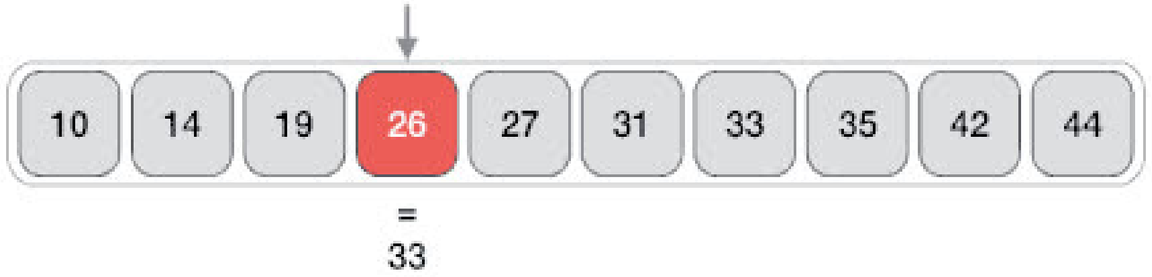
SEARCHING AND SORTING

Searching Algorithm

Linear Search

Binary Search

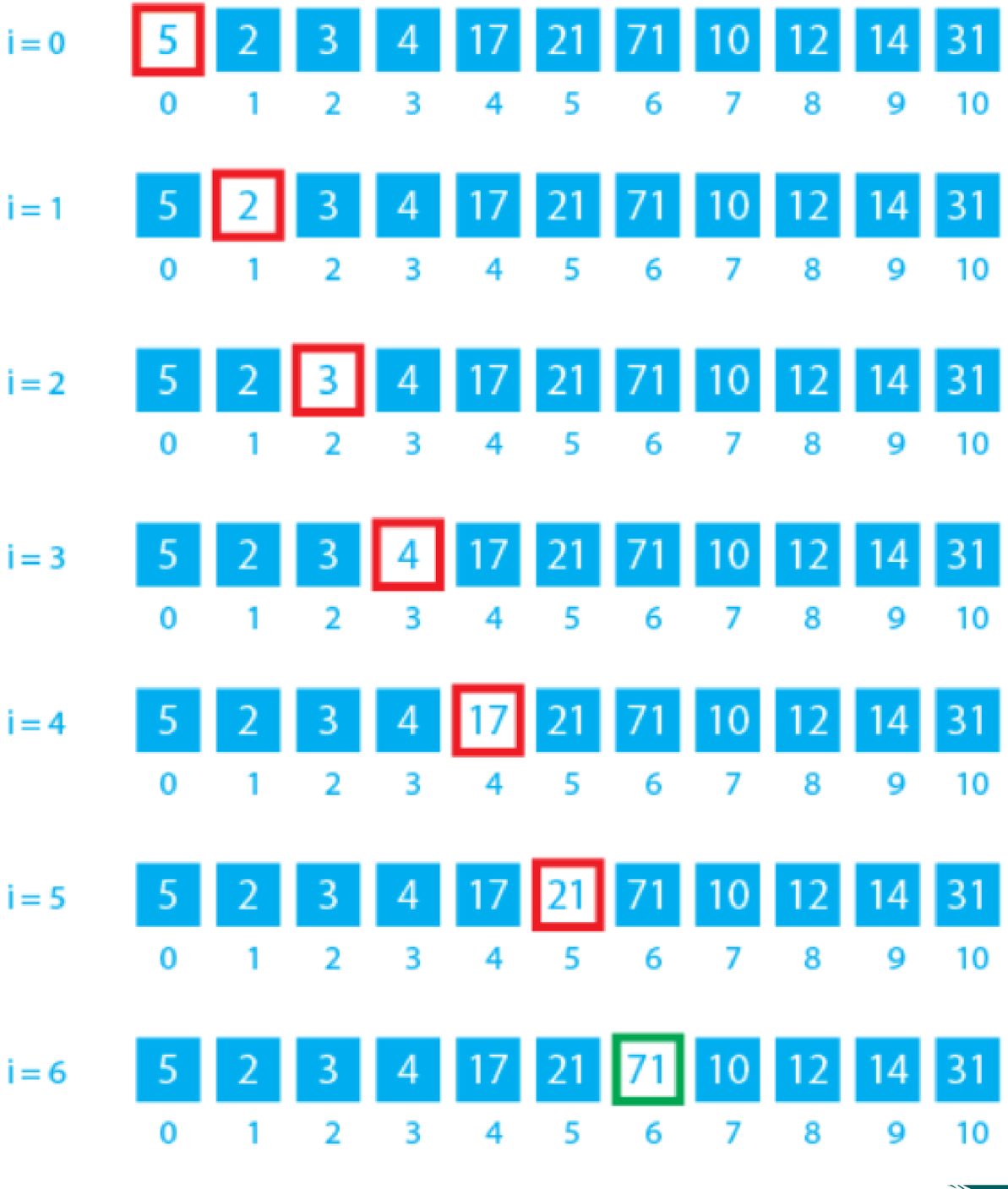
Linear **Search In Data Structure**



LINEAR SEARCH

- list is traversed from one end to other

```
int LS (int arr[] ,  int size ,  int
value)
      for(int i=0 ; i<size ; i++)
      {
            if(arr[i] == value)
                  return i;
       }
       return -1;
}
```


