Data Structure
Algorithms & Applications

CT-159

Prepared by
Muhammad Kamran

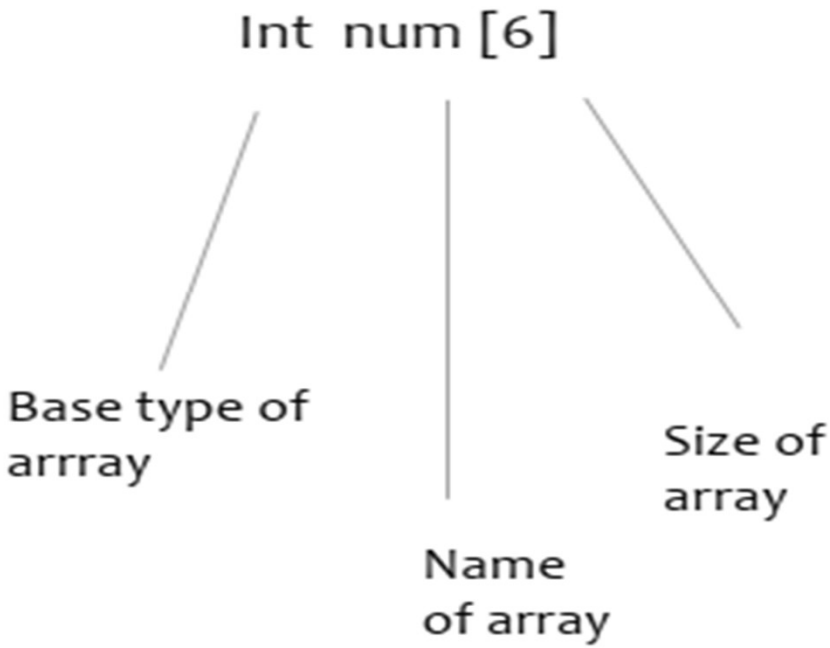
Array

- **A**rray is a collection of variables that can hold value of same type and reference by common name. It is a derived data Structure.
- Contiguous Memory Locations.
- Homogenous
- Indexing from Zero to Last Element.

Contiguous vs Non-contiguous

1	Contiguous memory allocation allocates consecutive blocks of memory to a file/process.	Non-Contiguous memory allocation allocates separate blocks of memory to a file/process.
2	Faster in Execution.	Slower in Execution.
3	It is easier for the OS to control.	It is difficult for the OS to control.
4	Wastage of memory is there.	No memory wastage is there.

Array



Num [0]
Num[1]
Num[2]
Num[3]
Num[4]
Num[5]

Need of Arrays

- To store processed large number of variables of same data type and reference/name.
- Easy understanding of program.
- Arrays provide O(1) random access lookup time. That means, accessing the 1st index of the array and the 1000th index of the array will both take the same time. This is due to the fact that array comes with a pointer and an offset value. The pointer points to the right location of the memory and the offset value shows how far to look in the said memory.

