

```
for (x, c, l) in zip(feature_pyramid, self.class_pred, self.loc_pred):  
    class_preds.append(c(x).permute(0, 2, 3, 1))  
    loc_preds.append(l(x).permute(0, 2, 3, 1))
```

# SEARCHING AND SORTING ARRAYS

CCDSALG T2 AY 2020-2021

# ARRAYS

3	19	7	1	21	17
0	1	2	3	4	5

- A linear data structure mainly used to store similar data.
- Data are stored sequentially within the array.
- Each element is referenced by an index or subscripts.

# SEARCHING ARRAYS

3	19	7	1	21	17
0	1	2	3	4	5

- **Problem:** We have an array of numbers. We want to identify the index of the array containing one specific number.
- **Considerations:**
  - How fast can we do it?
  - If the array has certain properties, can we do it faster?

# SEARCHING ARRAYS

3	19	7	1	21	17
0	1	2	3	4	5

## Easy solution:

1. Iterate over the array
2. At each index, check if the content is equal to the value
  - If equal, then return the index
  - If not equal, keep going
  - If we reach the last element, the value is not there

# LINEAR SEARCH

**Linear search**

Array     

6	3	0	5	1	2	8	-1	4
---	---	---	---	---	---	---	----	---

Element to search: 8

## Easy solution:

- Worst case scenario, the value is stored in the last index of the array. Thus, we will iterate over the whole array
- This algorithm is called **linear search**.

# LINEAR SEARCH

```
#include <stdio.h>
int main()
{
    int array[100], search, c, n;

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

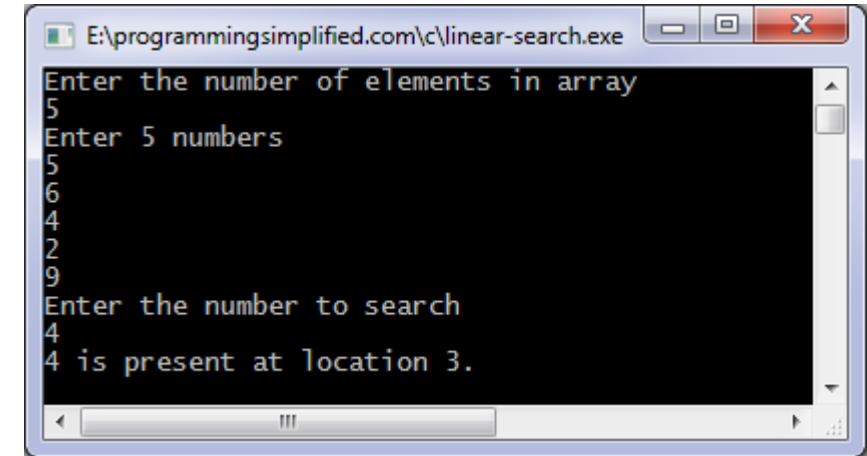
    printf("Enter %d integer(s)\n", n);

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

    printf("Enter a number to search\n");
    scanf("%d", &search);

    for (c = 0; c < n; c++)
    {
        if (array[c] == search)    /* If required element is found */
        {
            printf("%d is present at location %d.\n", search, c+1);
            break;
        }
    }
    if (c == n)
        printf("%d isn't present in the array.\n", search);

    return 0;
}
```



```
E:\programmingsimplified.com\c\linear-search.exe
Enter the number of elements in array
5
Enter 5 numbers
5
6
4
2
9
Enter the number to search
4
4 is present at location 3.
```

# ARRAYS

What if we are given a sorted array?

1	3	7	17	19	21
0	1	2	3	4	5

# BINARY SEARCH

## Alternative approach

1. Array  $a$ ,  $n$  elements in the array,  $l = 0$ ,  $h = n - 1$ ,  $m = \frac{l+h+1}{2}$ ,  $a_m$  is the value in the middle of the array,  $x$  is the value.
2. Compare  $a_m$  to  $x$ .
  - If  $a_m = x$ , then return  $m$ .
  - If  $a_m > x$ , since the array is sorted, then  $x$  must be in the first half of the array. Set  $h = m - 1$  and  $m = \frac{l+h+1}{2}$ .
  - Otherwise ( $a_m < x$ ),  $x$  must be in the second half of the array. Set  $l = m + 1$ ,  $m = \frac{l+h+1}{2}$ .
3. Keep going until either  $x$  is found or  $l > h$ .

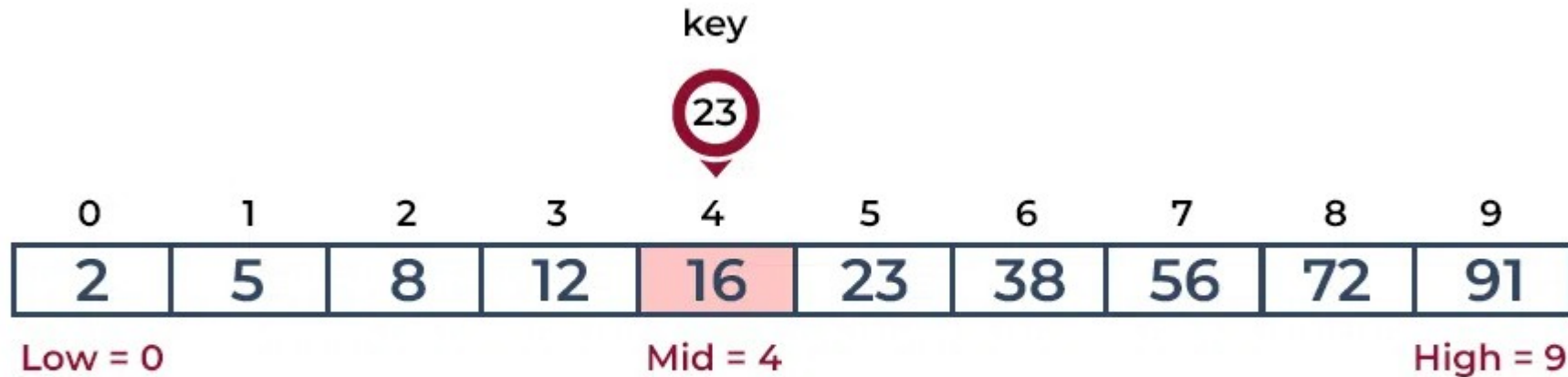


# BINARY SEARCH

## **Alternative approach**

1. Compare the middle element of the search space with the key.
2. If the key is found at middle element, the process is terminated.
3. If the key is not found at middle element, choose which half will be used as the next search space.
  1. If the key is smaller than the middle element, then the left side is used for next search.
  2. If the key is larger than the middle element, then the right side is used for next search.
4. This process is continued until the key is found or the total search space is exhausted.

