) stores only the non-zero elements along with their row and column indices.

Steps:

- 1 Count the number of rows m, columns n, and non-zero elements k.
  2 Create a (k+1)× 3 matrix:
- First row: m,n,k,— describes the size of the matrix and number of non-zero elements.

- Remaining k rows: each with row, column, value of non zero elements

Example:

Original 3×3 matrix:

$$A = \begin{bmatrix} 0 & 0 & 5 \\ 0 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix}$$

Here:

- Size: 3×3
- Non-zero elements: 2 (at positions $(0, 2)$ and $(2, 1)$)

Triplet Representation:

$$T = \begin{bmatrix} 3 & 3 & 2 \\ 0 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix}$$

Row by row:

- $3, 3, 2$: Matrix size 3×3, 2 non-zero elements.
- $0, 2, 5$: Element at 0^{th} row and 2^{nd} column is 5
- $2, 1, 3$: Element at 2^{nd} row and 1^{st} column is 3

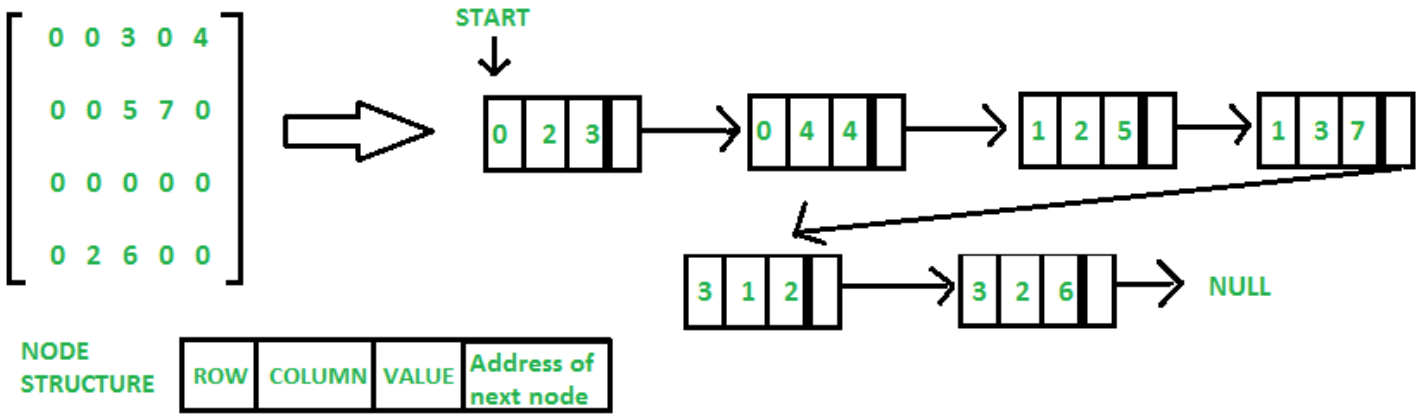
Time Complexity: O(NM), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.

Auxiliary Space: O(NM), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index - (row,column)
- **Next node:** Address of the next node



Time Complexity: O(N*M), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.
Auxiliary Space: O(K), where K is the number of non-zero elements in the array.

Searching Techniques

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data.

Searching techniques in data structures refer to algorithms designed to locate a specific element within a data collection. These techniques are crucial for efficiently retrieving information from various data structures like arrays, linked lists, trees, and graphs. Common search algorithms include linear search, binary search, and hashing.

Purpose & Benefits

- **Efficient Retrieval**: Search algorithms help locate elements within data structures quickly—crucial for speed in software applications.
- **Optimized Performance**: They reduce the time complexity of operations, especially in large datasets.
- **Data Organization**: Searching influences how data is stored and structured, shaping decisions on using arrays, trees, hash tables, etc.
- **Foundation for Advanced Algorithms**: Search logic forms the backbone of more complex methods, including machine learning and database queries.

Applications

- **Search engines** use data structures to retrieve relevant pages.
- **Databases** rely on indexes and trees for fast querying.
- **AI and ML** algorithms use search for decision-making and optimization.
- **Games** use pathfinding (like A* search) to move characters intelligently

Searching: List Searches- Sequential Search- Variations on Sequential Searches- Binary Search- Analyzing Search Algorithm- Hashed List Searches- Basic Concepts- Hashing Methods- Collision Resolutions- Open Addressing- Linked List Collision Resolution- Bucket Hashing.

When we search an item in an array, there are two most common algorithms used based on the type of input array.

- **Linear Search** : It is used for an unsorted array. It mainly does one by one comparison of the item to be search with array elements. It takes linear or O(n) Time.
- **Binary Search** : It is used for a sorted array. It mainly compares the array's middle element first and if the middle element is same as input, then it returns. Otherwise it searches in either left half or right half based on comparison result (Whether the mid element is smaller or greater). This algorithm is faster than linear search and takes O(Log n) time.

List Searches

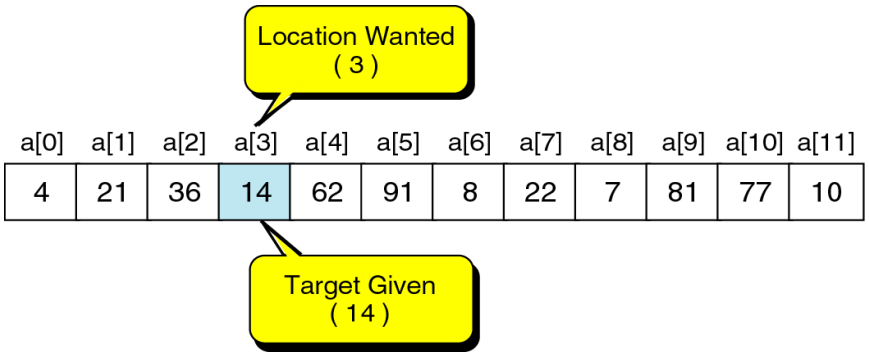
Searching is the process used to find the location of a target among a list of objects. The two basic searches for arrays are the sequential search and the binary search.

a) The sequential search can be used to locate an item in any array.
b) The binary search, on the other hand, requires an ordered list.

