



# **Design and Analysis of Algorithms**

## **SORTING ALGORITHMS**

**Submitted By:**

**00304092023 Akanchha Singh**

**06304092023 Shweta Rawat**

**07104092023 Tarni Balgoher**

**Submitted To:**

**Ms. Preeti Vats**

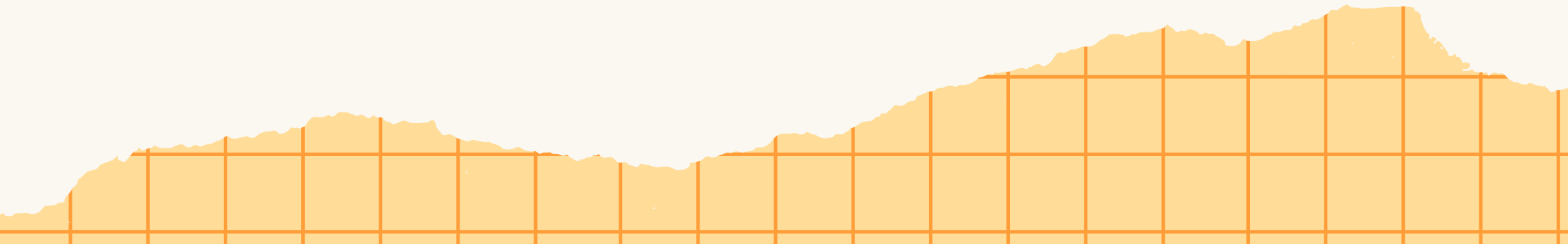
**Ms. Neetu Swaroop**





# Merge Sort

---

- Algo- Theory
  - Dry run
  - Pseudo code
  - TC and SC
  - Why use merge sort?
- 

# Algorithm

---

Merge Sort is a divide-and-conquer algorithm used to sort arrays efficiently.



## Steps of Merge Sort

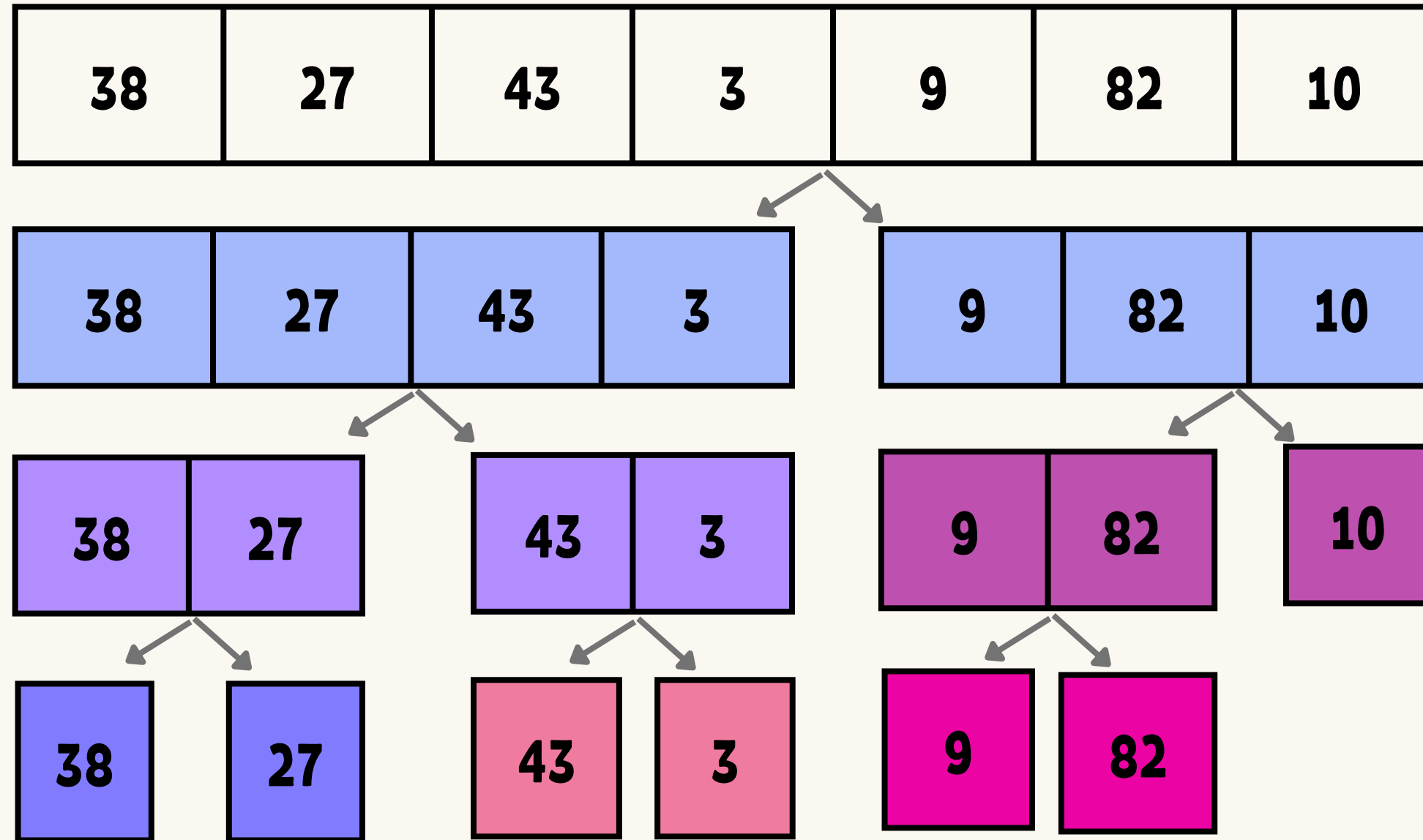
### 1. Divide the Array:

- Hypothetically split the array into two halves.
- Repeat this step recursively for each half until each subarray has only one element.