# BINARY SEARCH



Binary search

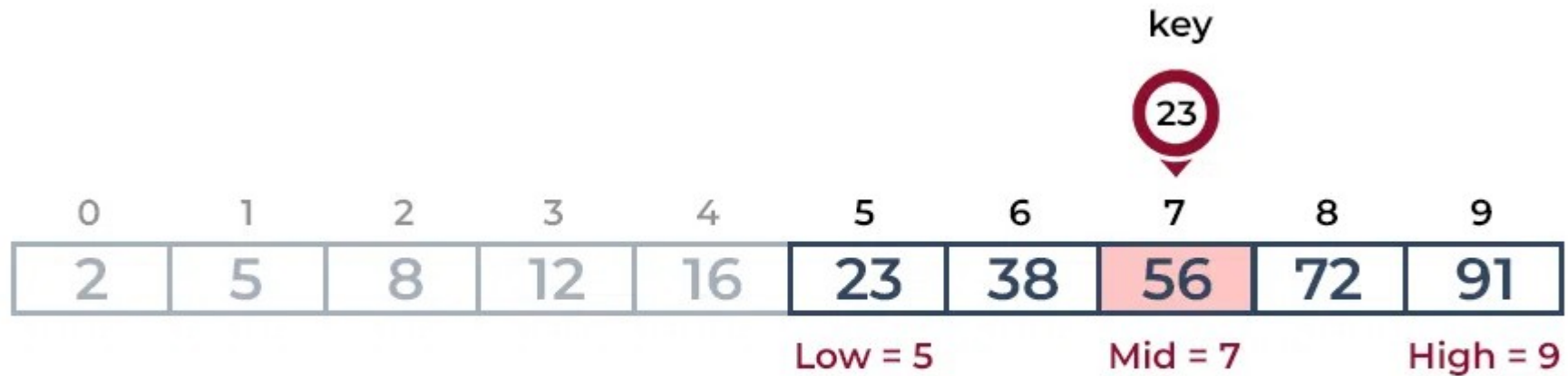


Consider an array  $\text{arr}[] = \{2, 5, 8, 12, 16, 23, 38, 56, 72, 91\}$ , and the target = 23.

First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.

# BINARY SEARCH

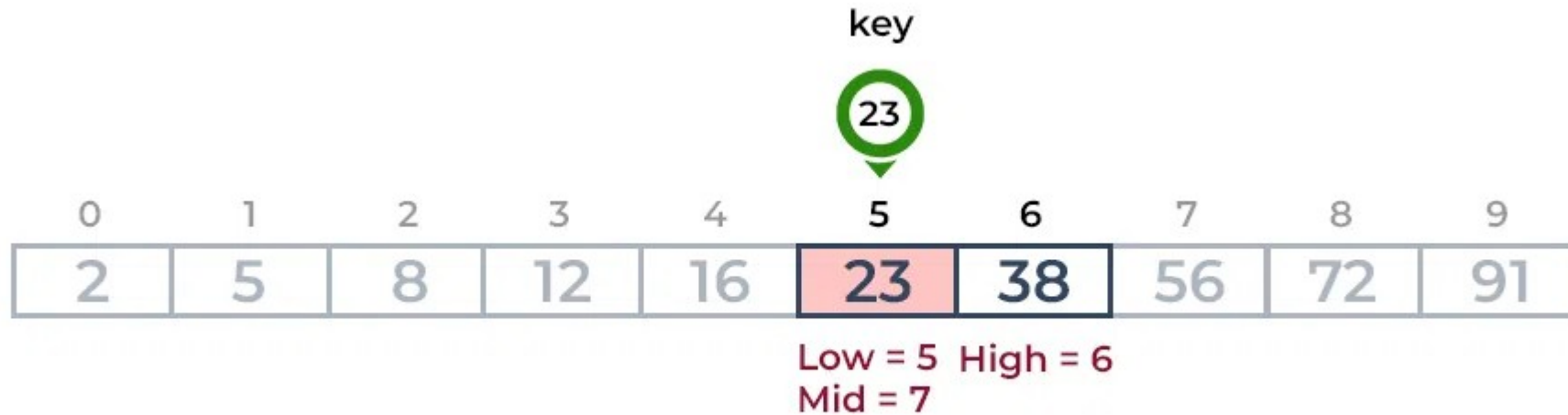


Binary search



Key is less than the current mid 56. The search space moves to the left.

# BINARY SEARCH



Binary search



Second Step: If the key matches the value of the mid element, the element is found and stop search.

# BINARY SEARCH

## Iterative

```
// C program to implement iterative Binary Search
#include <stdio.h>
```

```
// An iterative binary search function.
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

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

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

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

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

```
// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present"
                           " in array")
                  : printf("Element is present at "
                           "index %d", result);

    return 0;
}
```

# BINARY SEARCH (

## Recursive

```
// C program to implement recursive Binary Search
#include <stdio.h>
```

```
// A recursive binary search function. It returns
// location of x in given array arr[l..r] is
// present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 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, l, mid - 1,
x);

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

```
        // We reach here when element
is not
        // present in array
        return -1;
    }

// Driver code
int main()
{
    int arr[] = { 2, 3, 4, 10, 40
};
    int n = sizeof(arr) /
sizeof(arr[0]);
    int x = 10;
    int result =
binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? printf("Element is not
present in array")
        : printf("Element is
present at index %d", result);
    return 0;
}
```

# BINARY SEARCH

Binary search

steps: 0



Sequential search

steps: 0



[www.penjee.com](http://www.penjee.com)

# BINARY SEARCH

## Binary Search

- Assume the array contains  $n$  elements
- After the first test, the algorithm eliminated  $\frac{n}{2}$  elements of the array
- After the second test, the algorithm eliminated  $\frac{1}{2}$  of an array with  $\frac{n}{2}$  elements, ending up with  $\frac{n}{4}$  elements.
- This process will continue until the algorithm finds the value or if the algorithm finds that the value is not in the array



# SEARCHING ALGORITHMS

- Which technique is more efficient?
- What is the maximum possible number of comparisons needed to perform a linear search on an array of size  $n$ ?  
What is the minimum number of comparisons?
- What about using binary search?