1. Sequential search

The sequential search is used whenever the list is not ordered. Generally, you use this technique only for small lists or lists that are not searched often

Example 1:



In the sequential search, we start searching for the target at the beginning of the list and continue until we find the target. This approach gives us two possibilities: either we find it reach the end of the list.

Example 2:

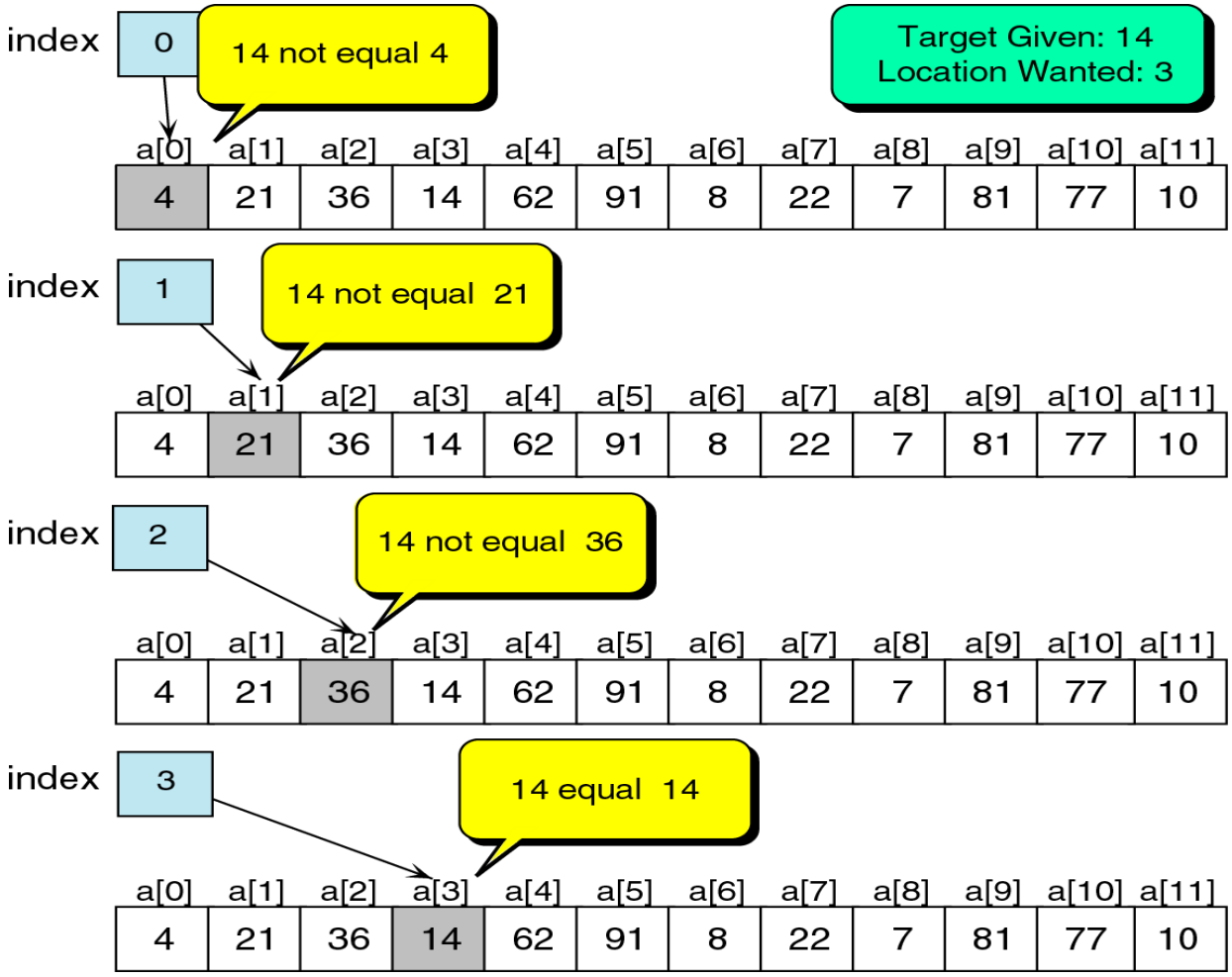


Figure: Successful search of an unordered list

Example 3:

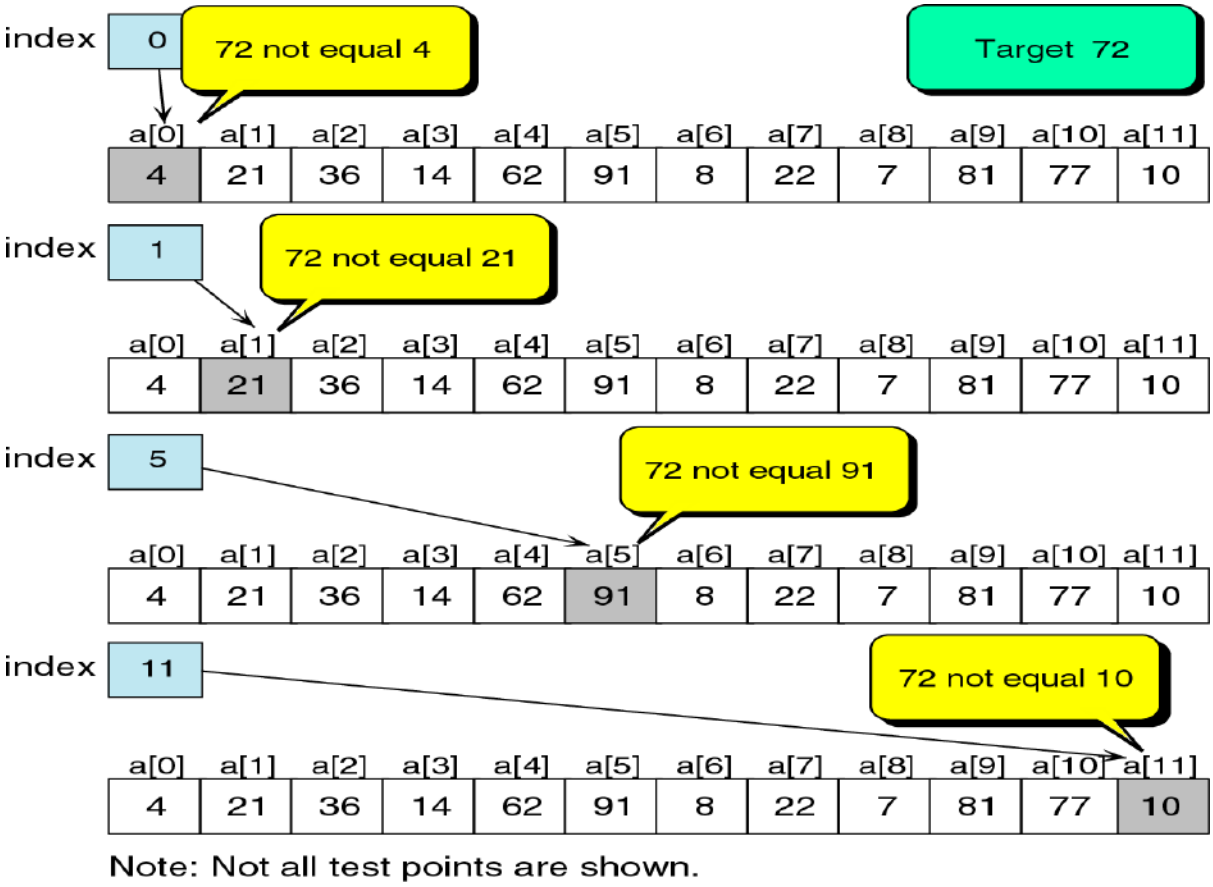


Figure: Unsuccessful search of an unordered list

Time and Space Complexity of Linear Search Algorithm:

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.

Applications of Linear Search Algorithm:

- **Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
- **Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with

- **Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
- **Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Advantages of Linear Search Algorithm:

- 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.

Disadvantages of Linear Search Algorithm:

- 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 Algorithm?

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

Linear Search Algorithm

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                       SET POS = I
                       PRINT POS
                       Go to Step 6
                  [END OF IF]
                   SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

2. Binary Search

The sequential search algorithm is very slow. If we have an array of 1000 elements, we must make 1000 comparisons in the worst case.

The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or the second half of the list.

mid = (begin + end) / 2

If it is in the first half, we do not need to check the second half. If it is in the second half, we do not need to test the first half.

In other words, we eliminate half the list from further consideration with just one comparison. We repeat this process, eliminating half of the remaining list with each test, until we find the target or determine that it is not in the list.

To find the middle of the list, we need three variables: one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list. We analyze two cases here: the target is in the list and the target is not in the list.

Example 1:

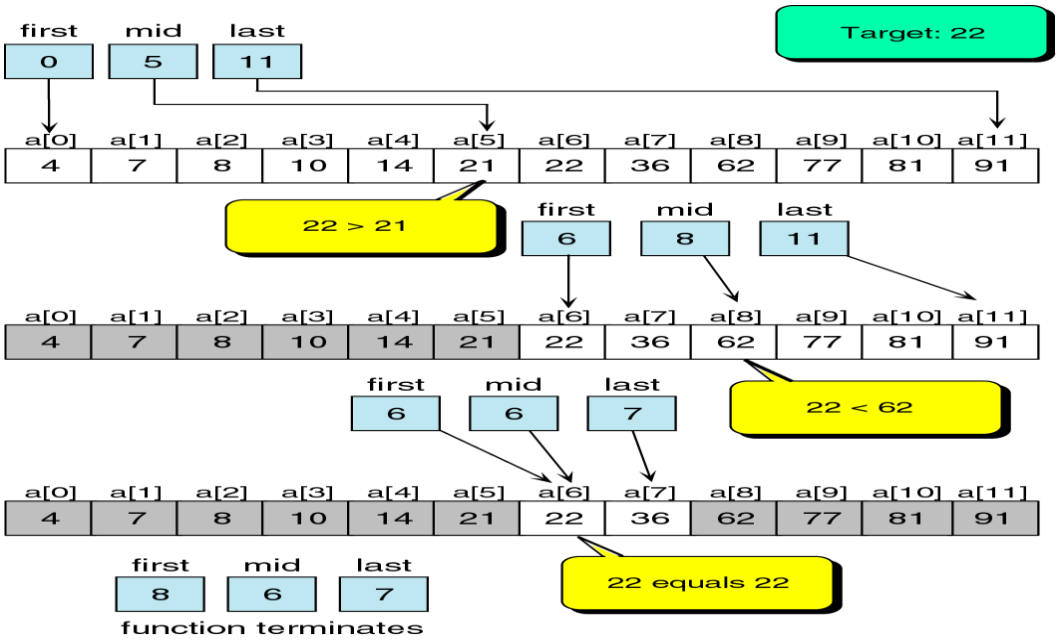


Figure: Successful search of binary search

Example 2:

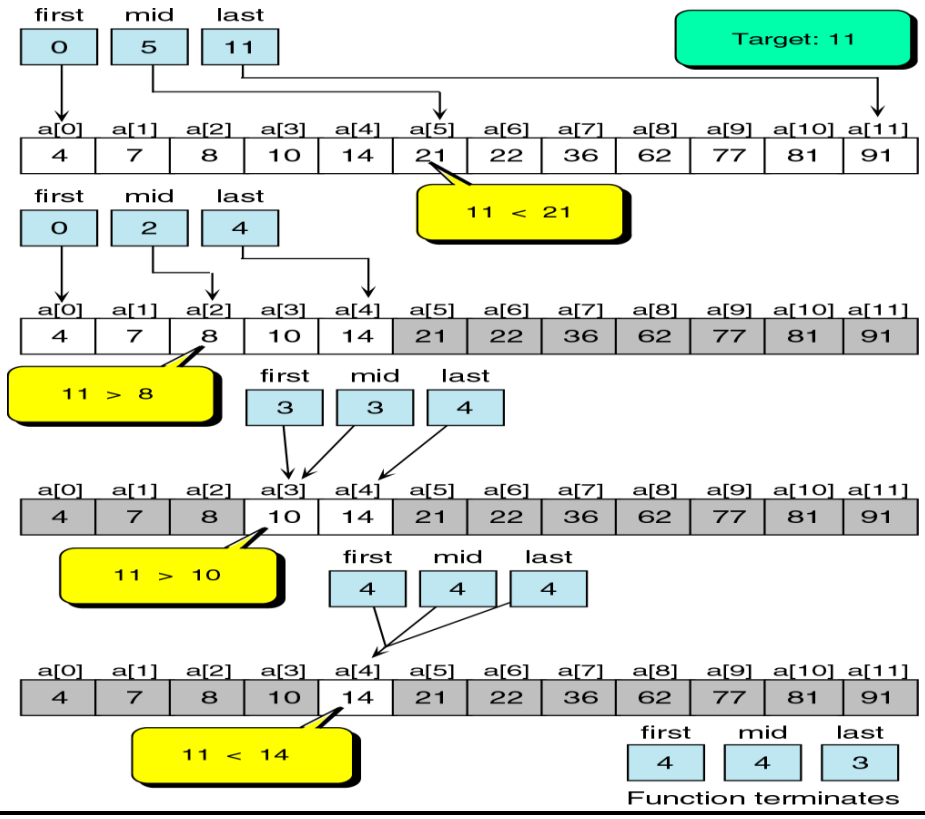


Figure: Unsuccessful search of binary search

The comparison of sequential search & binary search is as follows

List size	Iterations	
	binary	sequential
16	4	16
50	6	50
256	8	256
1000	10	1000
10000	14	10000
100000	17	100000
1000000	20	1000000

Binary Search Algorithm

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] 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
```

3. Fibonacci Search

The Fibonacci Search Algorithm makes use of the Fibonacci Series to diminish the range of an array on which the searching is set to be performed. With every iteration, the search range decreases making it easier to locate the element in the array. The detailed procedure of the searching is seen below –

Step 1 − As the first step, find the immediate Fibonacci number that is greater than or equal to the size of the input array. Then, also hold the two preceding numbers of the selected Fibonacci number, that is, we hold Fm, Fm-1, Fm-2 numbers from the Fibonacci Series.

Step 2 − Initialize the offset value as -1, as we are considering the entire array as the searching range in the beginning.

Step 3 − Until Fm-2 is greater than 0, we perform the following steps –

- Compare the key element to be found with the element at index **[min(offset+F_{m-2},n-1)]**. If a match is found, return the index.
- If the key element is found to be lesser value than this element, we reduce the range of the input from 0 to the index of this element. The Fibonacci numbers are also updated with $F_m = F_{m-2}$.

- But if the key element is greater than the element at this index, we remove the elements before this element from the search range. The Fibonacci numbers are updated as $F_m = F_{m-1}$. The *offset* value is set to the index of this element.

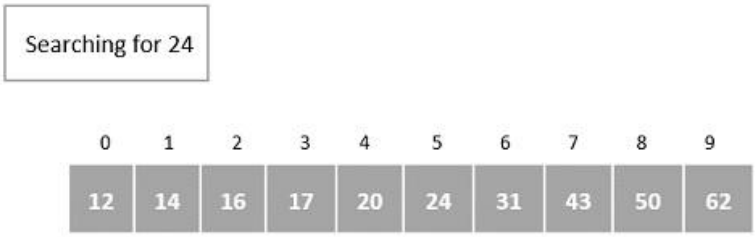
Step 4 − As there are two 1s in the Fibonacci series, there arises a case where your two preceding numbers will become 1. So if F_{m-1} becomes 1, there is only one element left in the array to be searched. We compare the key element with that element and return the 1st index. Otherwise, the algorithm returns an unsuccessful search.

Pseudo-code for Begin Fibonacci Search

```
n <- size of the input array
offset = -1
Fm2 := 0
Fm1 := 1
Fm := Fm2 + Fm1
while Fm < n do:
  Fm2 = Fm1
  Fm1 = Fm
  Fm = Fm2 + Fm1
done
while fm > 1 do:
  i := minimum of (offset + fm2, n  1)
  if (A[i] < x) then:
    Fm := Fm1
    Fm1 := Fm2
    Fm2 := Fm - Fm1
    offset = i
  end
  else if (A[i] > x) then:
    Fm = Fm2
    Fm1 = Fm1 - Fm2
    Fm2 = Fm - Fm1
  end
```