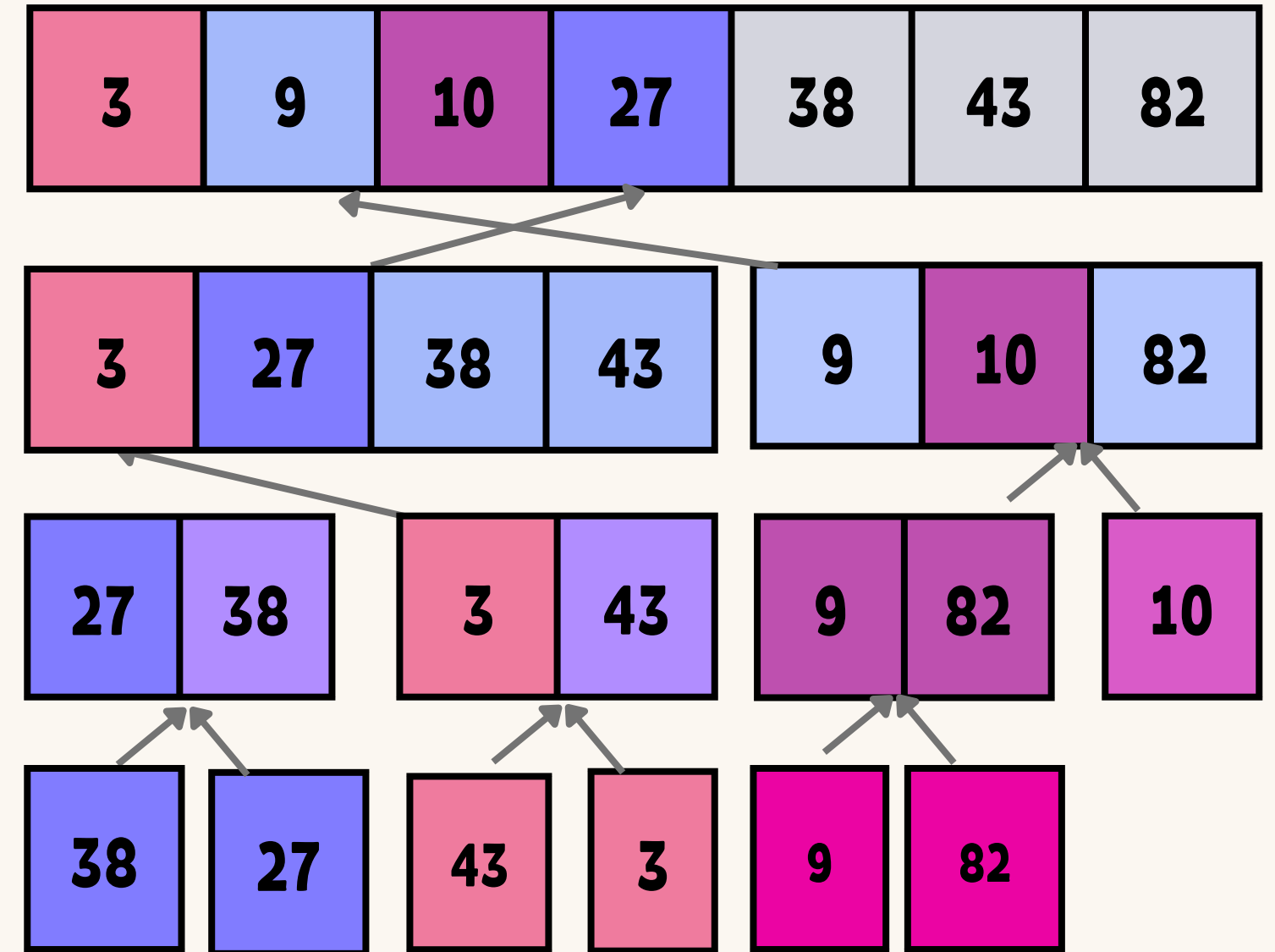
### 2. Merge the Arrays:

- Start merging the smallest arrays (with one element each).
- Compare elements from both arrays and arrange them in sorted order.
- Continue merging arrays until the whole array is sorted.

# Dry run



Divide



Merge

# Code

```
// Function to merge two halves
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of the left subarray
    int n2 = right - mid;    // Size of the right subarray

    vector<int> L(n1), R(n2); // Create temporary arrays

    for (int i = 0; i < n1; i++) // Copy data to temporary arrays
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left; // Merge the temporary arrays back into arr
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) { // Copy remaining elements of L[] (if any)
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) { // Copy remaining elements of R[] (if any)
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
// Function to perform Merge Sort
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Calculate the mid-point

        // Recursively divide and sort both halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

# Output