# SORTING ALGORITHMS

## Example

**Input:** a sequence of  $n$  numbers  $\{a_1, a_2, \dots, a_n\}$

**Output:** a permutation (reordering)  $\{a'_1, a'_2, \dots, a'_n\}$   
such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

**Solution:** Sorting algorithms

# SORTING ALGORITHMS

		 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random									
 Nearly Sorted									
 Reversed									
 Few Unique									

[Image credits](#)

# SORTING ALGORITHMS

## Properties

- **In-place** – The algorithm uses no additional array storage and hence, it is possible to sort very large lists without the need to allocate additional working storage.

# BUBBLE SORT

- Oldest, simplest, and slowest sort in use.
- Process: comparing each item in the list with the item next to it and swapping them if required. Repeat the process until it makes a pass all the way through the list without swapping any items – causes larger values to “bubble” to the end of the list, while smaller values “sink” towards the beginning of the list.
- Average and worst case:  $O(n^2)$
- In-place algorithm

# BUBBLE SORT

6 5 3 1 8 7 2 4

# BUBBLE SORT

```
void bubble_sort(int array[], int size) {  
    int i, j;  
    // loop to access each array element  
    for(i = 0; i < size; i++)  
        // loop to compare array elements  
        for(j = size - 1; j >= i + 1; j--)  
            // compare two adjacent elements  
            if (array[j] < array[j - 1])  
                // swapping occurs if elements are not in the intended order  
                swap(array[j], array[j - 1]);  
}
```

# BUBBLE SORT DEMO

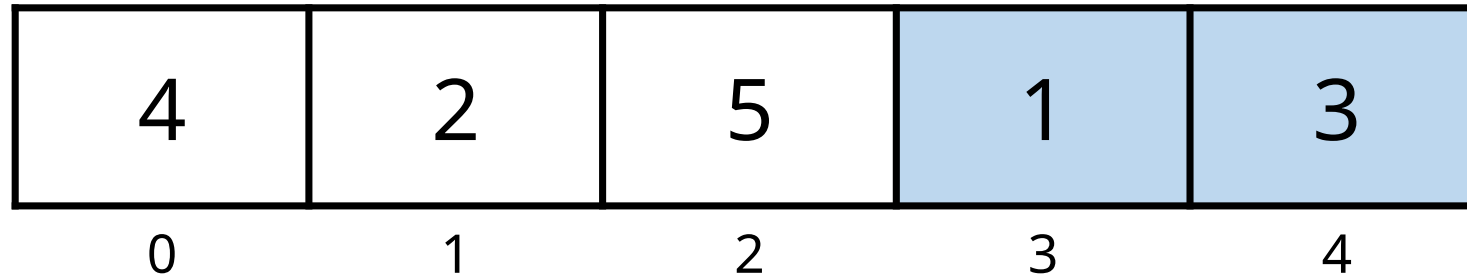
Sort the array below using bubble sort.

4	2	5	1	3
0	1	2	3	4



# BUBBLE SORT DEMO

Sort the array below using bubble sort.