```
    else
      return i;
    end
  done
  if (Fm1 and Array[offset + 1] == x) then:
    return offset + 1
  end
  return invalid location;
end
```

Example: Suppose we have a sorted array of elements {12, 14, 16, 17, 20, 24, 31, 43, 50, 62} and need to identify the location of element 24 in it using Fibonacci Search.



Step 1

The size of the input array is 10. The smallest Fibonacci number greater than 10 is 13.

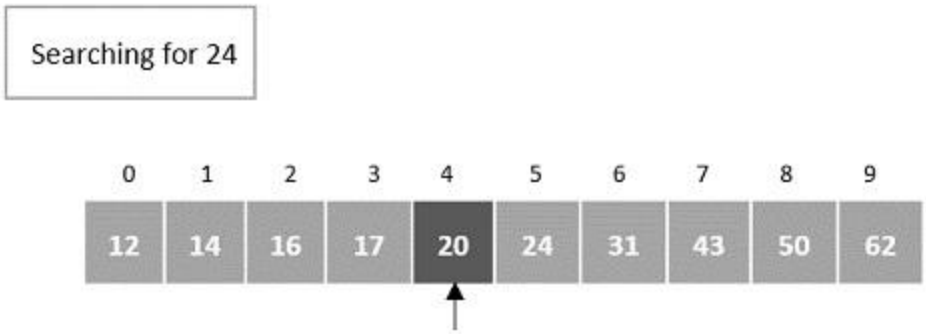
Therefore, $F_m = 13$, $F_{m-1} = 8$, $F_{m-2} = 5$.

We initialize offset = -1

Step 2

In the first iteration, compare it with the element at index = minimum (offset + F_{m-2}, n 1) = minimum (-1 + 5, 9) = minimum (4, 9) = 4.

The fourth element in the array is 20, which is not a match and is less than the key element.



Step 3

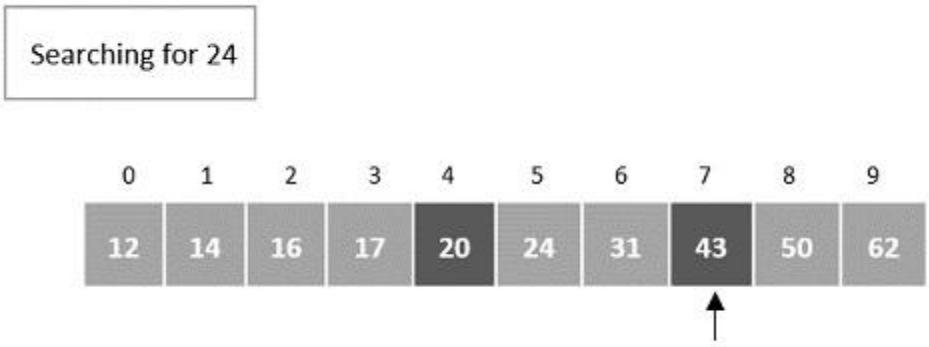
In the second iteration, update the offset value and the Fibonacci numbers.

Since the key is greater, the offset value will become the index of the element, i.e. 4. Fibonacci numbers are updated as $F_m = F_{m-1} = 8$.

Fm-1 = 5, Fm-2 = 3.

Now, compare it with the element at index = minimum (offset + F_{m-2}, n 1) = minimum (4 + 3, 9) = minimum (7, 9) = 7.

Element at the 7^{th} index of the array is 43, which is not a match and is also lesser than the key.



Step 4

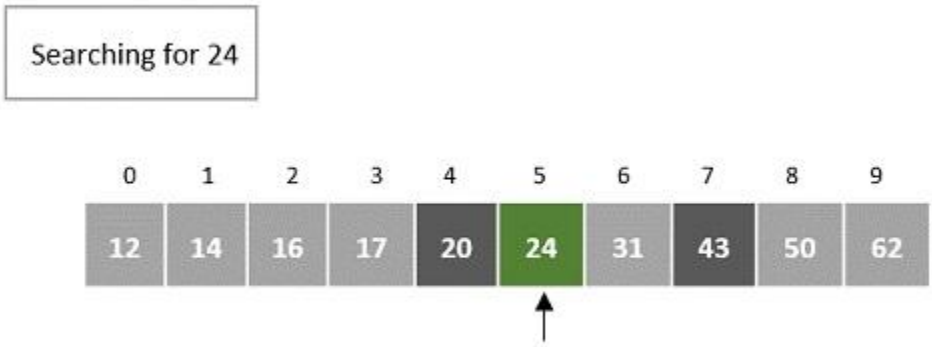
We discard the elements after the 7th index, so n = 7 and offset value remains 4.

Fibonacci numbers are pushed two steps backward, i.e. $F_m = F_{m-2} = 3$.

$F_{m-1} = 2$, $F_{m-2} = 1$.

Now, compare it with the element at index = minimum (offset + Fm-2, n 1) = minimum (4 + 1, 6) = minimum (5, 7) = 5.

The element at index 5 in the array is 24, which is our key element. 5^{th} index is returned as the output for this example array.



The output is returned as 5.

Points to remember related to offset

What is Offset in Fibonacci Search?

- **Offset keeps track of eliminated elements from the front of the array.**
- **Initially, it is set to -1 (meaning nothing is eliminated yet).**

When Moving Right (Element > mid element):

- **We eliminate the left subarray, including mid.**
- **So we update offset to mid.**
- **Then we shift Fibonacci numbers down (reduce search size).**

Important: Update offset when moving right.

When Moving Left (Element < mid element):

- **We eliminate the right subarray, so offset remains unchanged.**
- **We again shift Fibonacci numbers accordingly.**

Important: Do NOT update offset when moving left.

Program : // Write a C program for Fibonacci Search

```
#include <stdio.h>
int min(int, int);
int fibonacci_search(int[], int, int);
int min(int a, int b){
   return (a > b) ? b : a;
}
int fibonacci_search(int arr[], int n, int key){
   int offset = -1;
   int Fm2 = 0;
   int Fm1 = 1;
   int Fm = Fm2 + Fm1;
   while (Fm < n) {
      Fm2 = Fm1;
      Fm1 = Fm;
      Fm = Fm2 + Fm1;
   }
   while (Fm > 1) {
      int i = min(offset + Fm2, n - 1);
      if (arr[i] < key) {
         Fm = Fm1;
         Fm1 = Fm2;
         Fm2 = Fm - Fm1;
         offset = i;
```