```
Original array: 38 27 43 3 9 82 10
Sorted array: 3 9 10 27 38 43 82
```

```
=== Code Execution Successful ===
```



# TC and SC

---

## Time Complexity:

1. Best Case:  $O(n \log n)$
2. Worst Case:  $O(n \log n)$
3. Average Case:  $O(n \log n)$

## Why?

- Dividing the array takes  $O(\log n)$
- Merging subarrays takes  $O(n)$

## Space Complexity:

- $O(n)$  due to temporary arrays used during merging.
- 

# Why use merge sort?

---

## Why is Merge Sort better than Shell Sort and Insertion Sort?

- **Scalability:** Handles large datasets efficiently due to consistent  $O(n \log n)$  time complexity.
- **Stable Sorting:** Maintains relative order of equal elements, unlike Shell Sort.
- **Not Input Dependent:** Performs equally well regardless of input order, unlike Insertion Sort, which has  $O(n^2)$  for unsorted data.

## Real-Life Use Cases of Merge Sort

- **Sorting Large Files (External Sorting):** Efficiently sorts files too large to fit in memory (e.g., log files or database records).
- **Inversion Count in Arrays:** Calculates the number of "out-of-order" pairs in datasets (used in statistical analysis).
- **Sorting Linked Lists:** Works efficiently for linked lists since it doesn't rely on random access.

**insertion sort**



## Approach:

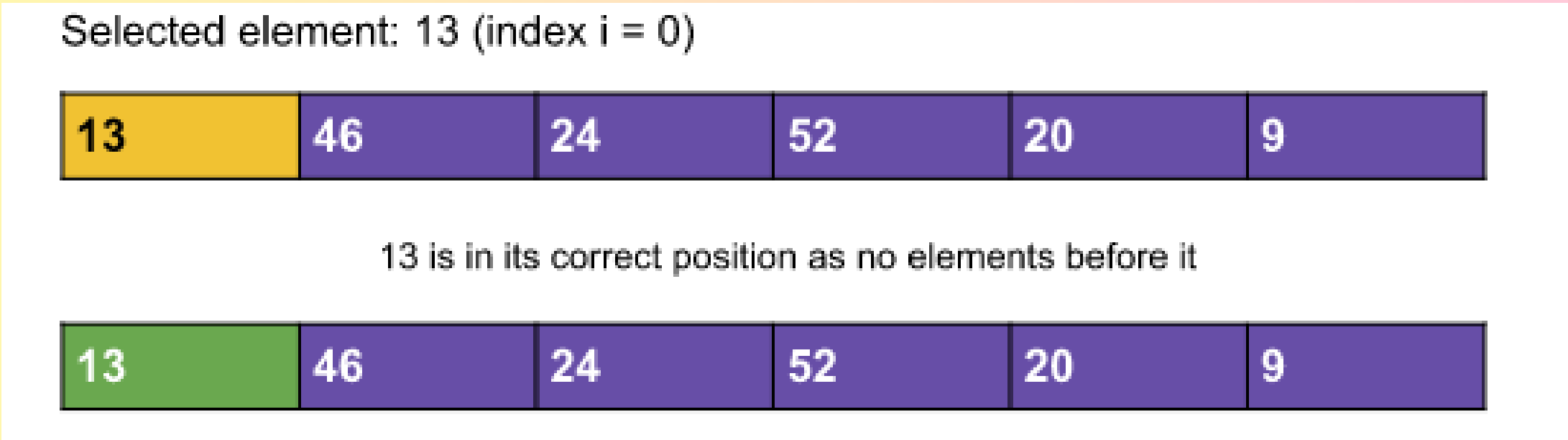
- Select an element in each iteration from the unsorted array(using a loop).
- Place it in its corresponding position in the sorted part and shift the remaining elements accordingly (using an inner loop and swapping).
- The “inner loop” basically shifts the elements using swapping.

## Dry Run:

- The purple color represents the unsorted array.
- The yellow color represents the current element that needs to be placed in the appropriate position.
- The green color represents the sorted array.

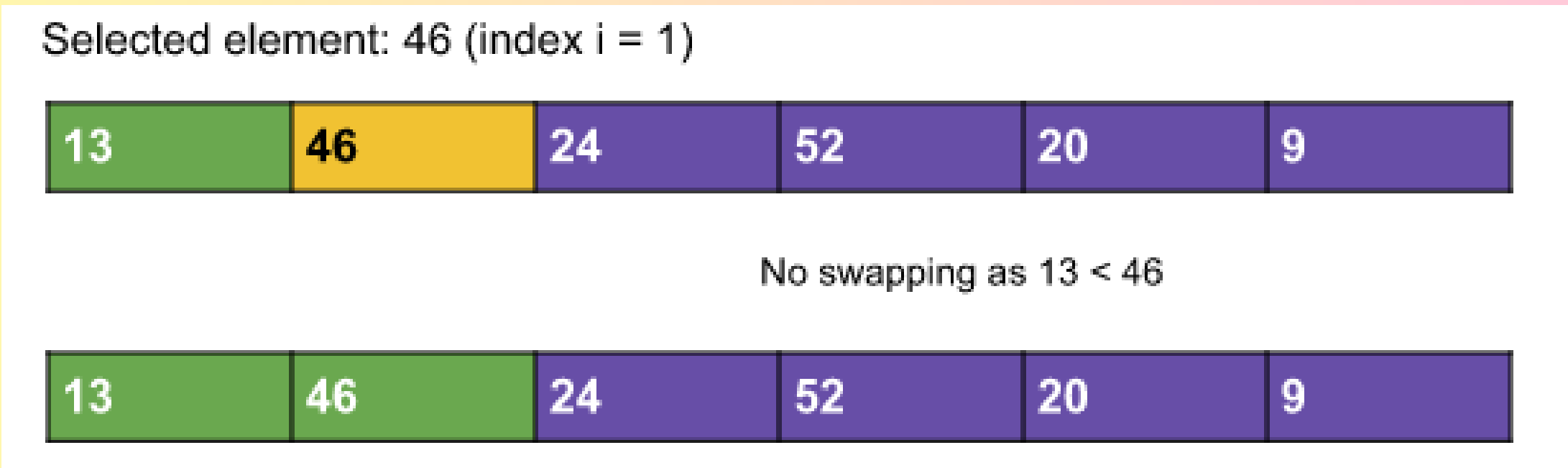