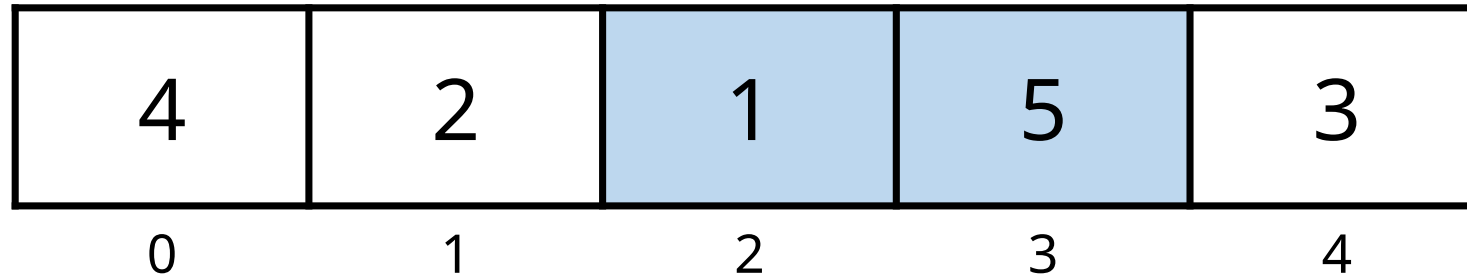


$$i = 0$$

$$j = 4$$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



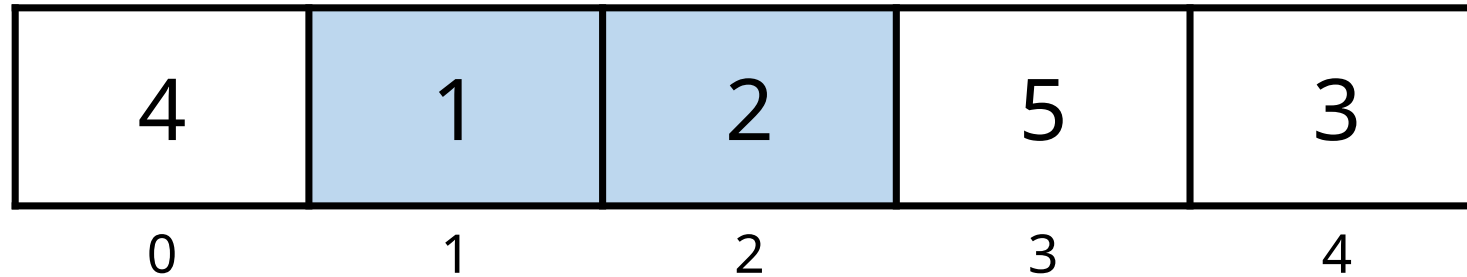
SWAP 1 and 5

$i = 0$

$j = 3$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



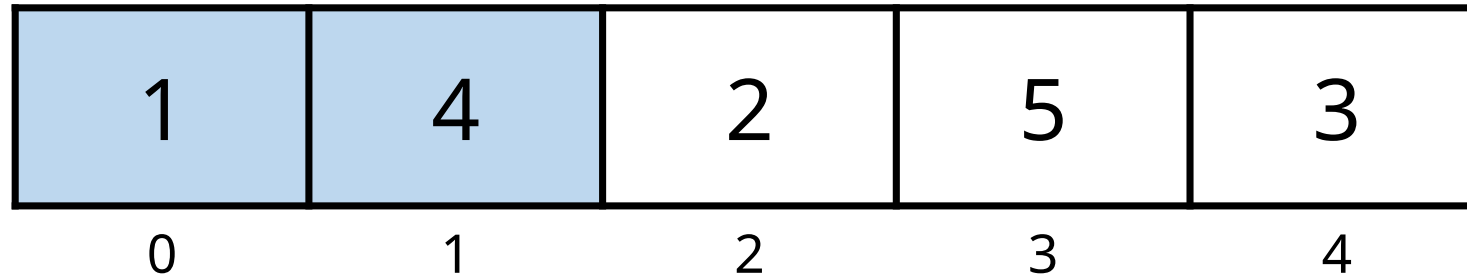
SWAP 1 and 2

$i = 0$

$j = 2$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



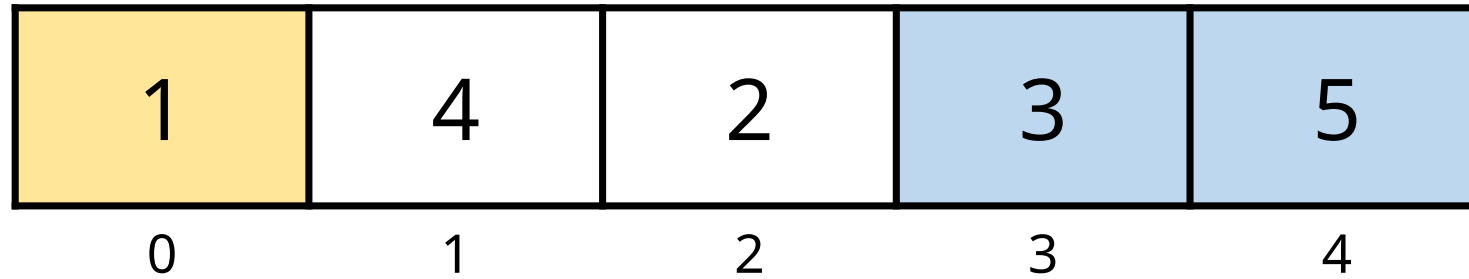
SWAP 1 and 4

$i = 0$

$j = 1$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



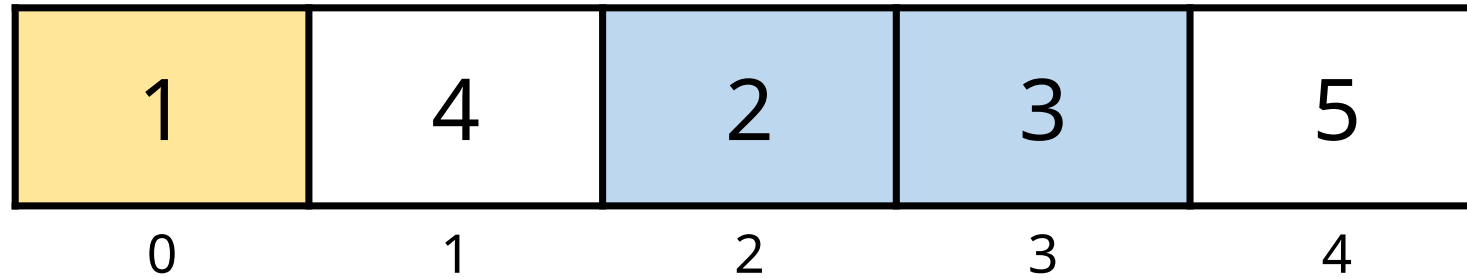
SWAP 3 and 5

$i = 1$

$j = 4$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.

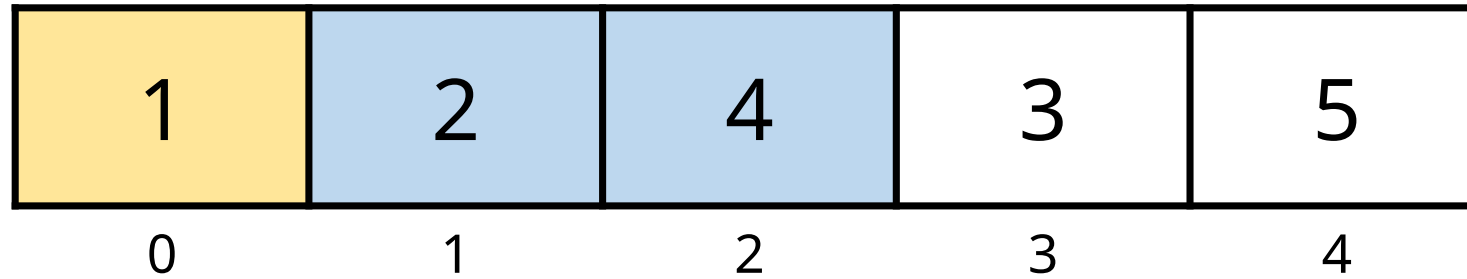


$$i = 1$$

$$j = 3$$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



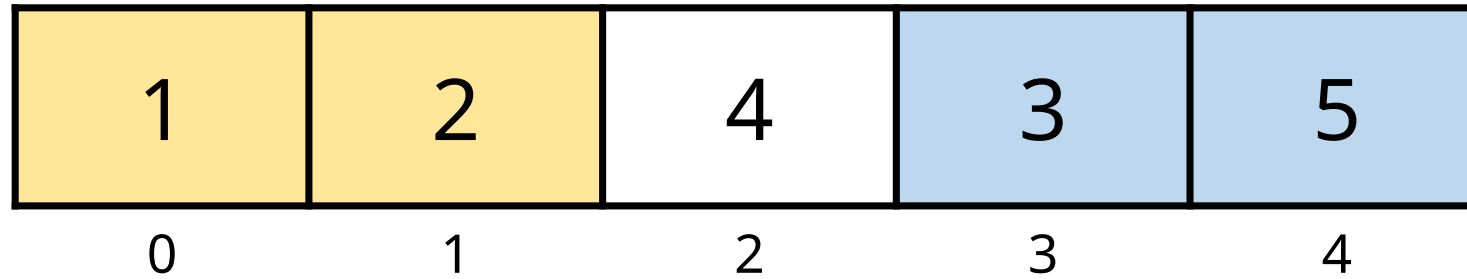
SWAP 2 and 4

$i = 1$

$j = 2$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



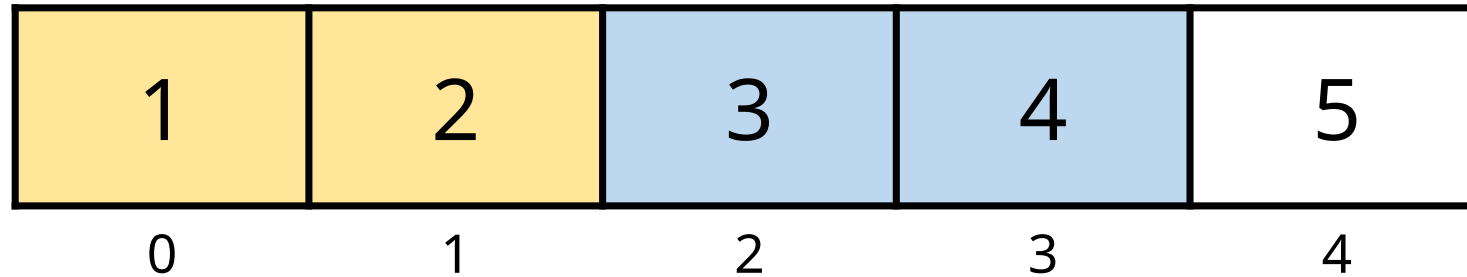
$$i = 2$$

$$j = 4$$



# BUBBLE SORT DEMO

Sort the array below using bubble sort.