i = 0
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10
i = 1
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10
i = 2
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10
i = 3
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10
i = 4
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10
i = 5
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10
i = 6
5 2 3 4 17 21 71 10 12 14 31
0 1 2 3 4 5 6 7 8 9 10


WHICH SEARCH DID YOU USE
LINEAR SERACH
WHAT DID IT COST?
TIME COMPLEXITY

TIME COMPLEXITY FOR LINEAR SEARCH

As we have to traverse the whole array for the worst case so the time complexity wil be O(n)


BINARY SEARCH
PROGRAMMER
Linear Search

BINARY SEARCH

BINARY SEARCH

Why binary search ?

The linear search takes a large amount of time to complete.

- Linear search is useful on very small sized arrays
- For very large search space binary search is an efficient method.


Binary search endings it's career
Linear search thinking it's cool

BINARY SEARCH

What is binary search ?

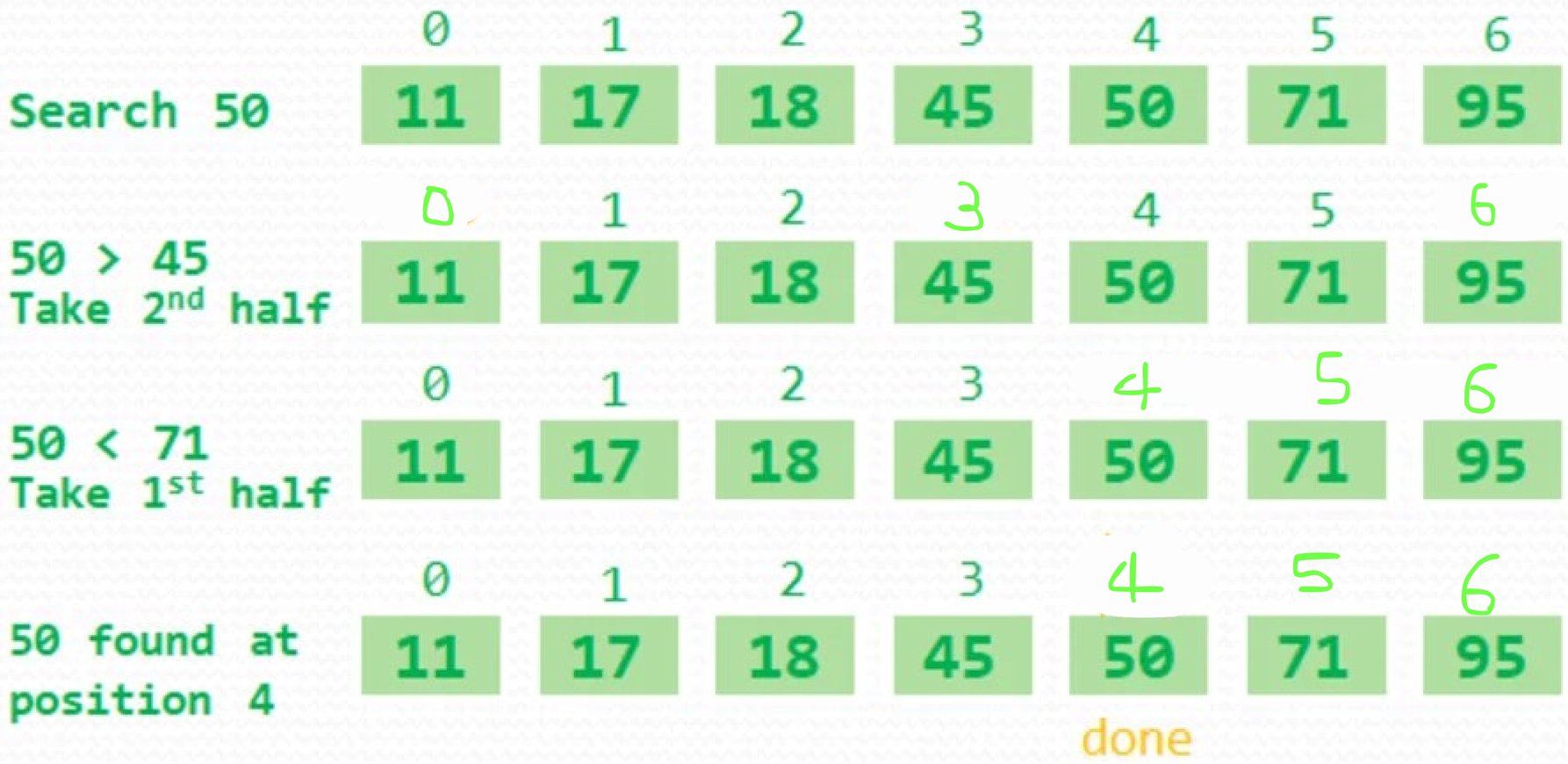
- list or dataset is divided repeatedly into half
- used on sorted array
- mid value is calculated.