**Outer loop iteration 1(selected index i = 0):**

The element at index i=0 is 13 and there is no other element on the left of 13. So, currently, the subarray up to the first index is sorted as it contains only element 13.



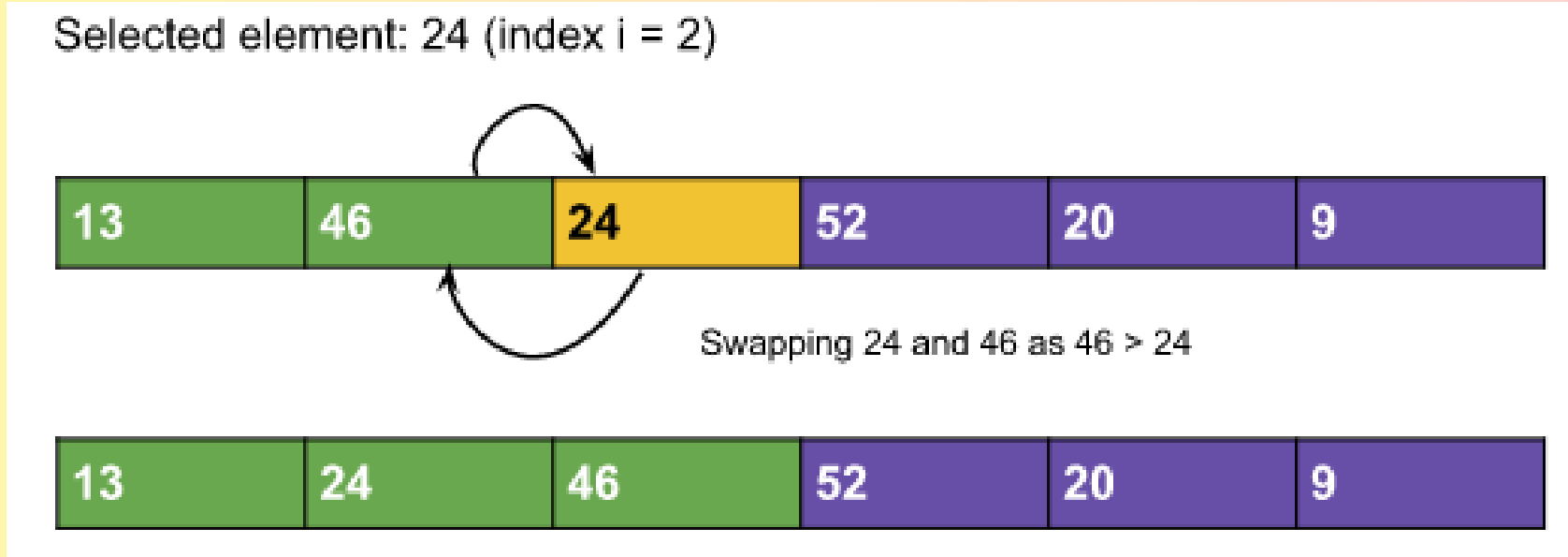
**Outer loop iteration 2(selected index i = 1):**

The selected element at index i=1 is 46. Now, we will try to move leftwards and put 46 in its correct position. Here,  $46 > 13$  and so 46 is already in its correct position. Now, the subarray up to the second index is sorted.



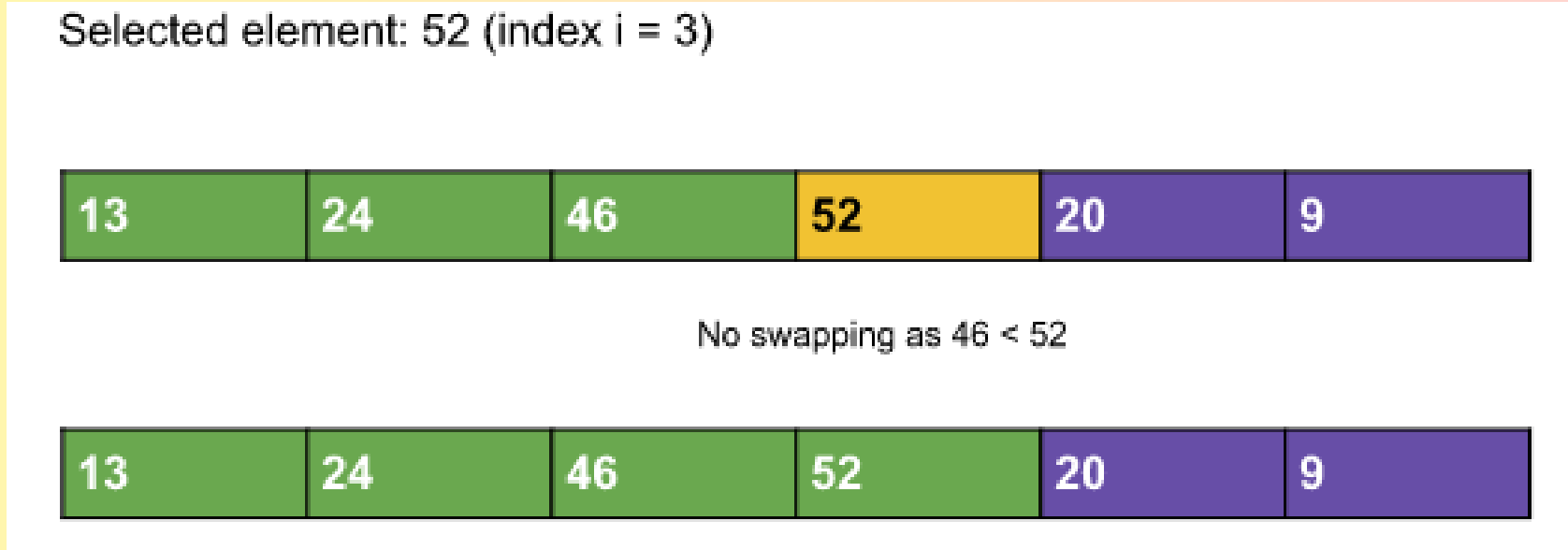
**Outer loop iteration 3(selected index i = 2):**

The selected element at index i=2 is 24. Now, we will try to move leftwards and put 24 in its correct position. Here, the correct position for 24 will be index 1. So, we will insert 24 in between 13 and 46. We will do it by swapping 24 and 46. Now, the subarray up to the third index is sorted.



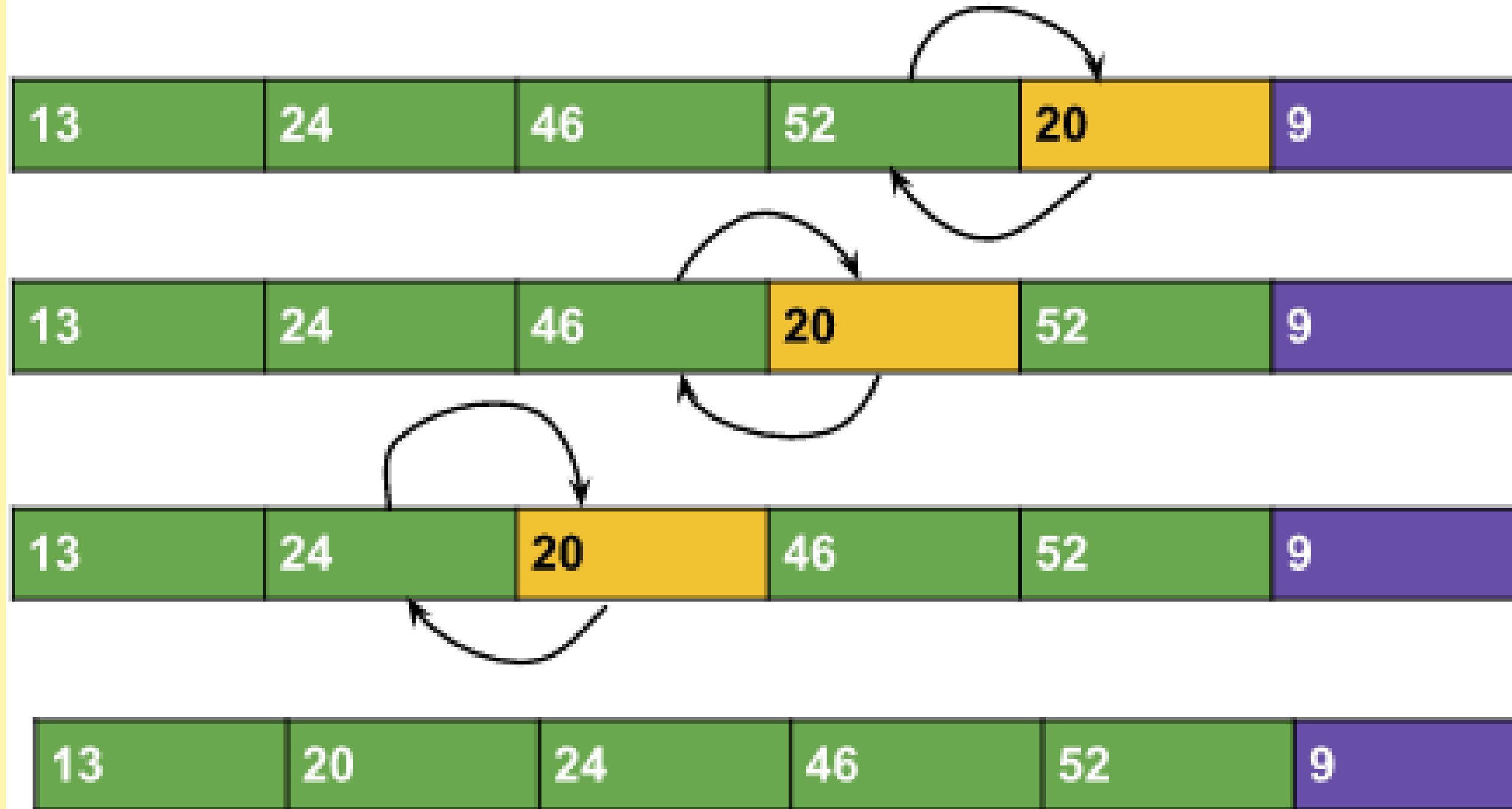