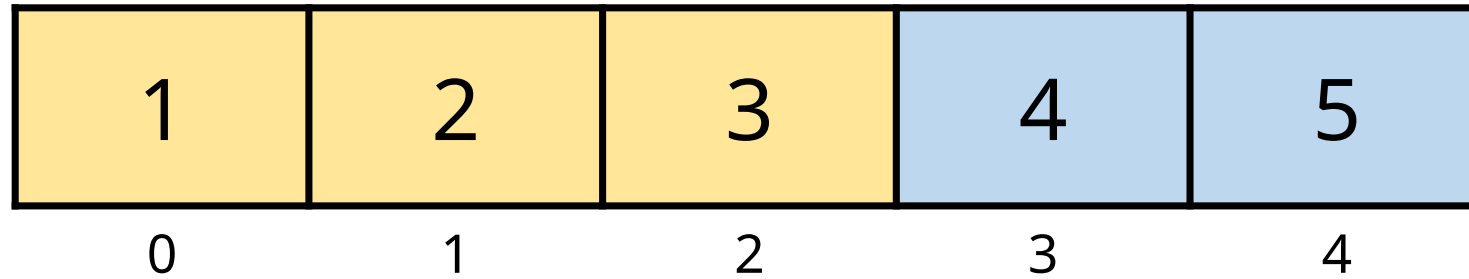
SWAP 3 and 4

$i = 2$

$j = 3$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.

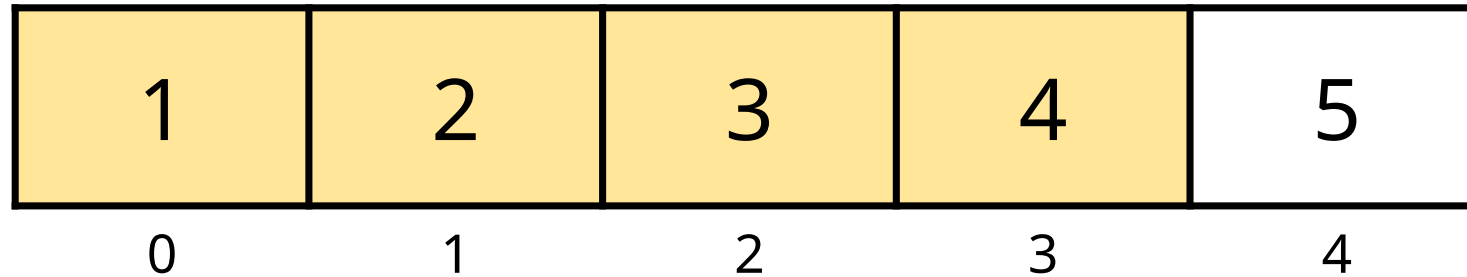


$$i = 3$$

$$j = 4$$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.



$$i = 4$$

$$j = 4$$

# BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	2	3	4	5
0	1	2	3	4

SORTED ARRAY

# INSERTION SORT

- Inserts each item into its proper place in the final sorted list.
- Like how people usually sort a hand of playing cards
- Average and worst case:  $O(n^2)$
- In-place algorithm

# INSERTION SORT

6 5 3 1 8 7 2 4

# INSERTION SORT

```
void insertion_sort(int array[], int size) {  
    int i, j, element;  
    for(i = 1; i < size; i++) {  
        element = array[i];  
        j = i;  
        while((j > 0) && (array[j - 1] > element)) {  
            array[j] = array[j - 1];  
            j--;  
        }  
        array[j] = element;  
    }  
}
```

# INSERTION SORT DEMO

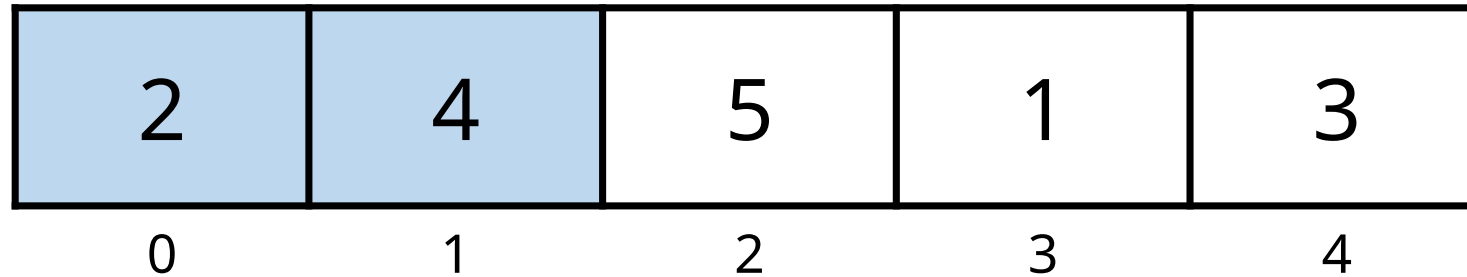
Sort the array below using insertion sort.

4	2	5	1	3
0	1	2	3	4



# INSERTION SORT DEMO

Sort the array below using insertion sort.