BINARY SEARCH

A condition for binary search :

- The list/array must be sorted.

Binary Search



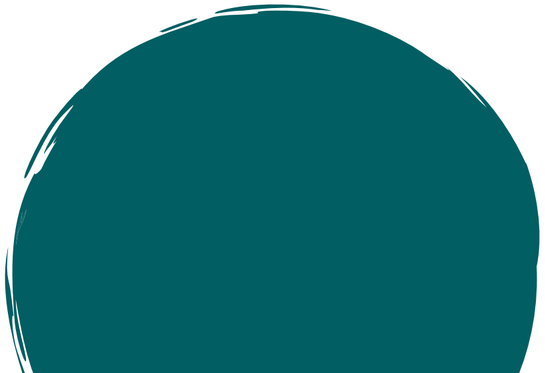
TIME COMPLEXITY FOR BINARY SEARCH

BEST CASE :

Best case is when the element is at the middle index of the array. It takes only one comparison to find the target element. So the best case complexity is O(1).

WORST CASE :

The time complexity for the worst case is O(log N).

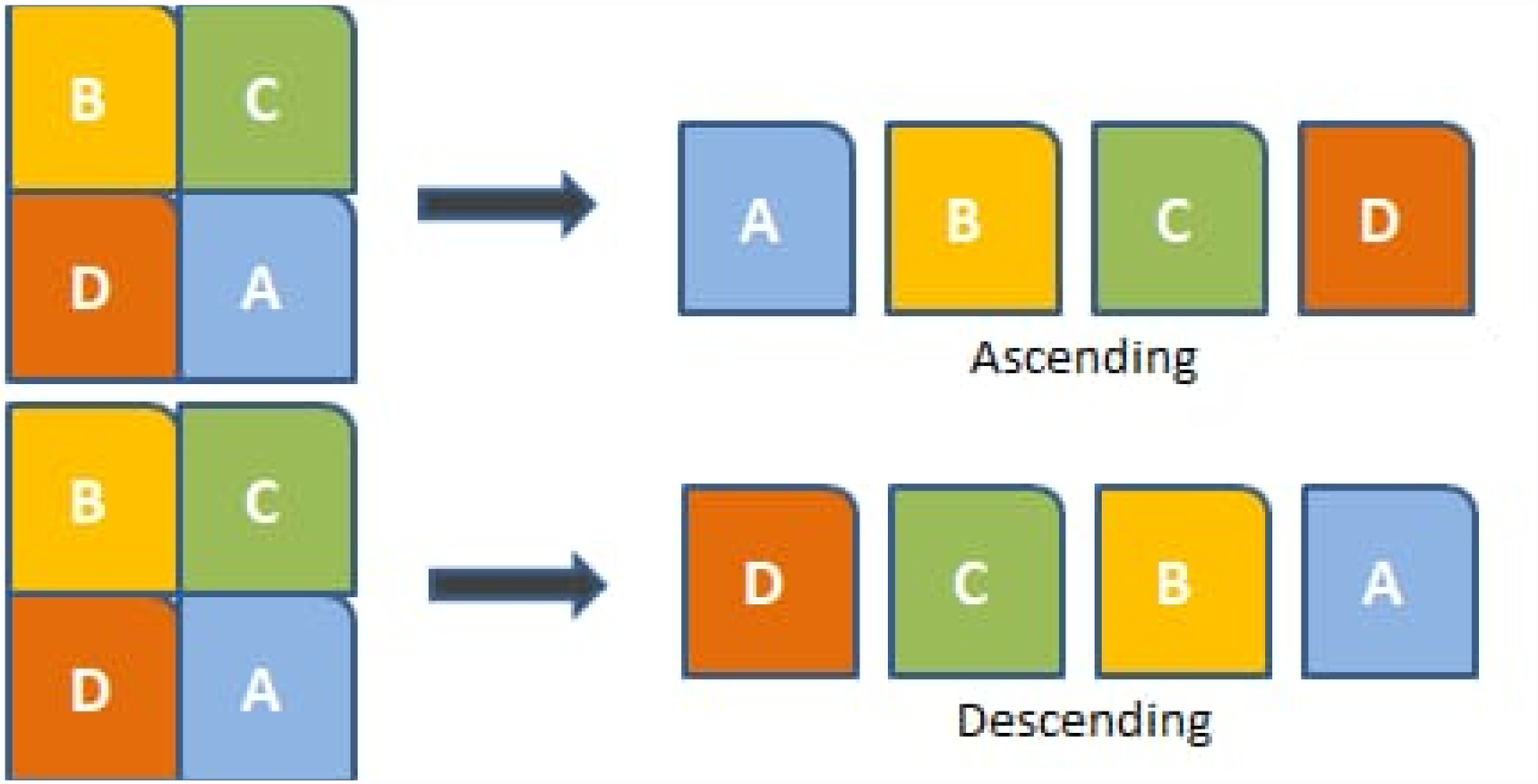


Q.Finding Square Root of a Number

Given a number find the square root of it.

eg.
number = 36
root : 6

Sorting



Selection Sort

Finding the minimum element in unsorted array and then swap with element with begining.

```
int  sortingincresing(int arr[],int n)
{
    int i;
    for( i=0;i<(n-1);i++)
    {