```
        } else if (arr[i] > key) {
            Fm = Fm2;
            Fm1 = Fm1 - Fm2;
            Fm2 = Fm - Fm1;
        } else
            return i;
    }
    if (Fm1 && arr[offset + 1] == key)
        return offset + 1;
    return -1;
}
int main(){
   int i, n, key, pos;
   int arr[10] = {6, 11, 19, 24, 33, 54, 67, 81, 94, 99};
   printf("Array elements are: ");
   int len = sizeof(arr) / sizeof(arr[0]);
   for(int j = 0; j<len; j++){
      printf("%d ", arr[j]);
   }
   n = 10;
   key = 67;
   printf("\nThe element to be searched: %d", key);
   pos = fibonacci_search(arr, n, key);
    if(pos >= 0)
```

```
        printf("\nThe element is found at index %d", pos);
    else
        printf("\nUnsuccessful Search");
}
```

4. Indexes Sequential Search

In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group.

Note: When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.

Steps: Indexed Sequential Search is a method that combines **indexing** and **sequential search** to find an element more efficiently.

- You divide the data into blocks.
- Create an **index table** that stores the first element (or key) of each block.
- First, search the index table to find which block might contain the target.
- Then perform a sequential search **within that block**.

Example:

A={3, 7, 10, 15, 21, 29, 34, 40, 46, 50, 55, 60}

You divide this list into blocks of size 4:

- Block 1: [3, 7, 10, 15]
- Block 2: [21, 29, 34, 40]
- Block 3: [46, 50, 55, 60]

Create an **index table** with the **first element** of each block:

Block	Index	First element
1	0	3
2	1	21
3	2	46

Task: Find if 34 is in the list.

Step 1: Search the index table

- 34 is greater than 21 (Block 2 start) but less than 46 (Block 3 start).
- So, 34 must be in **Block 2**.

Step 2: Sequential search in Block 2

- Search [21, 29, 34, 40] sequentially.
- Find 34 at the 3rd element in Block 2.

Advantages:

- Instead of searching 12 elements, you first check 3 in the index table.
- Then search just 4 elements in the block.
- So, search time is faster.

// C program for Indexed Sequential Search

```
#include <stdio.h>
#include <stdlib.h>

void indexedSequentialSearch(int arr[], int n, int k)
{
   int GN = 3; // GN is group number that is number of
            // elements in a group
   int elements[GN], index[GN], i, set = 0;
   int j = 0, ind = 0, start, end;
   for (i = 0; i < n; i += 3) {

      // Storing element
      elements[ind] = arr[i];

      // Storing the index
      index[ind] = i;
      ind++;
   }
```

```
    if (k < elements[0]) {
        printf("Not found");
        exit(0);
    }
    else {
        for (i = 1; i <= ind; i++)
            if (k <= elements[i]) {
                start = index[i - 1];
                end = index[i];
                set = 1;
                break;
            }
    }
    if (set == 0) {
        start = index[GN - 1];
        end = GN;
    }
    for (i = start; i <= end; i++) {
        if (k == arr[i]) {
            j = 1;
            break;
        }
    }
    if (j == 1)
        printf("Found at index %d", i);
    else
        printf("Not found");
}
void main()
{