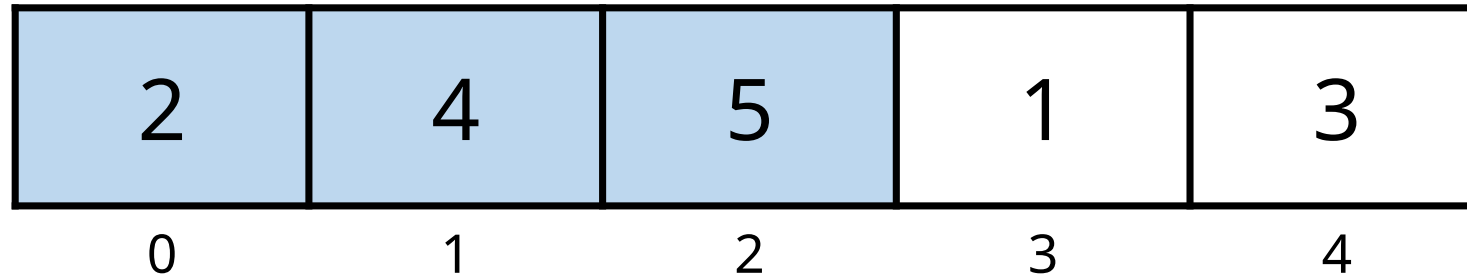


INSERT 2 at index 0

$i = 1$

# INSERTION SORT DEMO

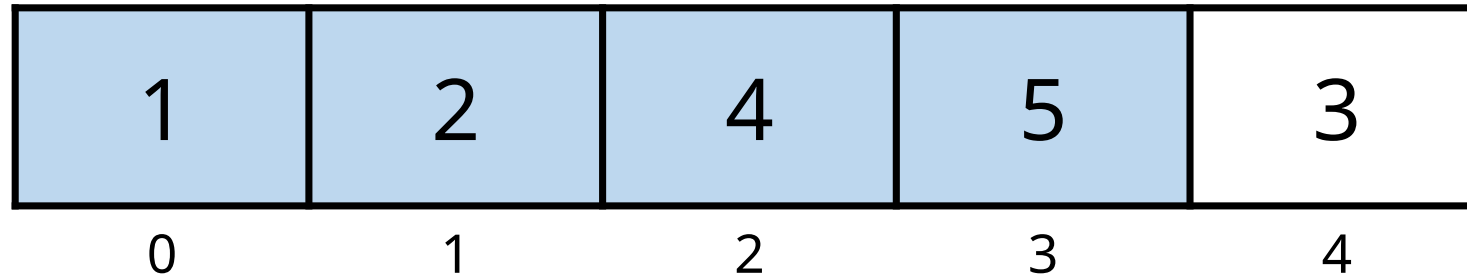
Sort the array below using insertion sort.



$i = 2$

# INSERTION SORT DEMO

Sort the array below using insertion sort.

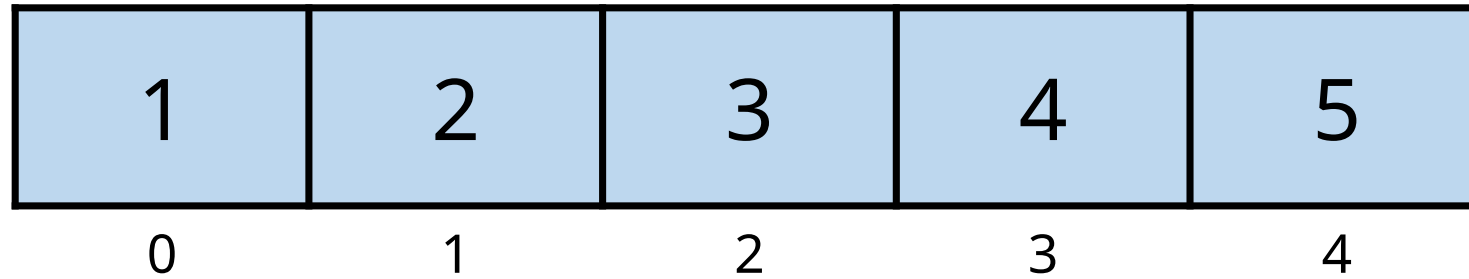


INSERT 1 at index 0

$i = 3$

# INSERTION SORT DEMO

Sort the array below using insertion sort.



INSERT 3 at index 2

$i = 4$

# INSERTION SORT DEMO

Sort the array below using insertion sort.

1	2	3	4	5
0	1	2	3	4

SORTED ARRAY

# SELECTION SORT

- For each position in the array, place the smallest element from the unsorted portion of the array
- Average and worst case:  $O(n^2)$
- In-place algorithm

# SELECTION SORT

5 3 4 1 2

Selection Sort

# SELECTION SORT

```
void selection_sort(int array[], int size) {  
    int i, j, min;  
    for(i = 0; i < size - 1; i++) {  
        min = i;  
        for(j = i + 1; j < size; j++)  
            if (array[j] < array[min])  
                min = j;  
        if (min != i)  
            swap(array[i], array[min]);  
    }  
}
```



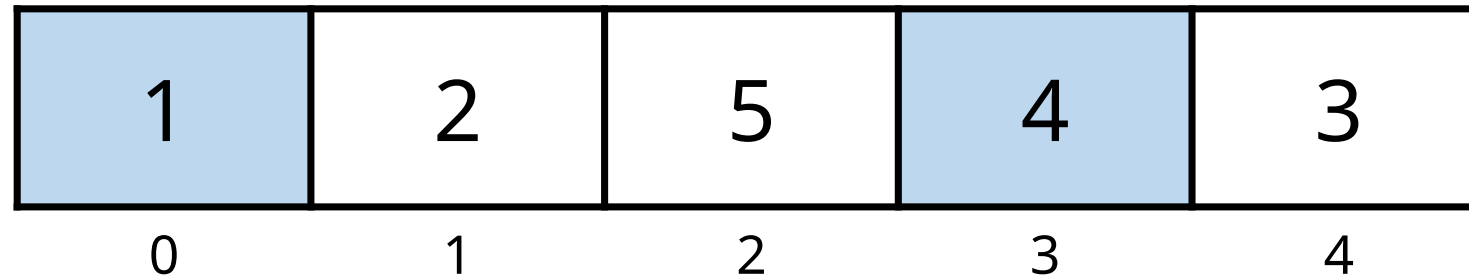
# SELECTION SORT DEMO

Sort the array below using selection sort.

4	2	5	1	3
0	1	2	3	4

# SELECTION SORT DEMO

Sort the array below using selection sort.

