Sorting and Searching

What is searching

What is sorting

Main types of searching strategies

Sequential

Binary / ternary …

Hashing

Main types of sorting strategies

Bubble sort
Insertion sort
Merge sort
Quick sort
Selection sort
Heap sort
….

Bubble sort

Bubble sort is a simple sorting algorithm.

This sorting algorithm is comparison-based algorithm

Each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Bubble sort

Its average and worst case complexities are of $O(n^2)$ where n is the number of items

This algorithm is not suitable for large data sets

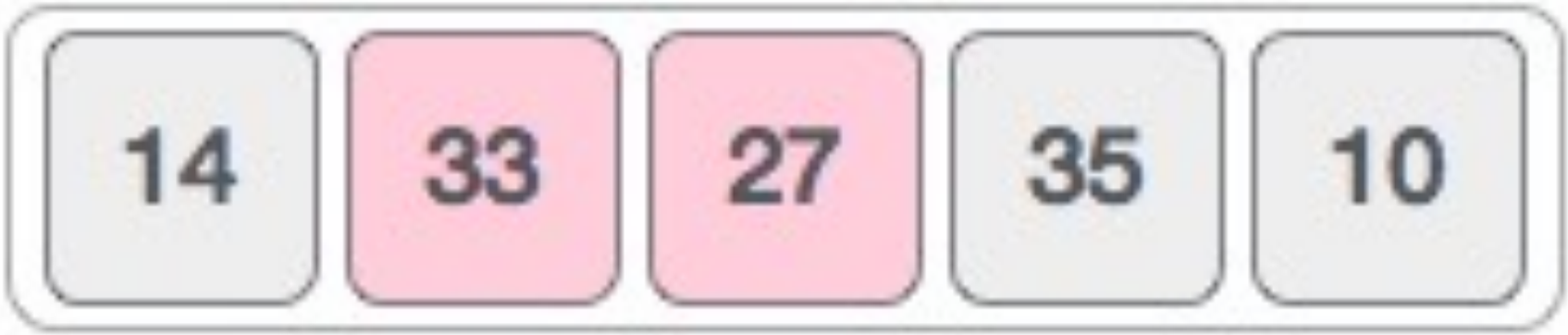
Bubble sort example

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations.

Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

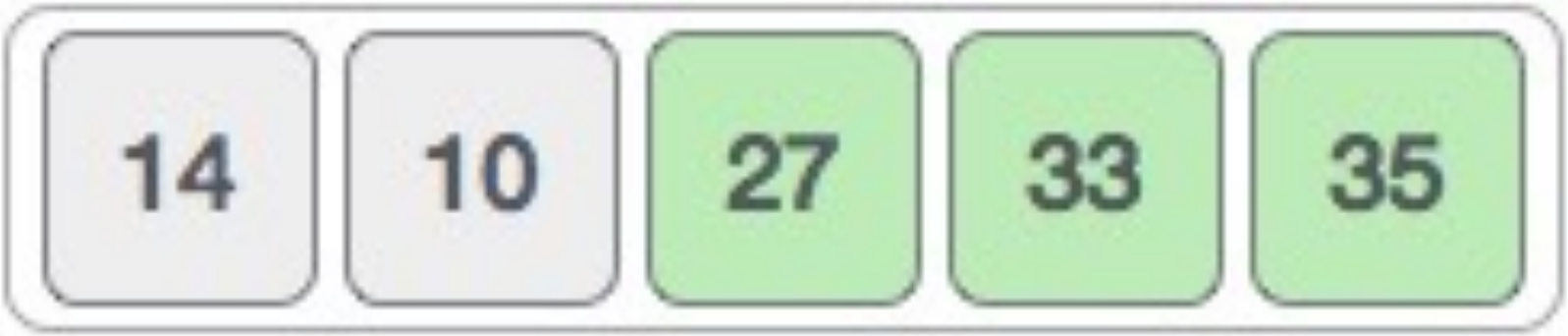
We swap these values.