    int arr[] = { 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Element to search
    int k = 8;
    indexedSequentialSearch(arr, n, k);
}
```

Sorting Techniques

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Why Sorting Algorithms are Important

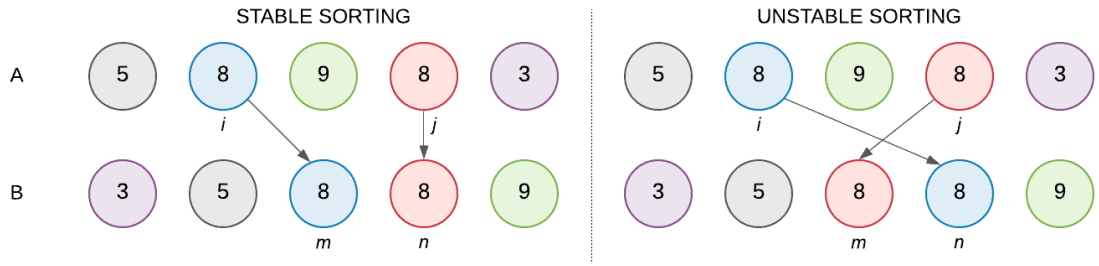
Sorting algorithms are essential in Computer Science as they simplify complex problems and improve efficiency. They are widely used in searching, databases, divide and conquer strategies, and data structures.

Key Applications:

- Organizing large datasets for easier handling and printing
- Enabling quick access to the k-th smallest or largest elements
- Making binary search possible for fast lookups in sorted data
- Solving advanced problems in both software and algorithm design

Sorting Basics

- **In-place Sorting:** An in-place sorting algorithm uses **constant space** for producing the output (modifies the given array only. Examples: Selection Sort, Bubble Sort, Insertion Sort and Heap Sort.
- **Internal Sorting:** Internal Sorting is when all the data is placed in the **main memory** or **internal memory**. In internal sorting, the problem cannot take input beyond allocated memory size.
- **External Sorting :** External Sorting is when all the data that needs to be sorted need not to be placed in memory at a time, the sorting is called external sorting. External Sorting is used for the massive amount of data. For example Merge sort can be used in external sorting as the whole array does not have to be present all the time in memory,
- **Stable sorting:** When two same items appear in the **same order** in sorted data as in the original array called stable sort. Examples: Merge Sort, Insertion Sort, Bubble Sort.
-