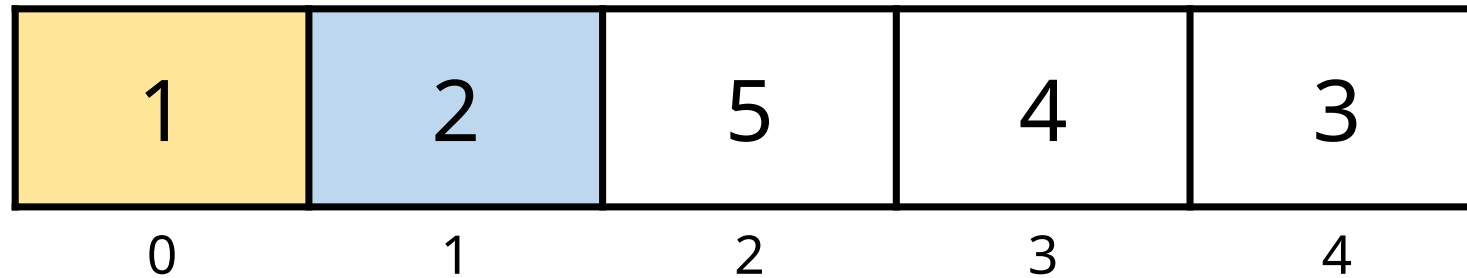


SELECT 1 from index 3 and swap with element at index 0

$i = 0$

# SELECTION SORT DEMO

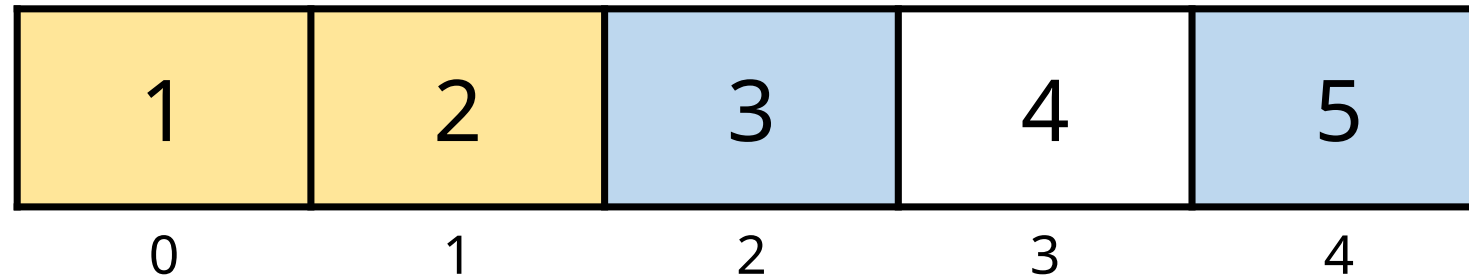
Sort the array below using selection sort.



$i = 1$

# SELECTION SORT DEMO

Sort the array below using selection sort.

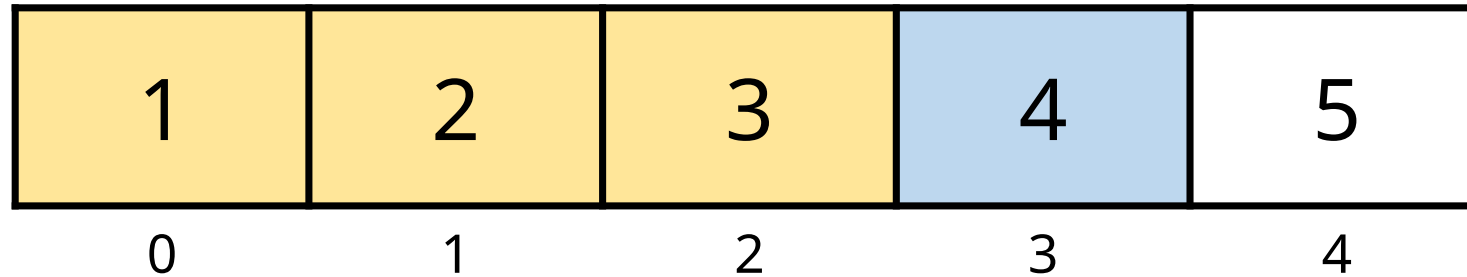


SELECT 3 from index 4 and swap with element at index 2

$$i = 2$$

# SELECTION SORT DEMO

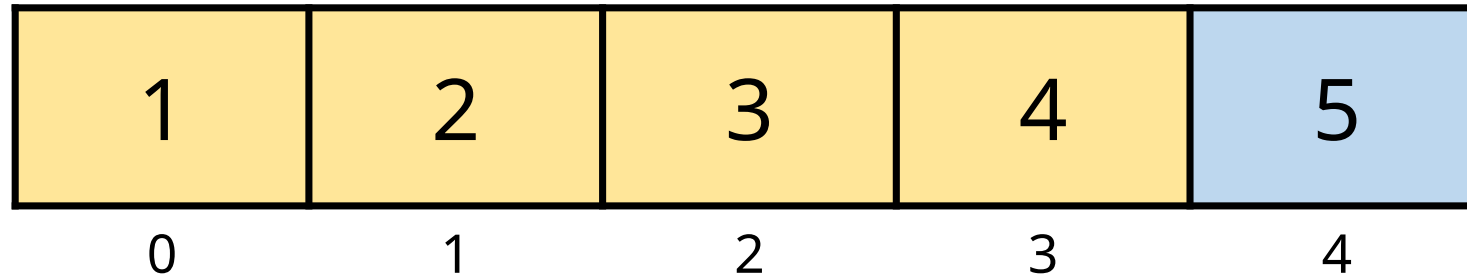
Sort the array below using selection sort.



$$i = 3$$

# SELECTION SORT DEMO

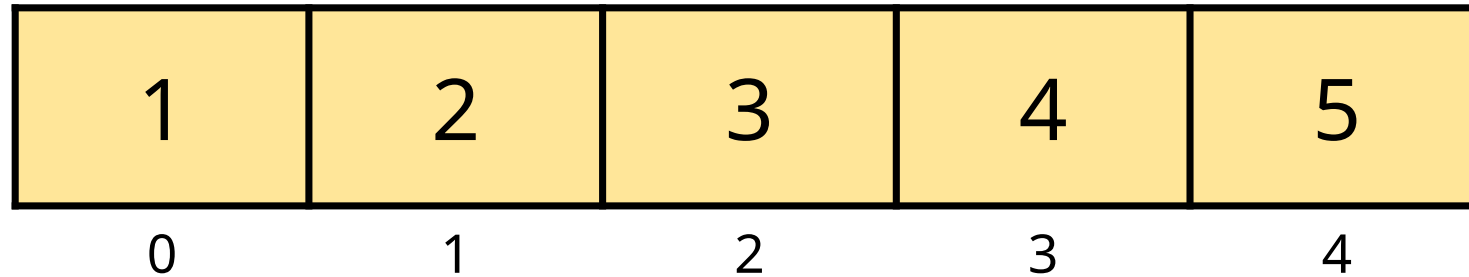
Sort the array below using selection sort.



$$i = 4$$

# SELECTION SORT DEMO

Sort the array below using selection sort.



$$i = 4$$

# SELECTION SORT DEMO

Sort the array below using selection sort.