Types of Array

- One Dimensional
- Two Dimensional
- Multi Dimensional

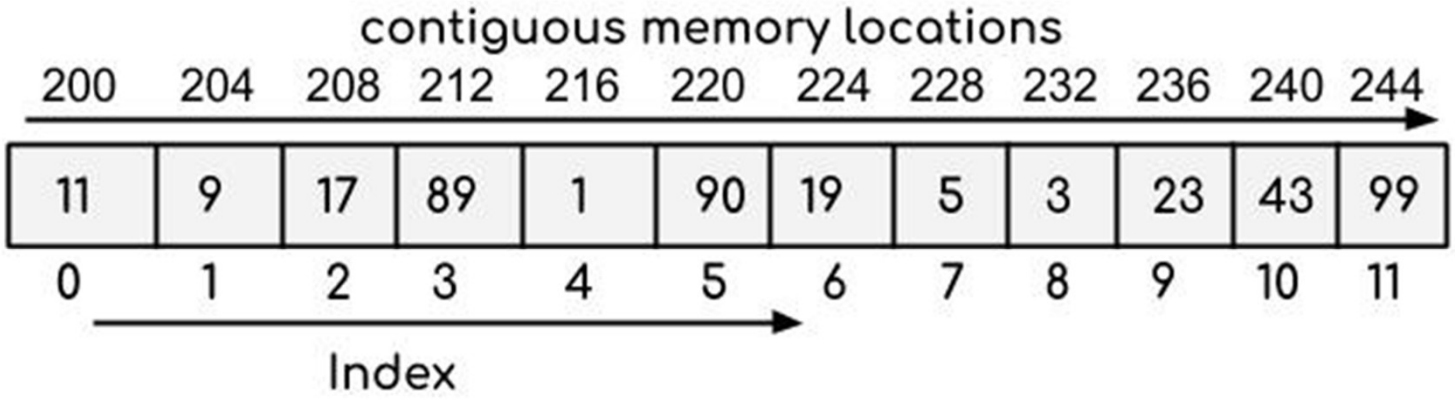
One Dimensional Array

- A one dimensional array is one in which one subscript /indices specification is needed to specify a particular element of array.
- **Declaration :**
  Data_type name_of_array[size_of_array];
- **Example:**
  Int num[20];
  Char name[25];

Memory Representation

- **Total memory in bytes:**

Size of array = size of array*size of(base type)



Two Dimensional Array

- A 2-d array is an array in which each element is itself an array.
- **Declaration :**
  Data_type name_of_array[row][col];
- **Example:**
  Int num[2][4];