        int minindex=i;
        for(int j=i+1;j<n;j++)
        {
            if(arr[j]<arr[minindex])
            {
                minindex=j;
            }
        }

        swap(arr[minindex],arr[i]);
    }
    return arr[i];
}
```

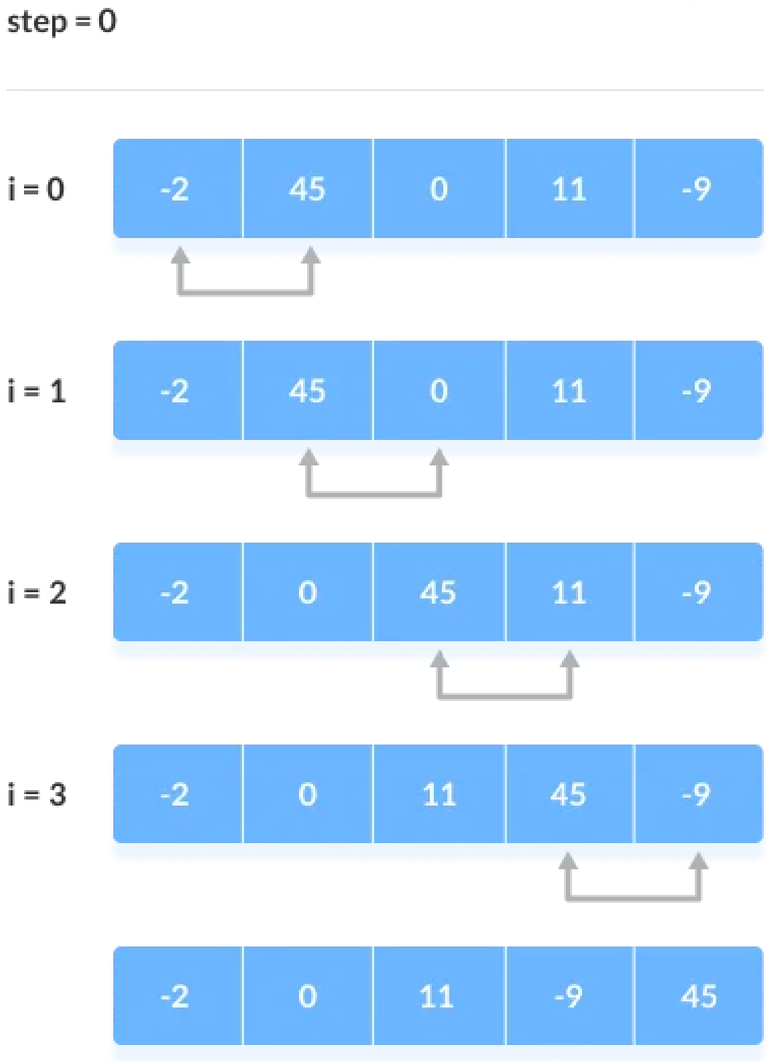
Time Complexity

Best Case: O(n^2)

Worst case : O(n^2)

Bubble Sort

Repeatedly compare two adjacent elements and if they are in wrong order swap



```
int bubblesort(int arr[],int n)
{
    int i,j;
    for(i=0;i<(n-1);i++)
    {
        int flag=1;
        for(j=0;j<(n-1);j++)
        {
            if(arr[j+1]<arr[j])
            {
                swap(arr[j],arr[j+1]);
                flag=0;
            }
        }
        if(flag==1)
         break;
    }
    return arr[j];
}
```