We find that we have reached the end of the array. After one iteration, the array should look like this −

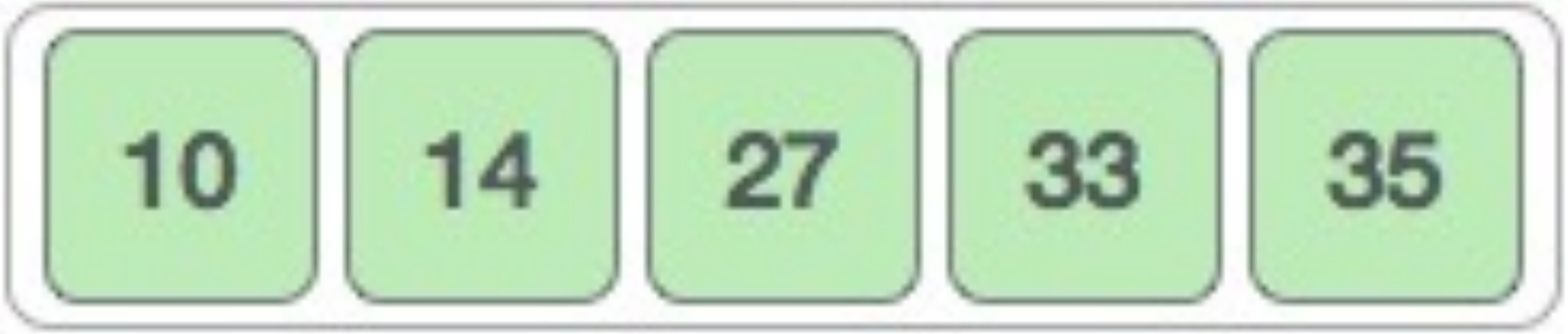
To be precise, let us see how an array should look like after each iteration.

After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.


14
10
27
33
35

And when there's no swap required, bubble sorts learns that an array is completely sorted.



```
int main()
{
  int array[100], n, c, d, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
```

```
for (c = 0 ; c < n - 1; c++)          // Loop for Pass
{
  for (d = 0 ; d < n – c - 1; d++)          // pairwise comp.
  {
    if (array[d] > array[d+1])
   /* For decreasing order use < */
    {
      temp       = array[d];                  // swap function…
      array[d]   = array[d+1];
      array[d+1] = temp;
    }
  }
}
```

For descending order –

```
for (c = 0 ; c < n - 1; c++)          // Loop for Pass
{
  for (d = 0 ; d < n – c - 1; d++)          // pairwise comp.
  {
    if (array[d] < array[d+1])              // Incorrect wrt dec.
   /* */
    {
      temp       = array[d];                // swap function…
      array[d]   = array[d+1];
      array[d+1] = temp;
    }
  }
}
```

```
  printf("Sorted list in ascending order:\n");

  for (c = 0; c < n; c++)
     printf("%d\n", array[c]);

  return 0;
}
```

Bubble sort program in C language using function

```
#include <stdio.h>
void bubble_sort(long [], long);

int main()
{
  long array[100], n, c;

  printf("Enter number of
elements\n");
  scanf("%ld", &n);

  printf("Enter %ld integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);

  bubble_sort(array, n);

 printf("Sorted list in
ascending order:\n");

  for (c = 0; c < n; c++)
     printf("%ld\n", array[c]);

  return 0;
}
```

```
void bubble_sort(long list[], long n)
{
  long c, d, t;
flag=1;
 for (c = 0 ; (c < n – 1)&&(flag==1); c++) {
           flag = 0;
    for (d = 0 ; d < n - c - 1; d++) {
      if (list[d] > list[d+1]) {
        /* Swapping */
        t         = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
           flag = 1;
      }
    }
  }
```