1 D vs 2 D Arrays

1 D ARRAY:

C	O	D	I	N	G	E	E	K
0	1	2	3	4	5	6	7	8

single row of elements

2 D ARRAY:

	col 0	col 1	col 2
i \ j	0	1	2
row 0 → 0	A	A	A
row 1 → 1	B	B	B
row 2 → 2	C	C	C

column

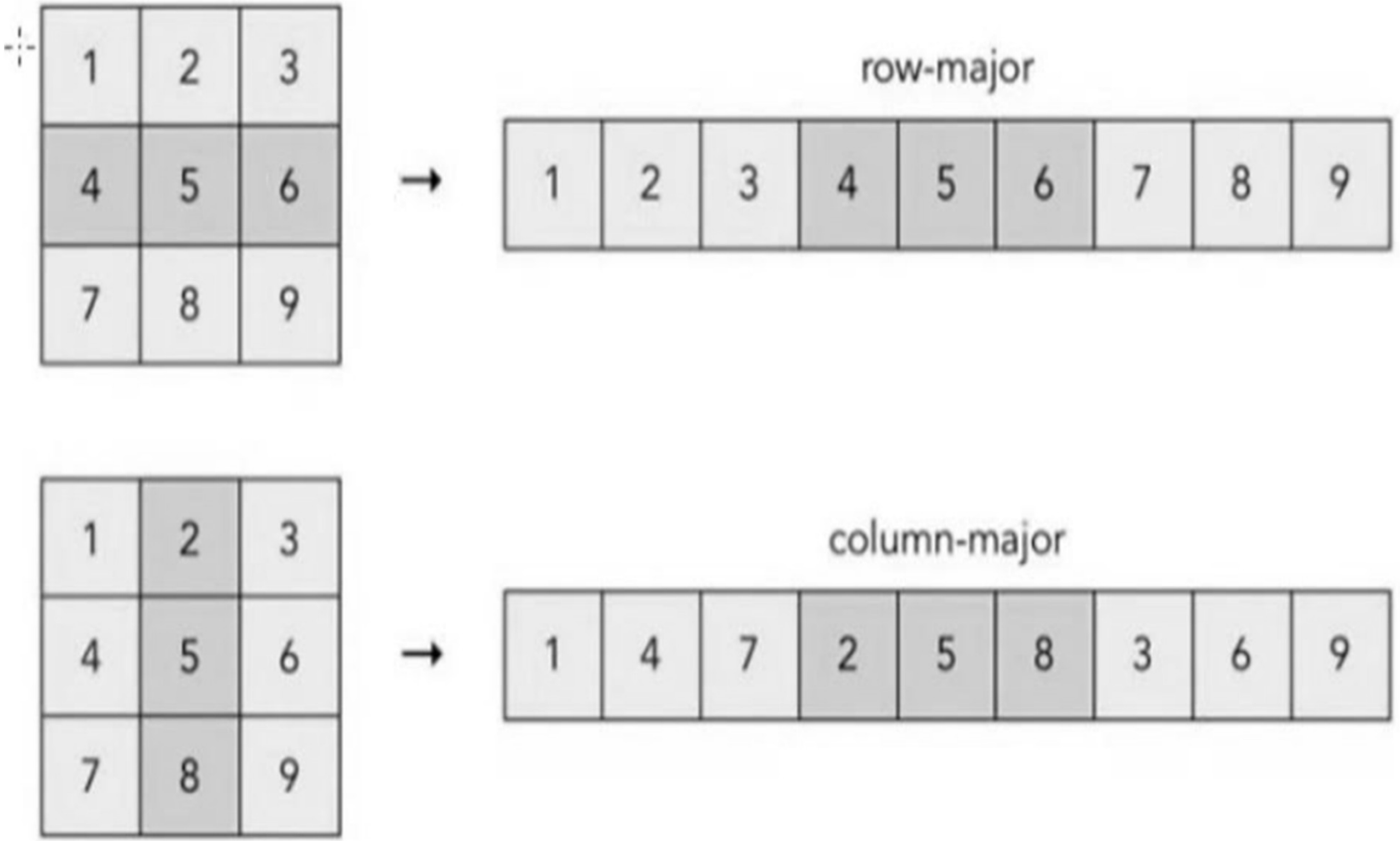
array elements