**Outer loop iteration 4(selected index i = 3):**

The selected element at index i=3 is 52. Now, we will try to move leftwards and put 52 in its correct position. Here, the correct position for 52 will be index 3. So, we need not swap anything. Now, the subarray up to the fourth index is sorted.



Selected element: 20 (index  $i = 4$ )

Swapping up to index 1



## Outer loop iteration

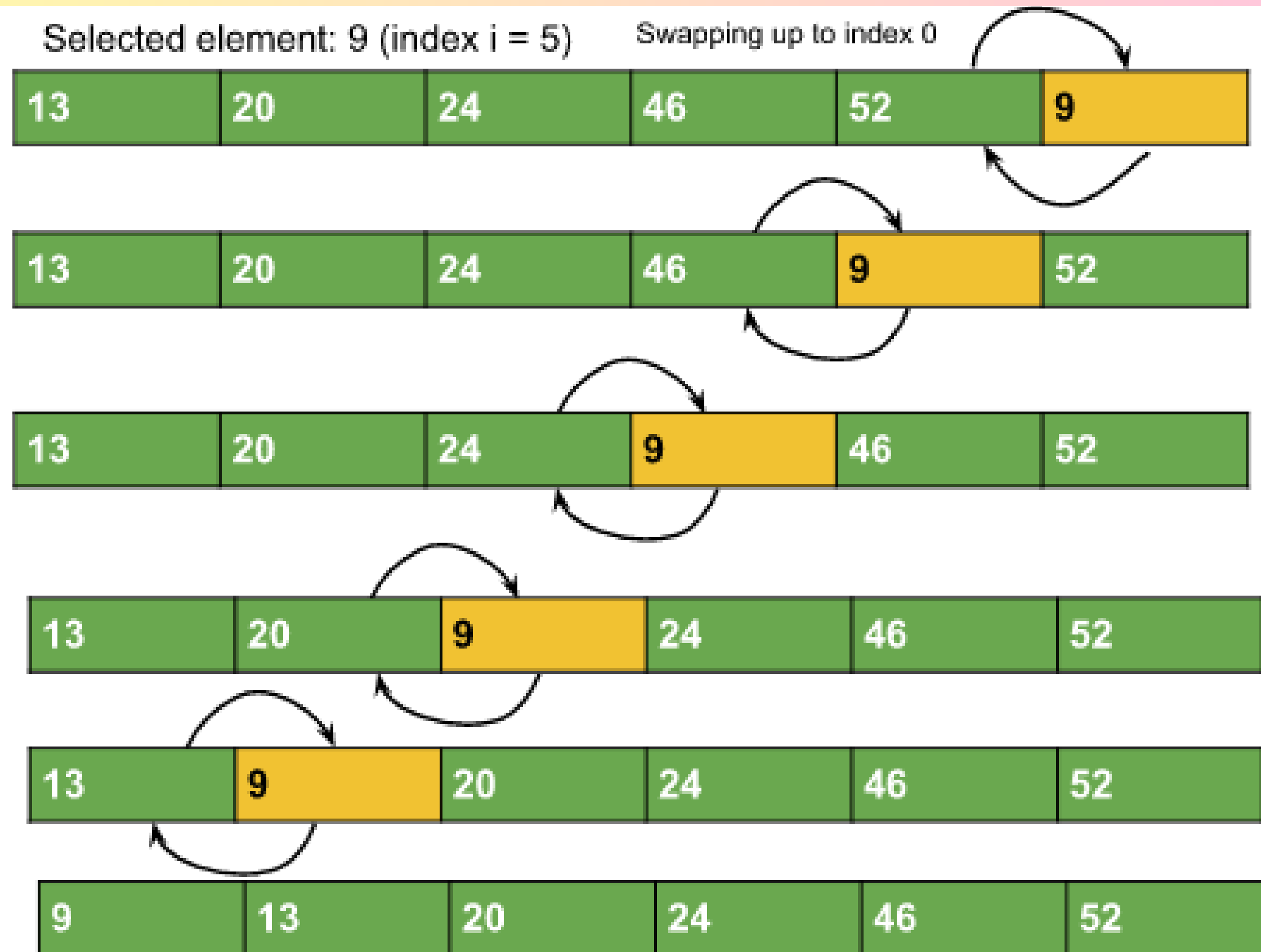
### 5(selected index $i = 4$ ):

The selected element at index  $i=4$  is 20.

Now, we will try to move leftwards and put 20 in its correct position.

Here, the correct position for 20 will be index 1. So, we need to swap adjacent elements until 20 reaches index 1.

Now, the subarray up to the fifth index is sorted.



## Outer loop iteration

### 6(selected index $i = 5$ ):

The selected element at index  $i=5$  is 9.

Now, we will try to move leftwards and put 9 in its correct position.

Here, the correct position for 9 will be index 0.

So, we need to swap adjacent elements until 9 reaches index 0.

Now, the whole array is sorted.

Code:

```
void insertionSort(int arr[], int n)
{
    //code here
    for(int i=1;i<=n-1;i++)
    {
        for(int j=i; j>=1;j--)
        {
            if(arr[j]<arr[j-1])
            {
                // swap(arr[j],arr[j-1]);
                int temp=arr[j];
                arr[j]=arr[j-1];
                arr[j-1]=temp;
            }
            else
            {
                break;
            }
        }
    }
}
```

## Time Complexity:

Best Case (Already Sorted):  $O(n)$

Worst Case (Reverse Sorted):  $O(n^2)$

Average Case (Random Order):  $O(n^2)$


## Space Complexity:

In-place:  $O(1)$

## Key Points:

Stable Sorting Algorithm

Best for small or nearly sorted data



# SHELL SORT

Definition

Algorithm

Dry Run

Time and Space Complexity

Code Implementation & Output





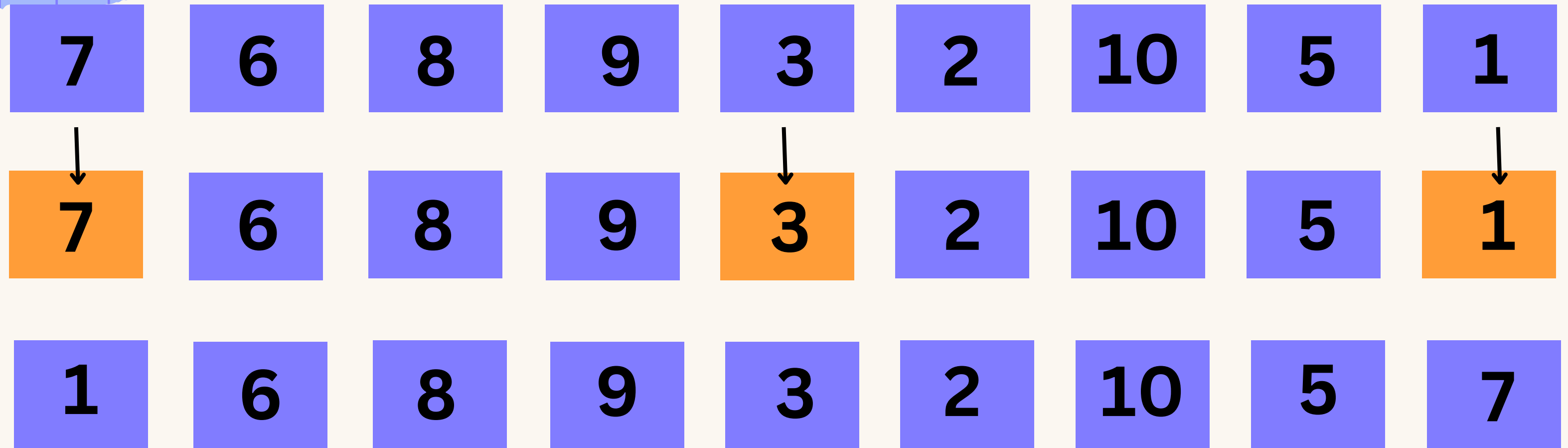
# Definition

- Developed in 1959 by Donald Shell
- Comparison-based sorting algorithm
- Similar to insertion sort, it is a variation of it
- Divides the array into smaller sub-parts, by picking elements at specific intervals known as **GAPS**.
- Further, it performs insertion sort on these smaller sub-parts

# Algorithm

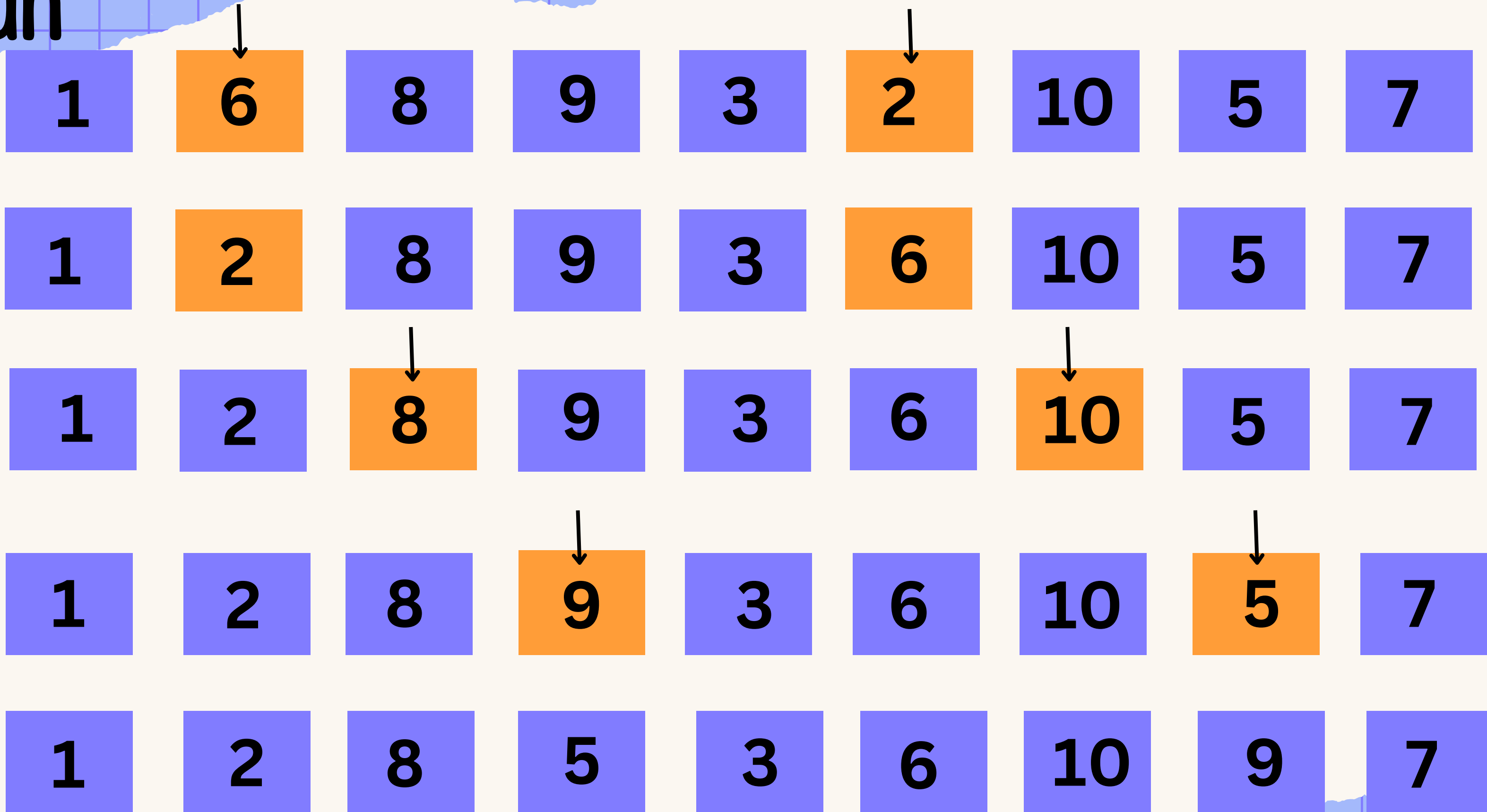
- Steps to solve using Shell Sort :
  1. Define gap size
  2. Divide into smaller sub-parts
  3. Sort sub-parts using insertion sort
    - a. Compare & swap elements
  4. Reduce gap with each comparison -  $n/2$  or 1 or etc
  5. Repeat till array is sorted