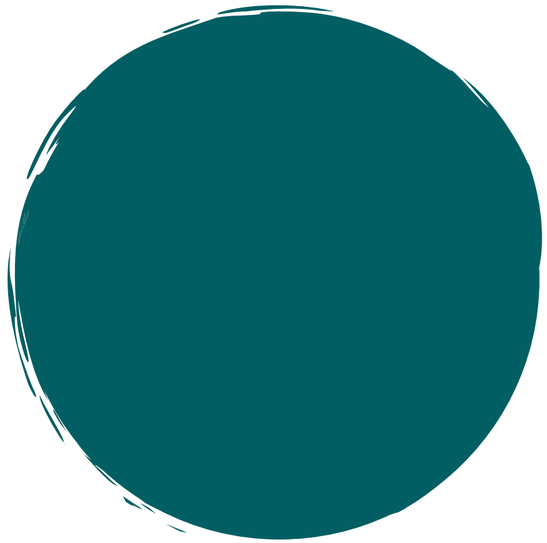
TIME COMPLEXITY :

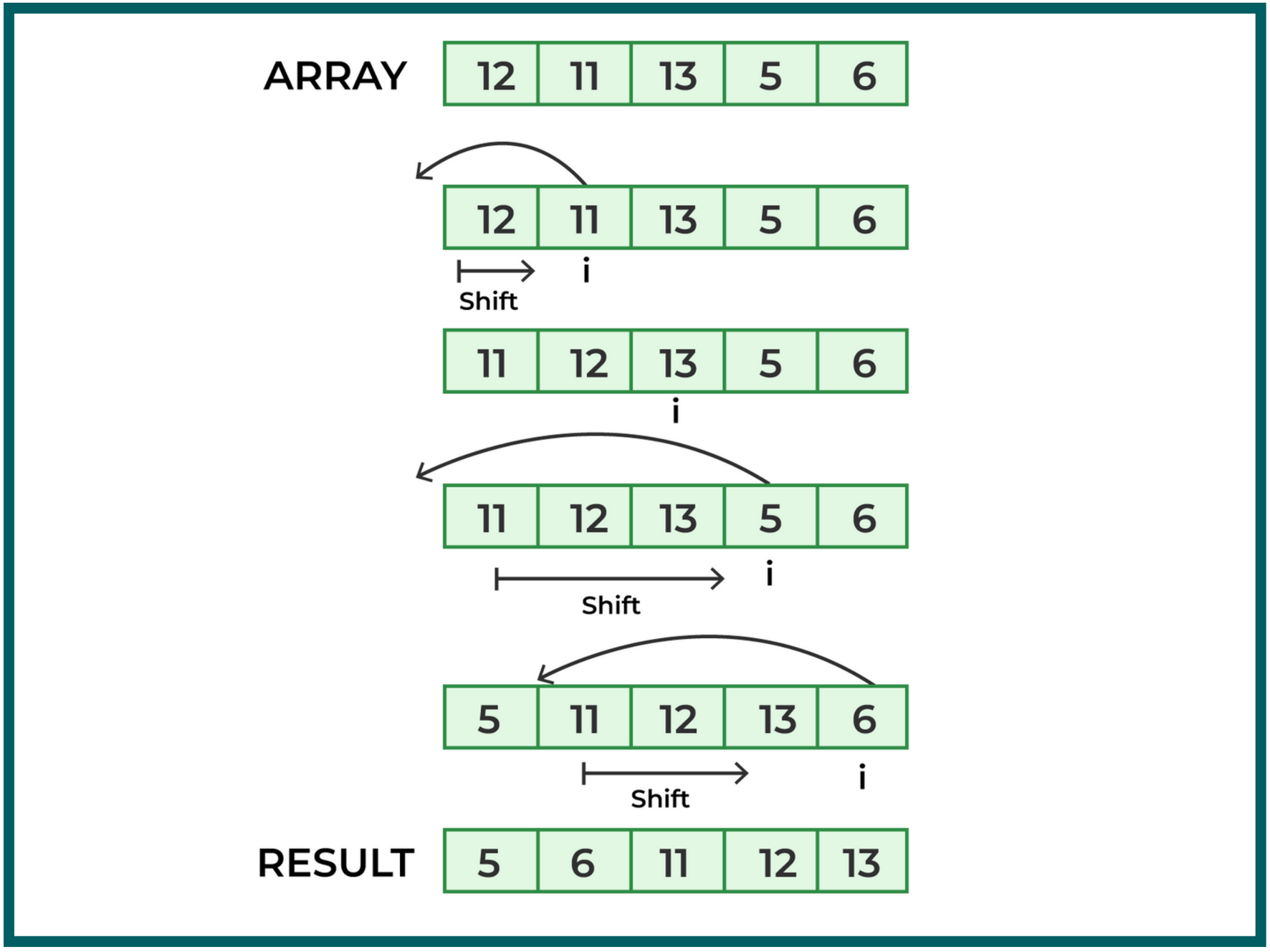
Best Time Complexity : O(n)