rows

Row major vs Column major

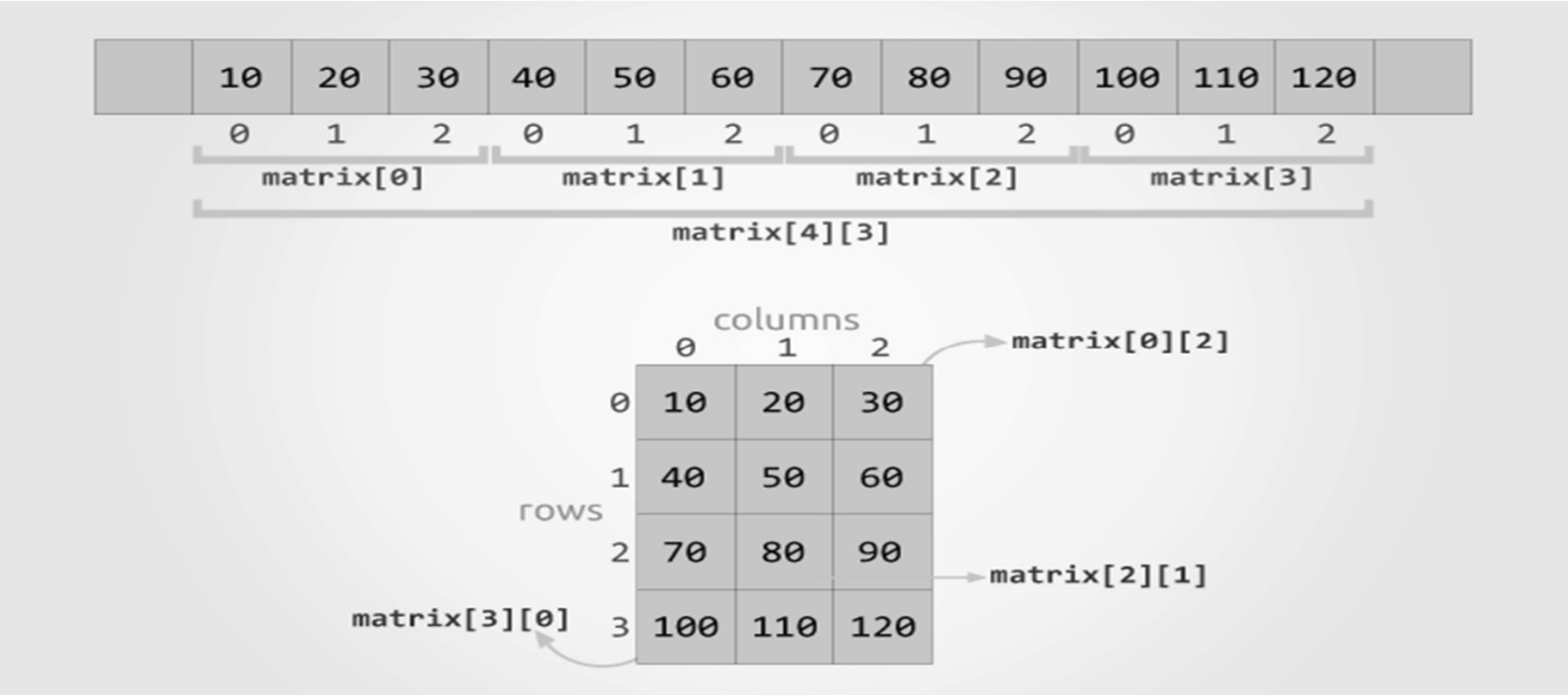
- Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a **Row-Wise fashion**.
- If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in **Column-major order**.