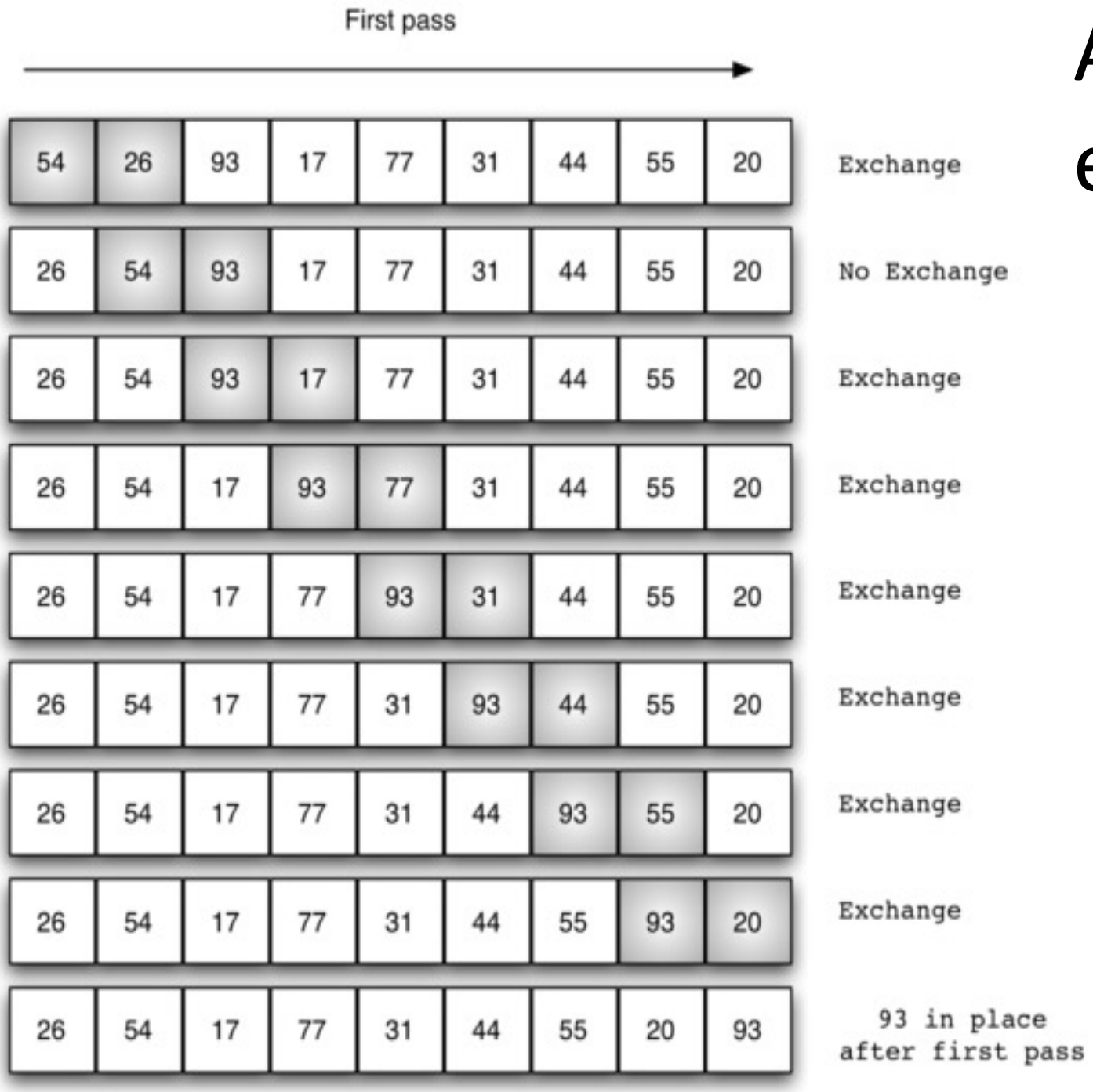
We can use the Bubble Sort algorithm to check if an array is sorted or not.

If no swapping takes place, then the array is sorted.

```
#include <stdio.h>
int is_Array_Sorted(int [], int);

int main()
{
  int a[100], n, c;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &a[c]);

  if (is_Array_Sorted(a, n))
    printf("The array is sorted.\n");
  else
    printf("The array isn't sorted.\n");

  return 0;
}
```

```
int is_Array_Sorted(int a[], int n) {
  int c, d, sorted = 1, t;

  for (c = 0 ; c < n - 1; c++) {
    for (d = 0 ; d < n - c - 1; d++) {
      if (a[d] > a[d+1]) {
        t = a[d];
        a[d] = a[d+1];
        a[d+1] = t;
        return 0;
      }
    }
  }
  return 1;
}
```

Another example



Recursive implementation of Bubble sort

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
        // Base case
        if (n == 1)
                return;

        // One pass of bubble sort. After
        // this pass, the largest element
        // is moved (or bubbled) to end.
        for (int i=0; i<n-1; i++)
                if (arr[i] > arr[i+1])
                        swap(arr[i], arr[i+1]);