Worst Time Complexity : O(n^2)

If there is no swapping is a pass,
then the array is sorted completely

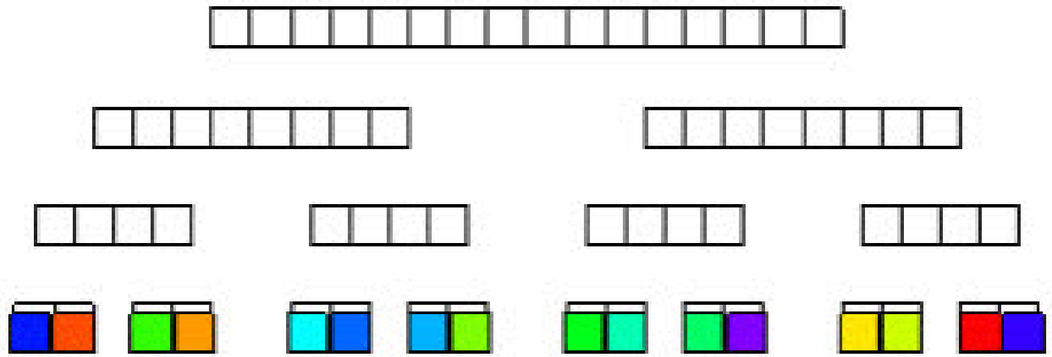
INSERTION SORT




ARRAY
12 11 13 5 6
12 11 13 5 6
i
Shift
11 12 13 5 6
i
11 12 13 5 6
i
Shift
5 11 12 13 6
i
Shift
RESULT
5 6 11 12 13