Row major vs Column major



Memory Representation 2D Array

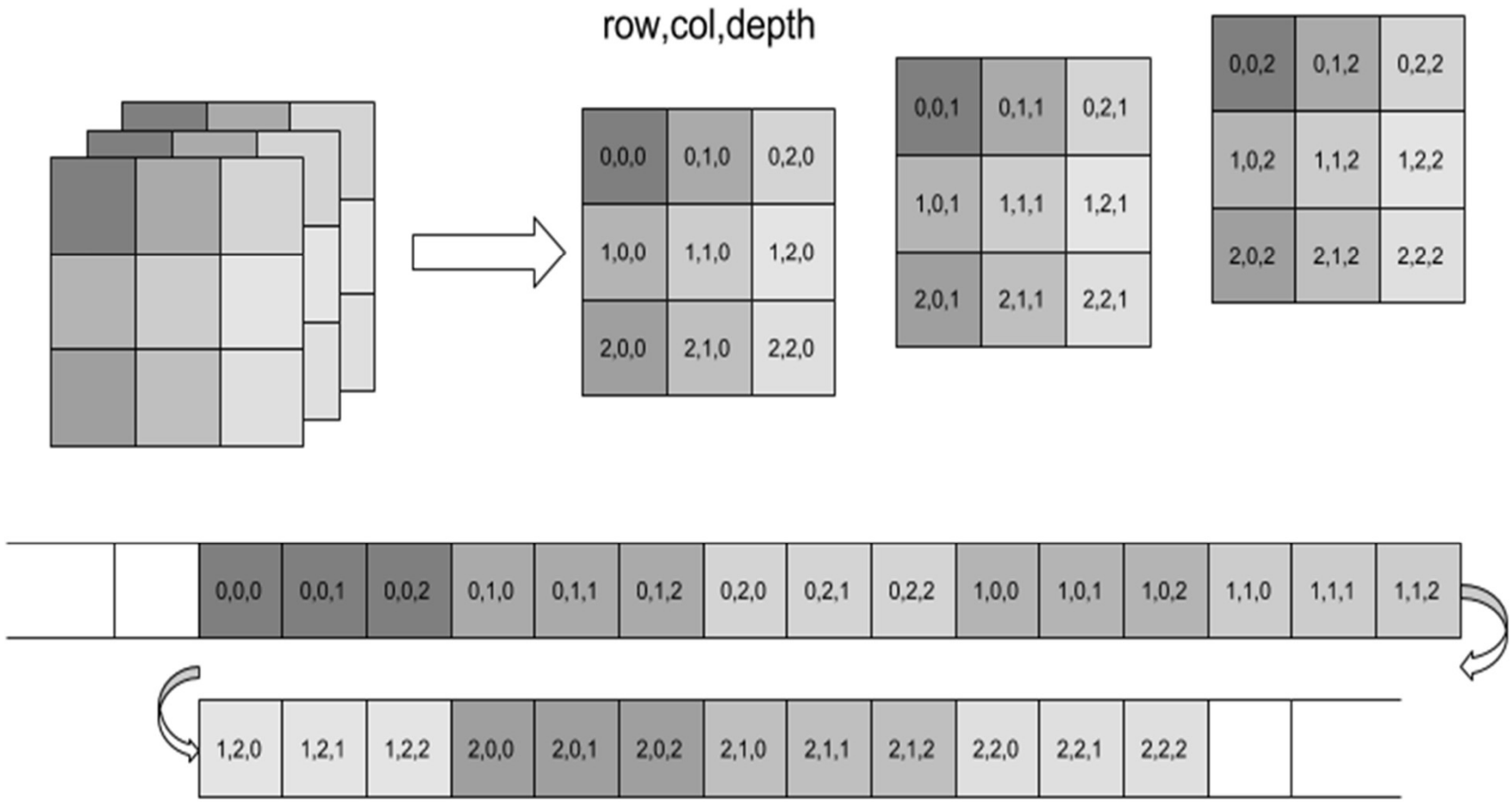
- Total bytes= no of rows*no of columns*size of(base type)