        // Largest element is fixed,
        // recur for remaining array
        bubbleSort(arr, n-1);
}
```

```
void printArray(int arr[], int n)
{
        for (int i=0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

// Driver program to test above functions
int main()
{
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        int n = sizeof(arr)/sizeof(arr[0]);
        bubbleSort(arr, n);
        printf("Sorted array : \n");
        printArray(arr, n);
        return 0;
}
```

Linear Search

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order.

We start at one end and check every element until the desired element is not found.

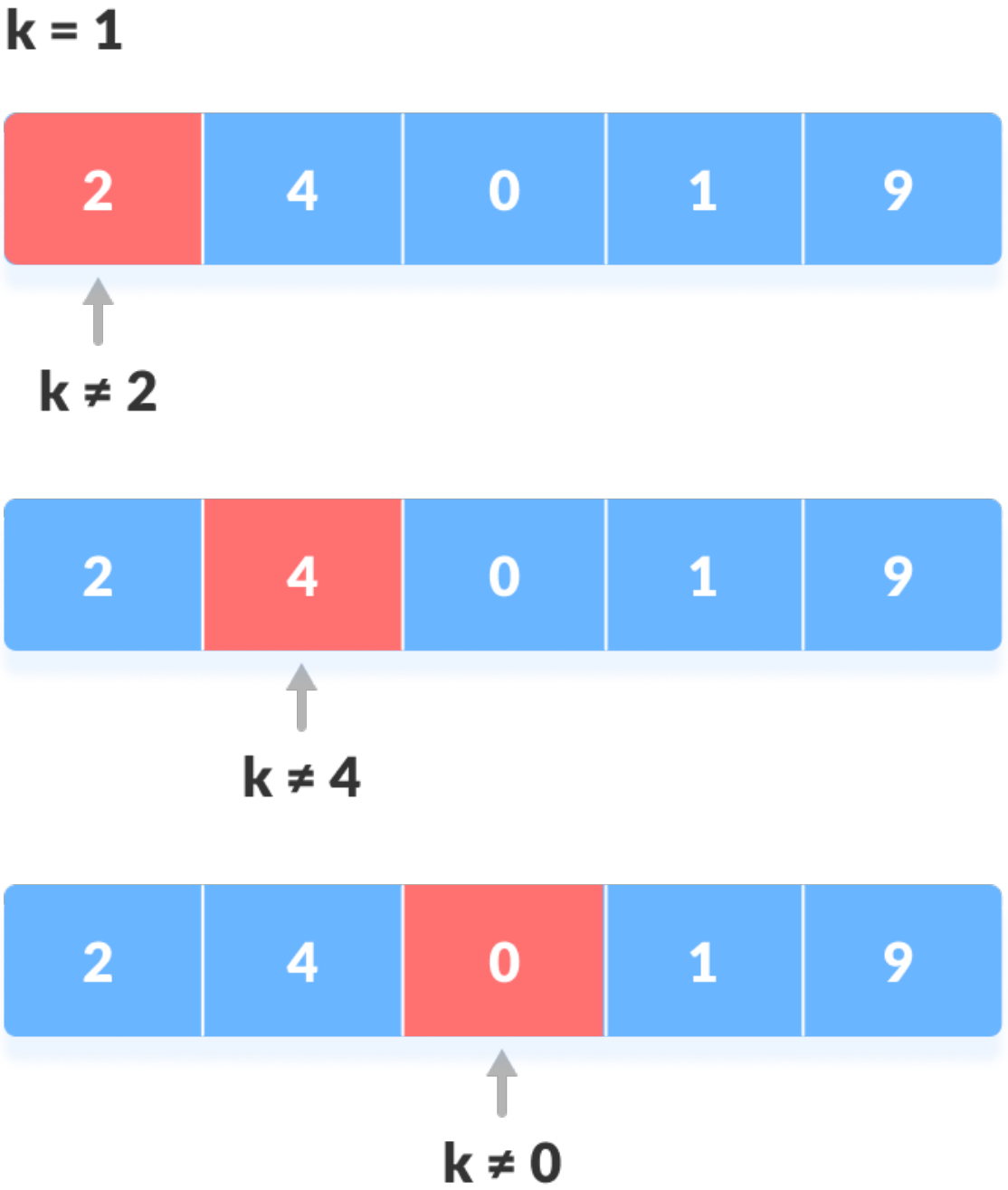
How Linear Search Works?

The following steps are followed to search for an element k = 1 in the list below.

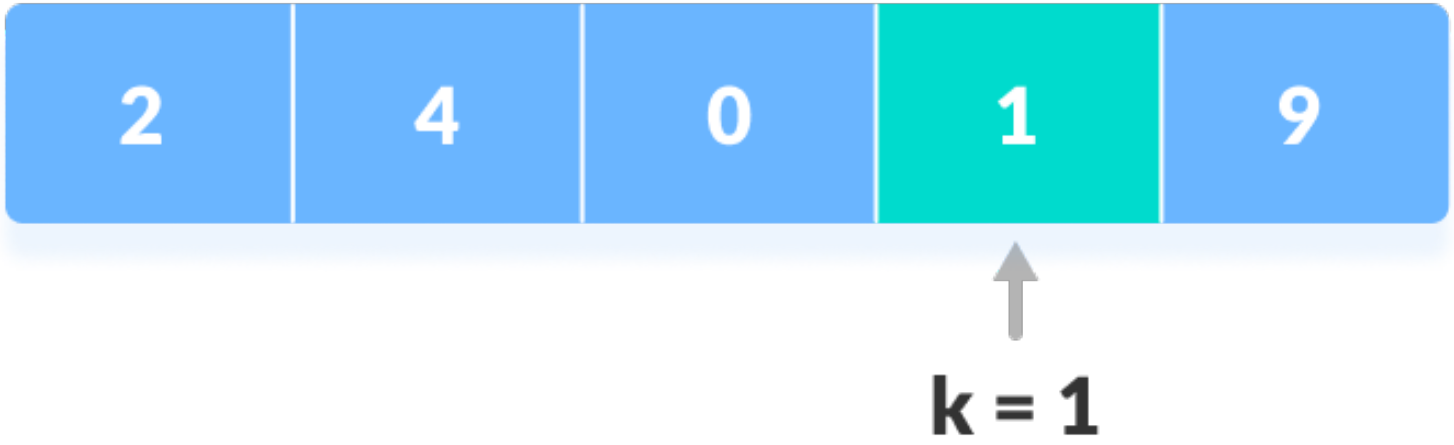
Array to be searched for

Start from the first element

compare k with each element x.



If x == k, return the index.



Else, return not found.

Linear Search Algorithm