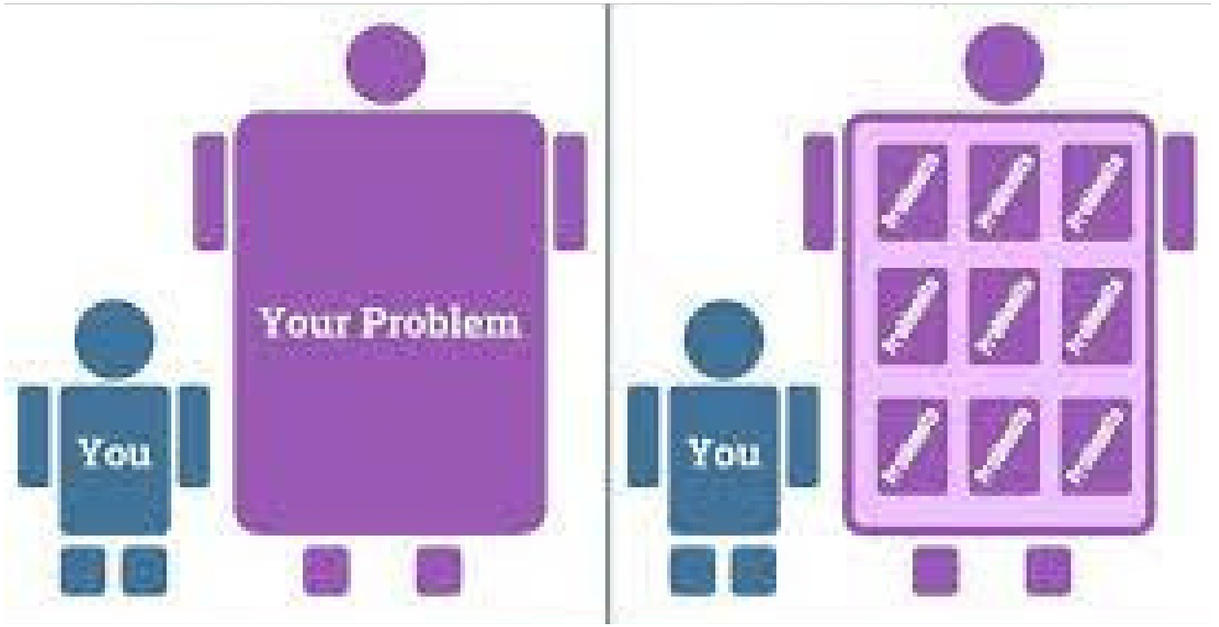

WAITING FOR
PROGRAM TO FINISH
WHEN YOU APPLY INSERTION
SORT ON LARGE SIZE ARRAY

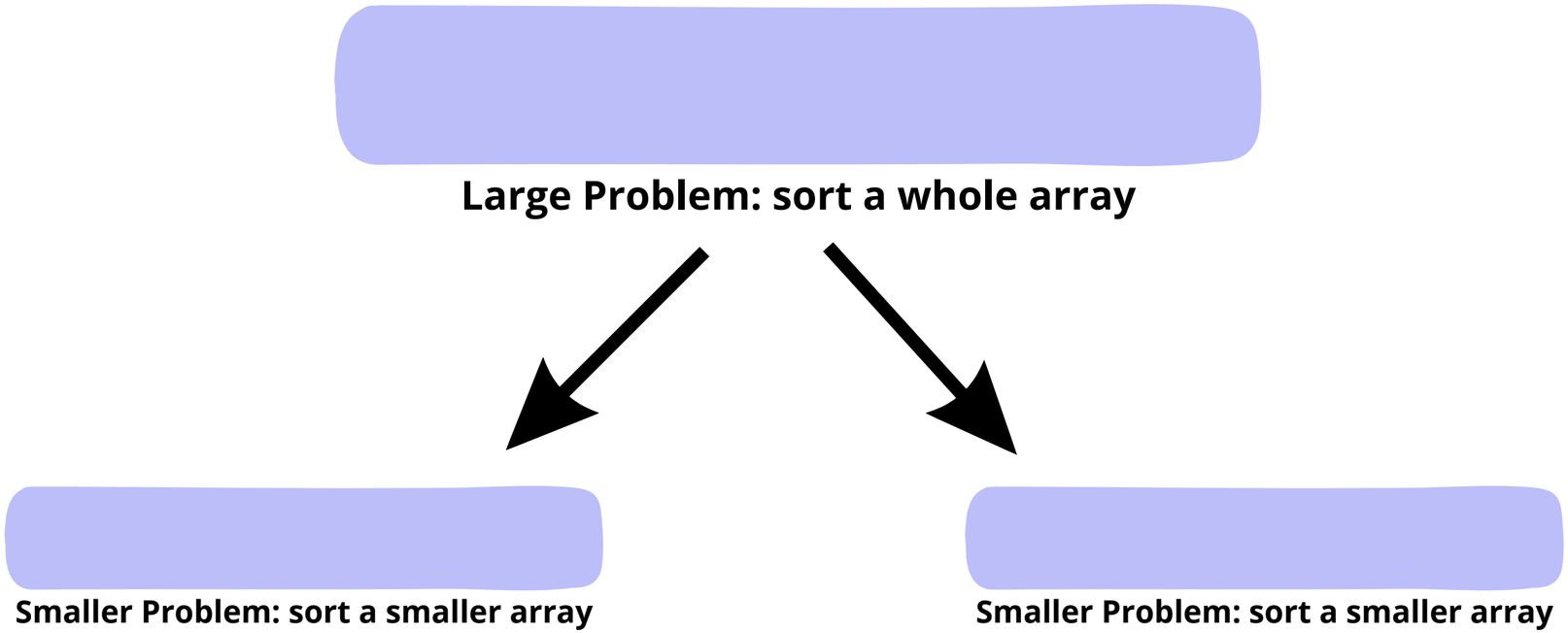
MERGE SORT




MERGE SORT
AN ARRAY
MERGE SORT
HALF AN ARRAY
HALF AN ARRAY

- Merge Sort is based on **Divide and Conquer**
- A Problem is divided into multiple sub-problems.
- Each Subproblem is solved and we combine a result of subproblems to get ans of larger problem.




Large Problem: sort a whole array
Smaller Problem: sort a smaller array
Smaller Problem: sort a smaller array

Should we divide array for an infinite time??

0	1	2	3	4	5	6	7
10	15	1	18	100	12	21	1


mergeSort(Arr, p, r):
1. if left > right return
2. mid = (left+right)/2
3. mergeSort(Arr, left, mid)
4. mergeSort(Arr, mid+1, right)
5. merge(Arr, left, mid, right)

```
merge(arr, low, mid, high)
1. Create a temporary array `temp`.
2. Initialize `left`=low and `right`= mid+1 pointers.
3. While both `left` and `right` are within their sub-arrays, do:
   - Compare elements and append the smaller to `temp`, then increment the pointer.
4. If elements remain in the left sub-array, copy them to `temp` (Case 1).
5. If elements remain in the right sub-array, copy them to `temp` (Case 2).
6. Transfer elements from `temp` to the original array `arr`.
```

Lets see the code!

Time Complexity: O(nlogn)

- **Best Case: O(n * $\log_2(n)$) - Rarely observed, occurs when the array is nearly sorted.**
- **Worst Case: O(n * $\log_2(n)$) - Happens when the array is in a reverse order.**
- **Average Case: O(n * $\log_2(n)$) - Expected performance for most scenarios**

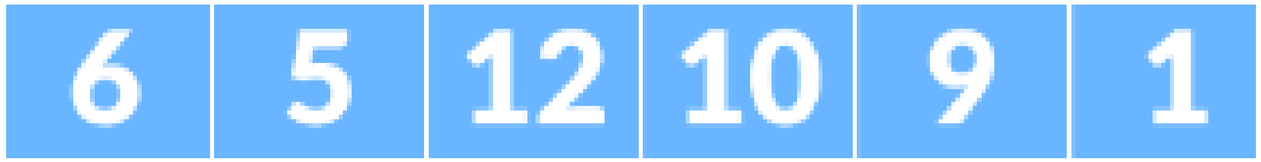
Space Complexity: ?