Multi Dimensional Array

- An array with dimensions more than two . The maximum limit of array is compiler dependent.
- **Declration:**
  Data_type array_name [a][b][c][d] ….[n];
- Array of 3 or more dimensional are not often use because of massive memory requirement and complexity.

Memory Representation



Searching Problem

Finding a Phone Number in Phone Book??

Algorithm 1 for Finding a Phone Number

```
findNumber(person) {
for (p = number of first page; p <= number of the last page; p++) {
if person is found on page p {
return the person's phone number
}
}
return NOT_FOUND
}
```

Algorithm 2 for Finding a Phone Number

```
findNumber(person) {
min = the number of the first page
max = the number of the last page
while (min <= max) {
mid = (min + max) / 2 // page number of the middle page
if person is found on page mid {
return the person's number
} else if the person’s name comes earlier in the book {
max = mid – 1
} else {
min = mid + 1
}
}
return NOT_FOUND
}
```

Searching Problem

The phonebook problem is one example of a common task: searching for an item in a collection of data. another example: searching for a record in a database

- **Algorithm 1** is known as sequential search, also called linear search.
- **Algorithm 2** is known as binary search. Only works if the items in the data collection are sorted.

Searching Algorithms

- Based on the type of search operation, these algorithms are generally classified into two categories:
- **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked.
- **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Sequential Search as they repeatedly target the centre of the search structure and divide the search space in half.

Linear Search

- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm/Pseudo-code

Linear Search (Array A, Value x)

- **Step 1**: Set i to 1
- **Step 2**: if i > n then go to step 7
- **Step 3**: if A[i] = x then go to step 6
- **Step 4**: Set i to i + 1
- **Step 5**: Go to Step 2
- **Step 6**: Print Element x Found at index i and go to step 8.
- **Step 7**: Print element not found
- **Step 8**: Exit

Linear Search

Binary Search

- Binary search is a search algorithm used to find the position of a target value within a sorted array.
- It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty.
- The search interval is halved by comparing the target element with the middle value of the search space.