```
LinearSearch(array, key)
  for each item in the array
    if item == value
      return its index
```

```
// Linear Search in C

#include <stdio.h>

int search(int array[], int n, int x)
{
  for (int i = 0; i < n; i++)
    if (array[i] == x)
      return i;
  return -1;
}
```

```
int main()
{
  int array[] = {2, 4, 0, 1, 9};
  int x = 1;
  int n = sizeof(array) / sizeof(array[0]);

  int result = search(array, n, x);

  (result == -1) ? printf("Element not found") :
printf("Element found at index: %d", result);

}
```

Comparison mechanisms

- Multiple strategies
- Starting from the naïve strategy
- Better strategy

- Time complexity -
- Space complexity

Comparison mechanisms

- In case of linear search
- The time complexity is F(N) = (N + 1)/2
- Best case is 1 comparison
- Worst case is N comparison
- F(N) = (N+1) / 2 = (½).N + (½) = O(N)

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array.

In this approach, the element is always searched in the middle of a portion of an array.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

- 1 3 56 8
- 1 3 8 56
- ……
- 2 5 1 7
- 1 2 5 7

Binary Search Algorithm can be implemented in two ways which are discussed below.

Iterative Method
Recursive Method

The recursive method follows the **divide and conquer** approach.

The general steps for both methods are discussed below.

The array in which searching is to be performed is:

Let x = 4 be the element to be searched.

Set two pointers (indices) low and high at the lowest and the highest positions respectively.

Find the middle element mid of the array ie. **arr[(low + high)/2] = 6.**

If x == mid, then return mid.

Else, compare the element to be searched with m.

If x > mid, compare x with the middle element of the elements on the right side of mid.

This is done by setting low = mid + 1.

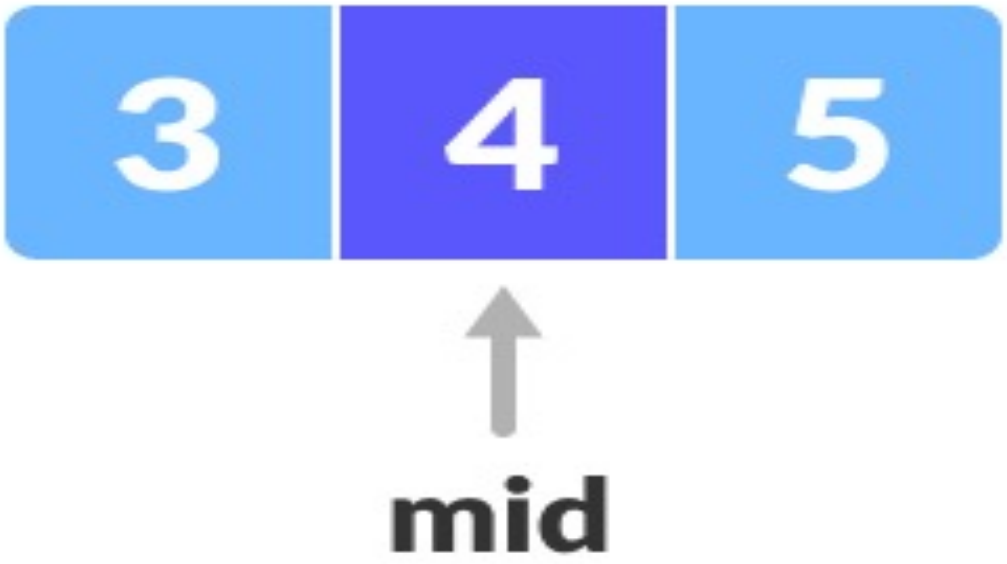
Else, compare x with the middle element of the elements on the left side of mid.

This is done by setting high = mid - 1.


3
4
5
6
7
8
9
low
high

Repeat previous steps
until

low meets high.



x = 4 is found

Binary Search Algorithm

Iteration Method

do until the pointers low and high meet each other.

```
mid = (low + high)/2
if (x == arr[mid])
    return mid
else if (x > arr[mid]) // x is on the right side
    low = mid + 1
else                   // x is on the left side
    high = mid - 1
```

Recursive Method

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > data[mid]       // x is on the right side
            return binarySearch(arr, x, mid + 1, high)
        else                        // x is on the right side
            return binarySearch(arr, x, low, mid - 1)
```

```
int binarySearch(int array[], int x, int low, int high)
{
        while (low <= high)
        {
                int mid = low + (high - low) / 2;
                if (array[mid] == x)
                        return mid;
                if (array[mid] < x)
                        low = mid + 1;
                else
                        high = mid - 1;
        }
        return -1;
}
```

```
int main(void)
{
	int array[] = {3, 4, 5, 6, 7, 8, 9};
	int n = sizeof(array) / sizeof(array[0]);
	int x = 4;
	int result = binarySearch(array, x, 0, n - 1);
	if (result == -1)
		printf("Not found");
	else
		printf("Element is found at index %d",
result);
	return 0;
}
```

```
#include <stdio.h>
int binarySearch(int array[], int x, int low, int high)
{
        if (high >= low)
        {
                int mid = low + (high - low) / 2;
                if (array[mid] == x)
                        return mid;
                if (array[mid] > x)
                        return binarySearch(array, x, low,
mid - 1);
                return binarySearch(array, x, mid + 1, high);
        }
        return -1;
}
```

```
int main(void)
{
	int array[] = {3, 4, 5, 6, 7, 8, 9};
	int n = sizeof(array) / sizeof(array[0]);
	int x = 4;
	int result = binarySearch(array, x, 0, n - 1);
	if (result == -1)
		printf("Not found");
	else
		printf("Element is found at index %d",
result);
}
```

Time complexity of binary search

- Always in every step we divide the array into two parts – until we get the number
- X
- X/2
- X/4
- X/8
- X/16
- ……. 1