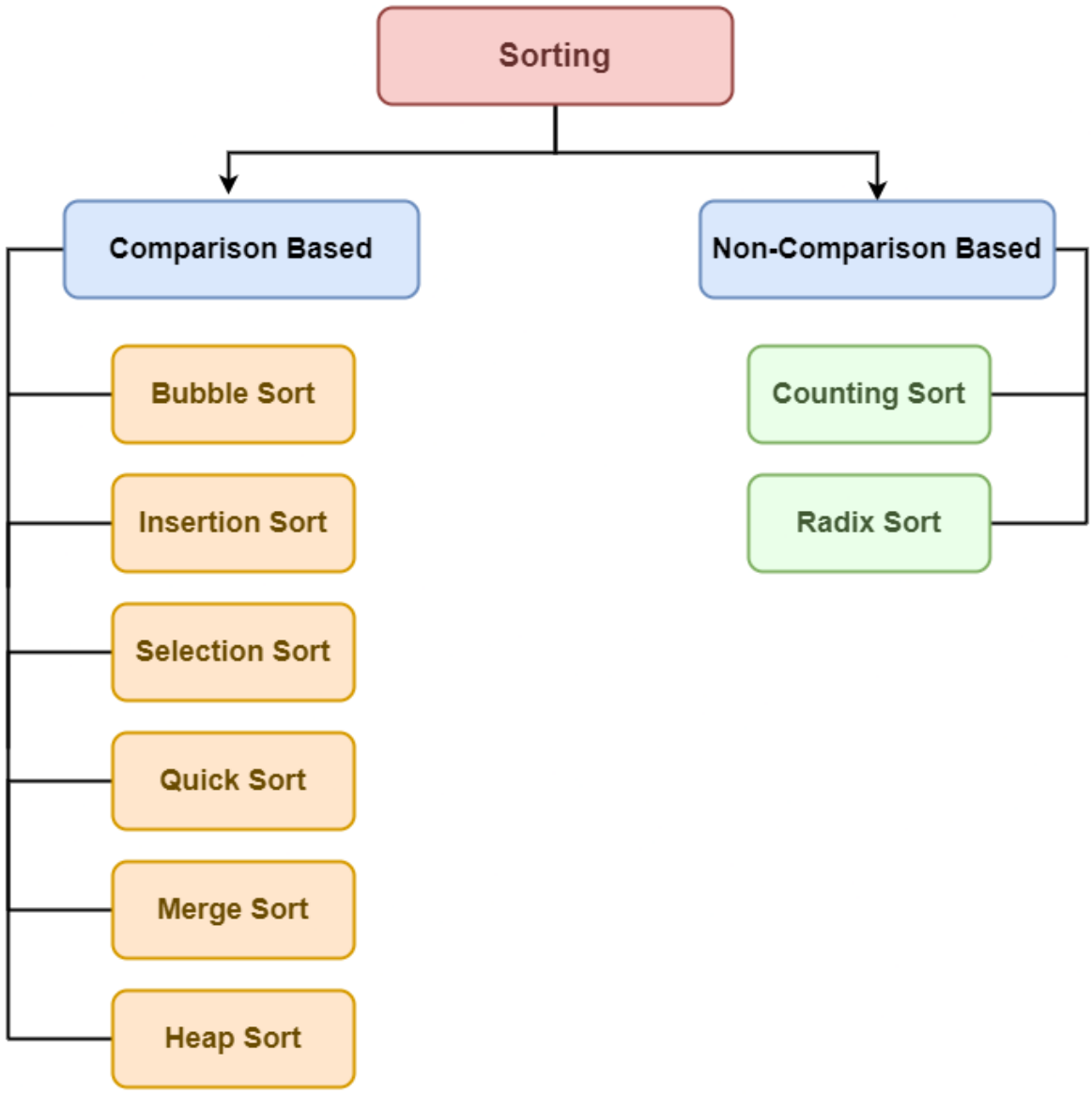


- **Hybrid Sorting:** A sorting algorithm is called Hybrid if it uses more than one standard sorting algorithms to sort the array. The idea is to take advantages of multiple sorting algorithms. For Example IntroSort uses Insertions sort and Quick Sort.
- A hybrid sorting algorithm combines two or more different sorting algorithms to leverage their respective strengths and mitigate their weaknesses, aiming for improved overall performance across various data types and scenarios.

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

- **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
- **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)
- **Non-comparison-based sorting algorithms** are sorting techniques that do **not compare** elements directly using operators like <, >, or ==.
- Instead, they use **mathematical properties**, **indexes**, or **keys** (like digits or bits) to place elements in the correct order.



1. Bubble Sort –

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts. This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Technique

The basic methodology of the working of bubble sort is given as follows:

(a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.

(b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

(c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

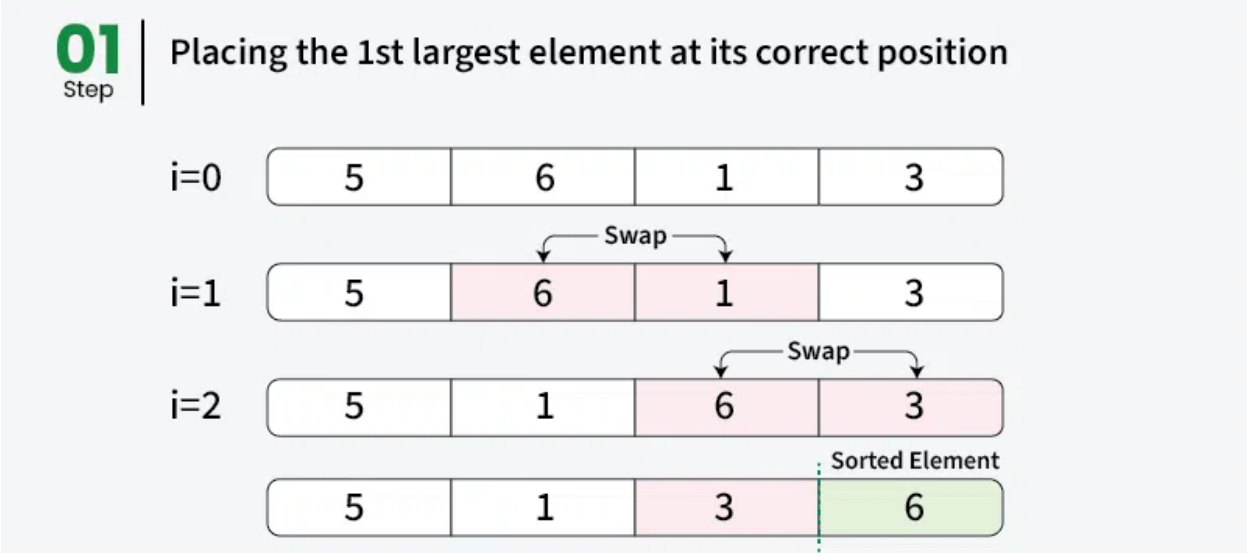
(d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

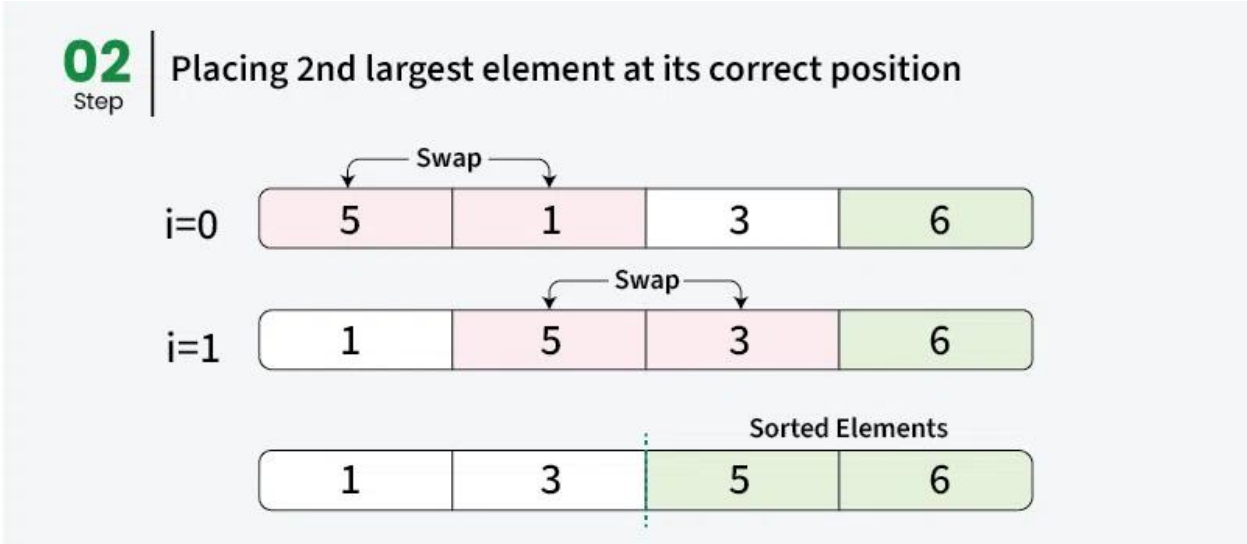
Bubble Sort algorithm

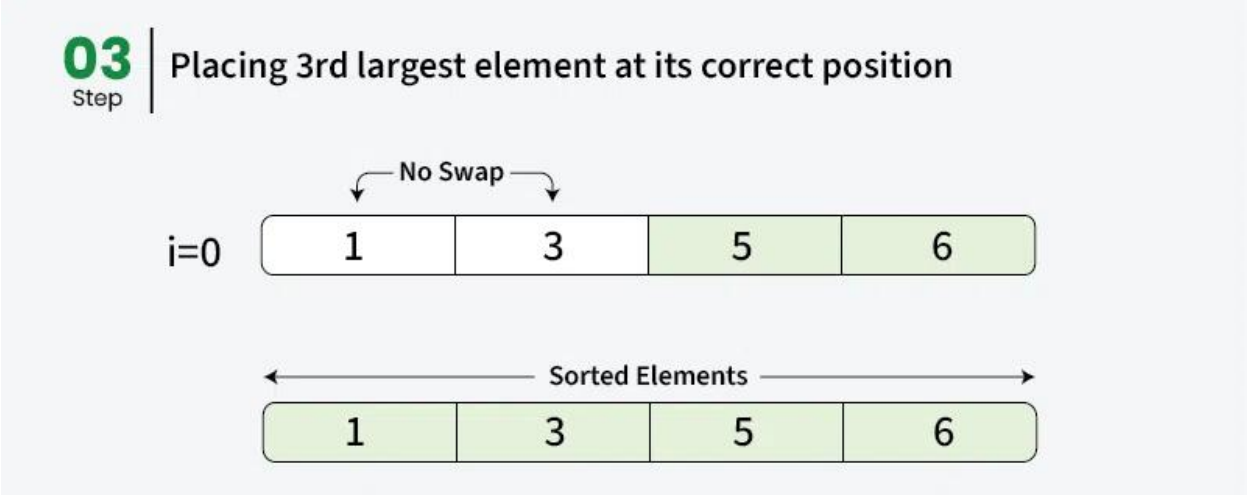
```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For I = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Example: Sort the given elements in ascending order using Bubble sort







1. Insertion Sort