Conditions to apply Binary Search Algorithm

To apply Binary Search algorithm:

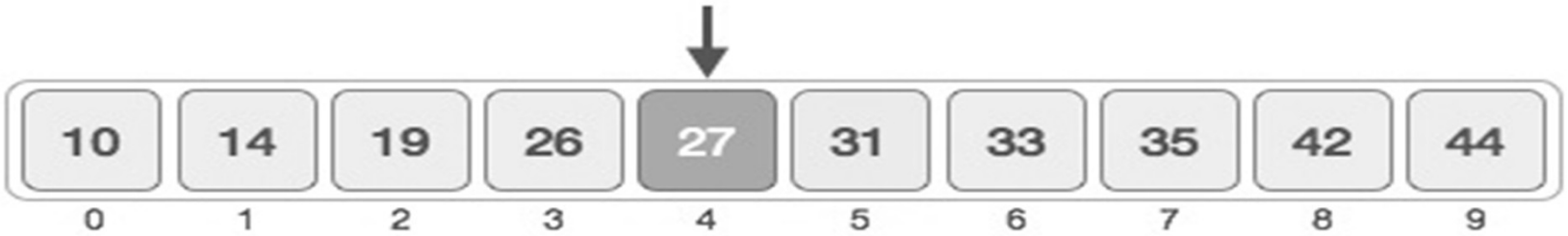
- The data structure must be sorted.
- Access to any element of the data structure should take constant time.

Algorithm

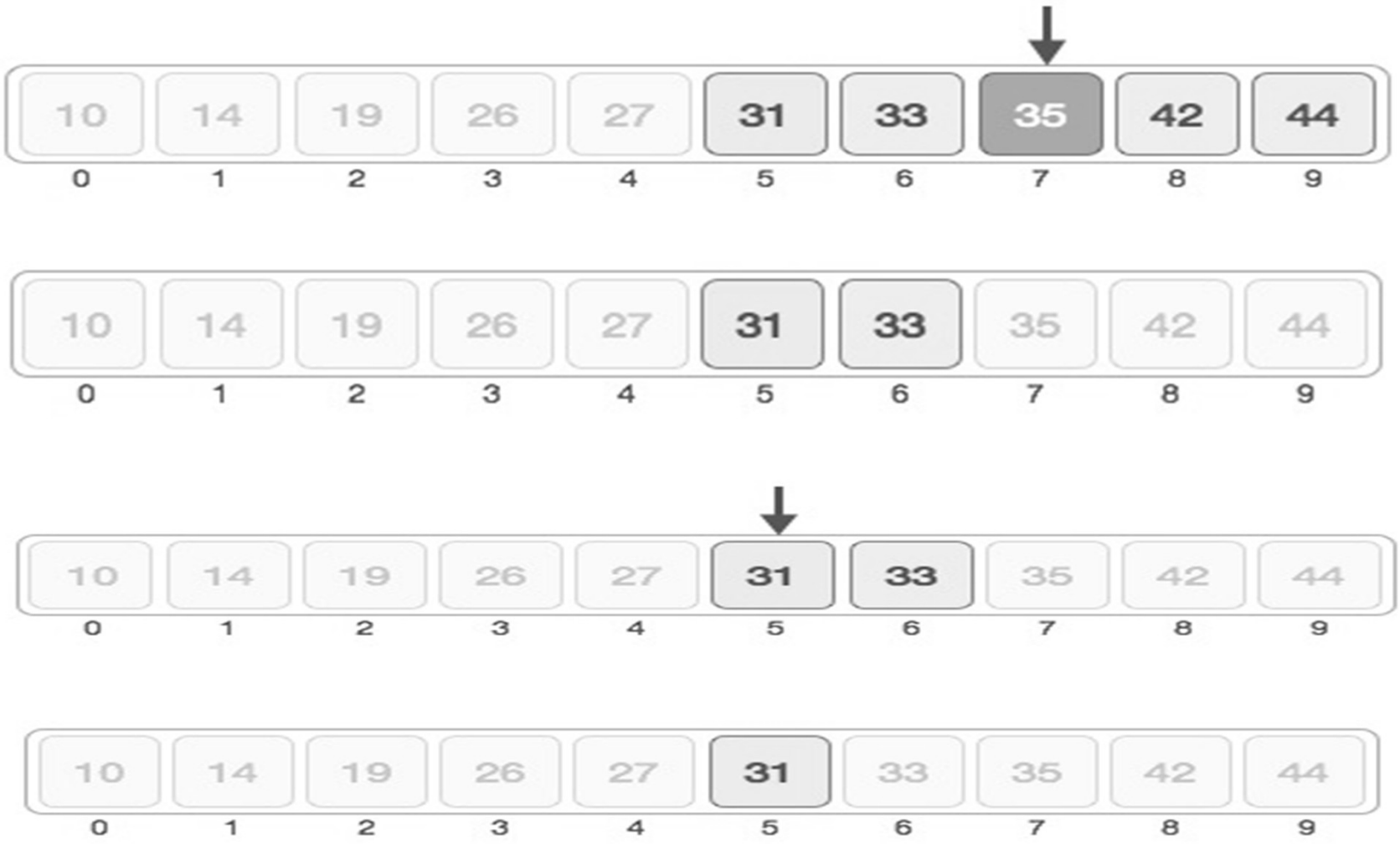
- Divide the search space into two halves by finding the middle index “mid”.
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search 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.