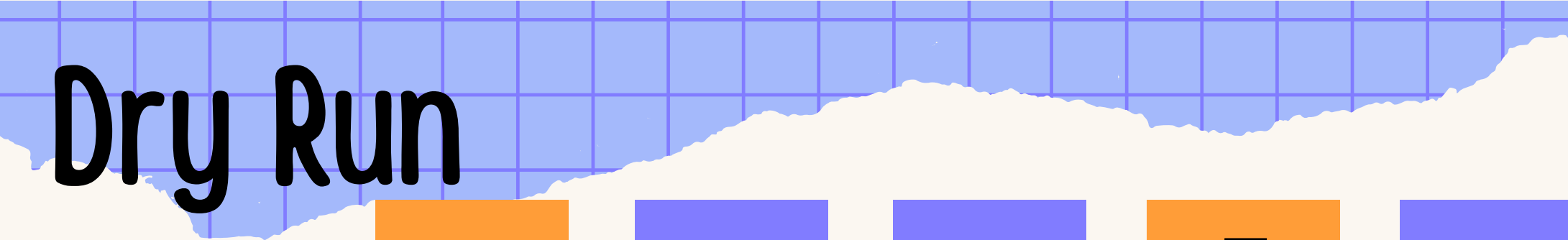
# Dry Run



- Gap Size : 3
- Swapping elements in smaller sub-parts known as SHELLS

# Dry Run





# Dry Run



1	2	8	5	3	6	10	9	7
---	---	---	---	---	---	----	---	---

Gap Size : 2

1	2	8	5	3	6	7	5	10
---	---	---	---	---	---	---	---	----

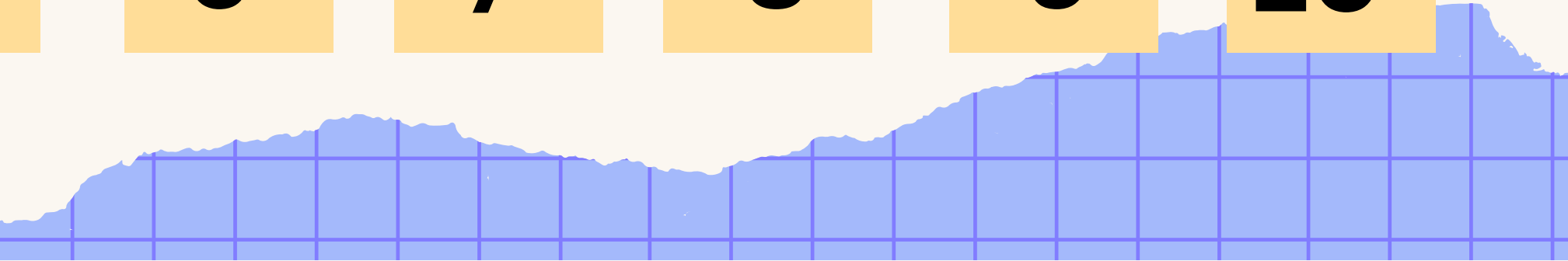
1	2	6	5	3	7	10	9	8
---	---	---	---	---	---	----	---	---

Gap Size : 1



1	2	6	5	3	6	10	9	8
---	---	---	---	---	---	----	---	---

1	2	3	5	6	7	8	9	10
---	---	---	---	---	---	---	---	----



# Time and Space Complexity

## Time Complexity

- varies on the gap size defined
- Average Case :  $O(n \log n)$
- Worst Case :  $O(n^3/2)$  or  $O(n \log^2 n)$
- Best Case :  $O(n \log n)$

## Space Complexity

- In-place sorting algorithm :  $O(1)$

# Code

```
#include <iostream>
using namespace std;

int shellSort(int arr[], int n)
{
    for (int gap = 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < n; i += 1)
        {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
    }
    return 0;
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

int main()
{
    int arr[] = {7, 6, 8, 9, 3, 2, 10, 5, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Array before sorting: \n";
    printArray(arr, n);

    shellSort(arr, n);

    cout << "\nArray after sorting: \n";
    printArray(arr, n);

    return 0;
}
```

```
Array before sorting:
7 6 8 9 3 2 10 5 1
Array after sorting:
1 2 3 5 6 7 8 9 10
```



**THANK  
YOU!**