Merge Sort

Quick Sort

Sab log apne apne ghar jao

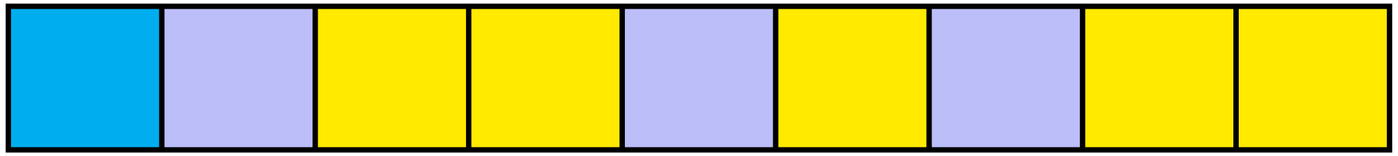
Quick Sort

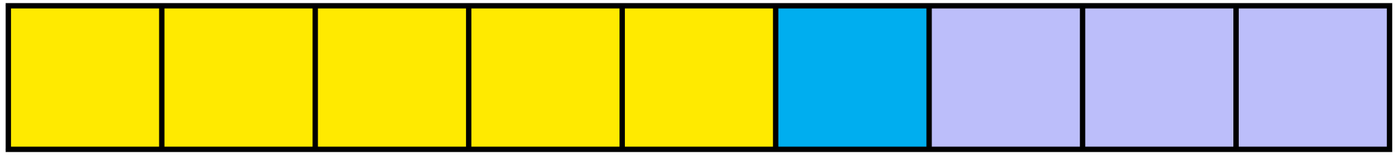


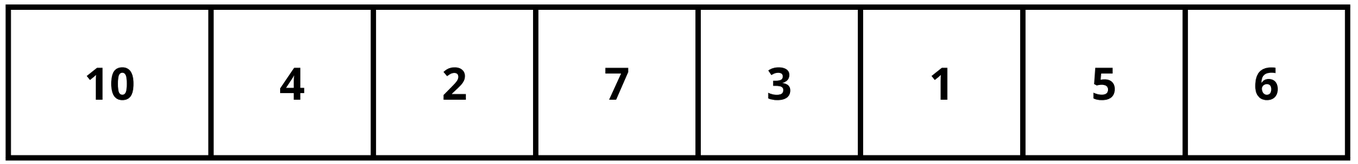
- Divide and Conquer Algorithm
- Pick a **Pivot element** and Place it at its correct position.

How to choose a pivot?

- Shift the smaller Elements left to pivot and greater elements to right of pivot.






10
4
2
7
3
1
5
6

```cpp
void quickSort(vector<int> &arr, int low, int high)
{    if (low < high) {
        int pIndex = partition(arr, low, high);
        quickSort(arr, low, pIndex - 1);
        quickSort(arr, pIndex + 1, high);
    }
}
```

```cpp
int partition(vector<int> &arr, int low, int high)
{   int pivot = arr[low];
    int i = low;
    int j = high;

    while (i < j) {
        while (arr[i] <= pivot && i <= high - 1) {
            i++;
        }

        while (arr[j] > pivot && j >= low + 1) {
            j--;
        }
        if (i < j) swap(arr[i], arr[j]);
    }
    swap(arr[low], arr[j]);
    return j;
}
```

Time Complexity: O(nlogn)

- **Best Case: O(n * $\log_2(n)$) - Occurs when the array is nearly sorted.**
- **Worst Case: O(n^2) - Happens when the array is in reverse order or when a poor pivot strategy is chosen.**
- **Average Case: O(n * $\log_2(n)$) - Expected performance for most scenarios.**

Space Complexity: ?

Quick Sort vs Merge Sort

Quick Sort:

- **Efficient and often faster than Merge Sort.**
- **In-place sorting, conserving memory.**
- **Requires good pivot selection to avoid worst-case scenarios.**
- **Adapts well to various data types and performs well when optimized.**

Merge Sort:

- **Stable and consistent with O(n * $\log_2$(n)) time complexity for all cases.**
- **Uses additional memory for merging.**
- **Provides predictable performance regardless of data characteristics.**
- **No worst-case concerns like Quick Sort's potential O(n^2) worst-case time complexity.**

WHEN YOU UNDERSTAND
QUICK SORT SOMEHOW !!

Q.Search in 2D Matrix

Q.Search in Rotated Sorted Array


THANK
YOU