Binary Search

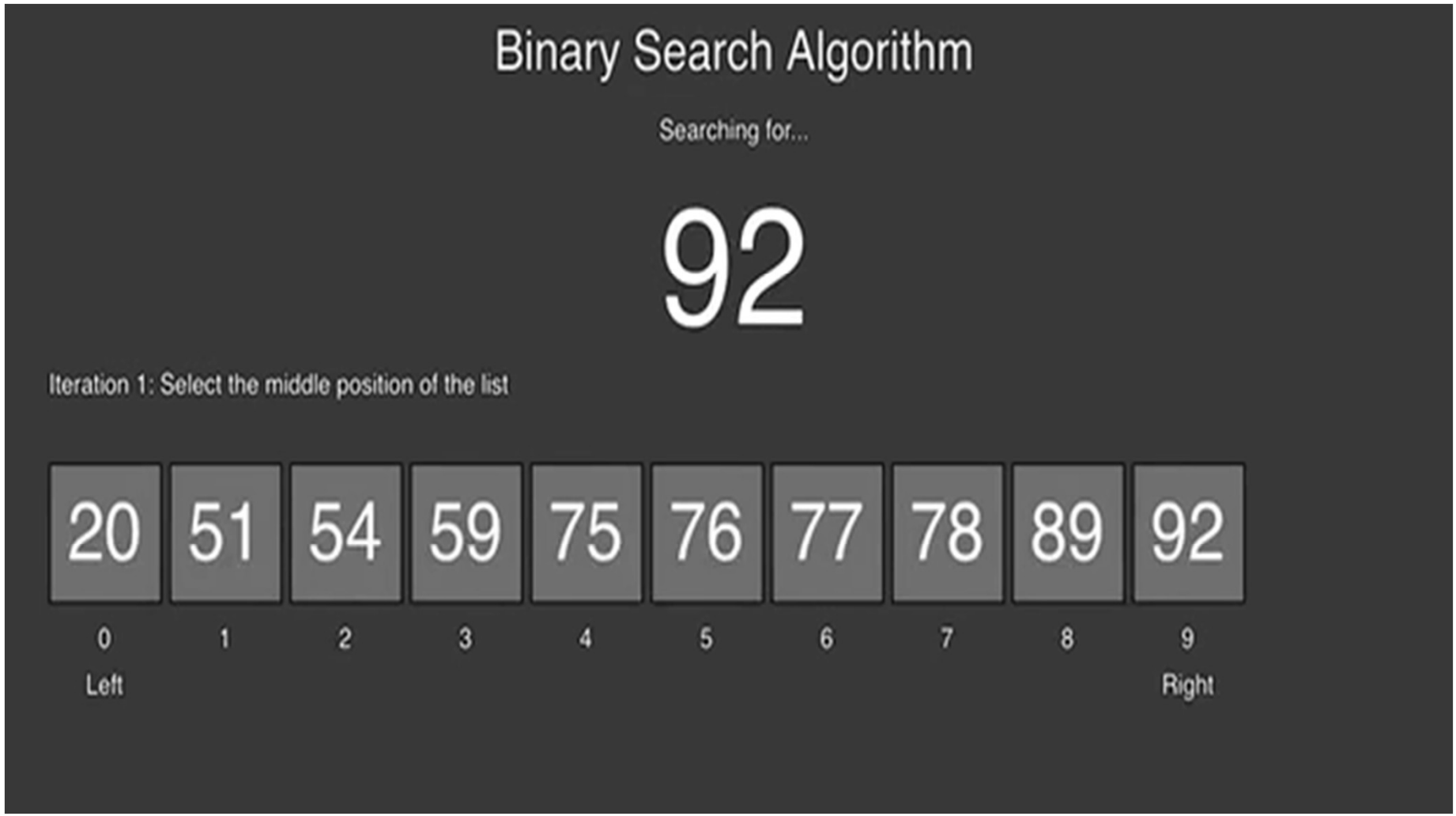

```
mid = low + (high - low) / 2
```



Binary Search



Binary Search



Iterative Binary Search

```
int binarySearch(int arr[], int low, int high, int x)
{
   while (low <= high) {
      int mid = low + (high - low) / 2;

      // Check if x is present at mid
      if (arr[mid] == x)
         return mid;

      // If x greater, ignore left half
      if (arr[mid] < x)
         low = mid + 1;

      // If x is smaller, ignore right half
      else
         high = mid - 1;
   }

   // If we reach here, then element was not present
   return -1;
}
```

Recursive Binary Search

```
int binarySearch(int arr[], int low, int high, int x)
{
   if (high >= low) {
      int mid = low + (high - low) / 2;

      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;

      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, low, mid - 1, x);

      // Else the element can only be present
      // in right subarray
      return binarySearch(arr, mid + 1, high, x);
   }
  return -1;
}
```

In recursive approach, we create a recursive function and compare the mid of the search space with the key. And based on the result either return the index where the key is found or call the recursive function for the next search space.

Benefits

- A **binary search algorithm** is a fairly simple search algorithm to implement.
- It is a significant improvement over linear search and performs almost the same in comparison to some of the harder to implement search algorithms.
- The **binary search algorithm** breaks the list down in half on every iteration, rather than sequentially combing through the list. On large lists, this method can be really useful.

Time Complexity

For each recurrence in the recurrence relation for binary search, we convert the problem into one subproblem, with runtime T(N/2). Therefore:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Substituting into the master theorem, we get:

$$\mathrm{T(n)} = \mathrm{aT}\left(\frac{n}{b}\right) + f(n)$$
$$a = 1, b = 2, f(n) = 1$$

Now, because $log_b a$ is 0 and f(n) is 1, we can use the second case of the master theorem because:

$$f(n) = O(1) = O(n^0) = O(n^{log_b a})$$

This means that:

$$T(n) = O(n^{log_b a} log(n)) = O(n^0 log(n)) = O(log(n))$$

$T(n) = aT(n/b) + f(n)$,

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, $a \geq 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function.

$T(n) = aT(n/b) + f(n)$

where, T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

$\epsilon > 0$ is a constant.

Thank You