1	2	3	4	5
0	1	2	3	4

SORTED ARRAY



# MERGE SORT

Based on the **divide-and-conquer** paradigm

## Divide Step

- If given array  $A$  has zero or one element, then return that it is sorted
- Otherwise, divide  $A$  into two arrays,  $A_1$  and  $A_2$ , each containing about half of the elements of  $A$ .

# MERGE SORT

Based on the **divide-and-conquer** paradigm

## Recursion Step

- Recursively sort array A1 and array A2

## Conquer Step

- Combine the elements back in A by merging sorted arrays A1 and A2 into sorted sequence

# MERGE SORT

Average and worst case:  $n \log_2 n$

Not in-place

# MERGE SORT

```
void merge_sort(int array[], int l, int r) {  
    if(l < r) {  
        int m = (l + r) / 2;  
        merge_sort(array, l, m);  
        merge_sort(array, m + 1, r);  
        merge(array, l, m, r);  
    }  
}
```

# MERGE SORT

```
void merge(int array[], int l, int m, int r) {  
    int i = 0, j = 0, k = l;  
    int n1 = m - l + 1;  
    int n2 = r - m;  
    int L[n1], R[n2];  
    /* Copy data to temp arrays L[] and R[] */  
    for (i = 0; i < n1; i++)  
        L[i] = arr[l + i];  
    for (j = 0; j < n2; j++)  
        R[j] = arr[m + 1 + j];  
    ...  
}
```

# MERGE SORT

```
while (i < n1 && j < n2) {  
    if (L[i] <= R[j]) {  
        arr[k] = L[i];  
        i++;  
    }  
    else {  
        arr[k] = R[j];  
        j++;  
    }  
    k++;  
}  
...
```

# MERGE SORT

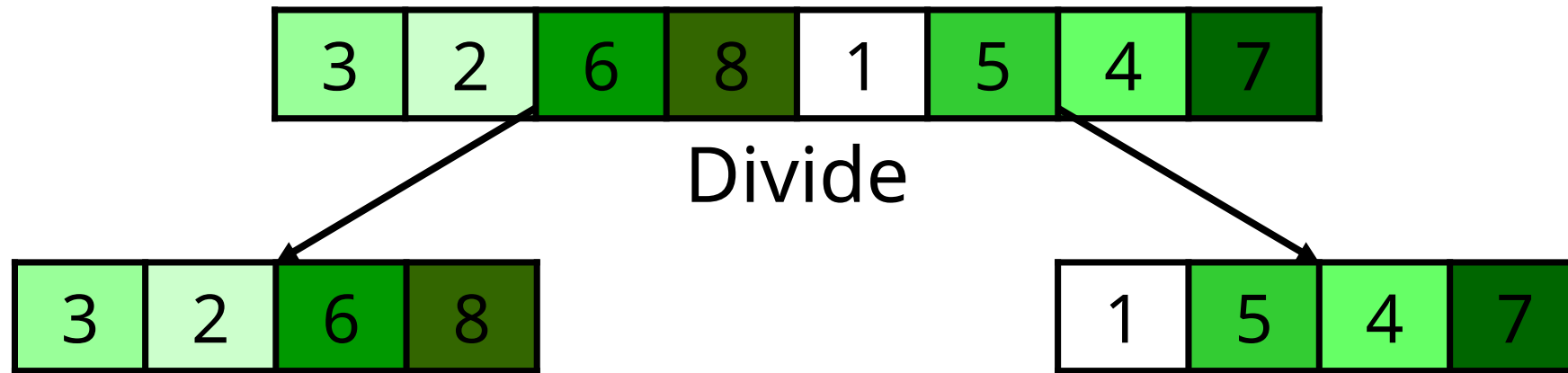
```
/* Copy the remaining elements of temporary arrays */  
while (i < n1) {  
    arr[k] = L[i];  
    i++;  
    k++;  
}  
while (j < n2) {  
    arr[k] = R[j];  
    j++;  
    k++;  
}  
}
```

# MERGE SORT DEMO

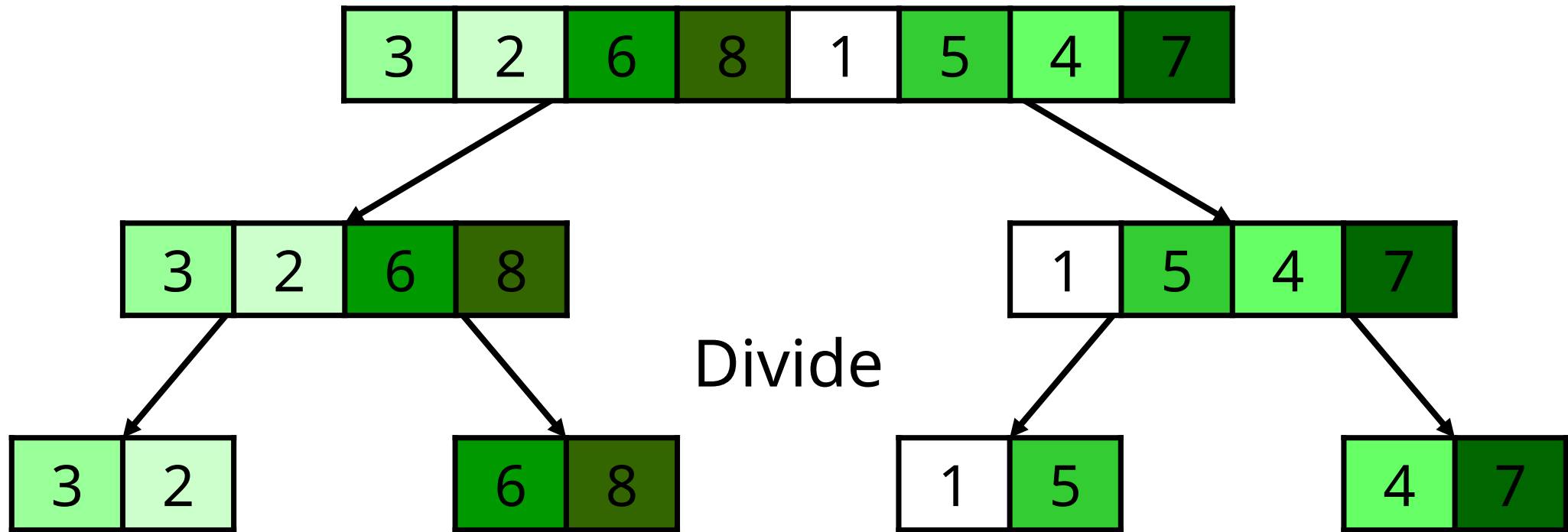




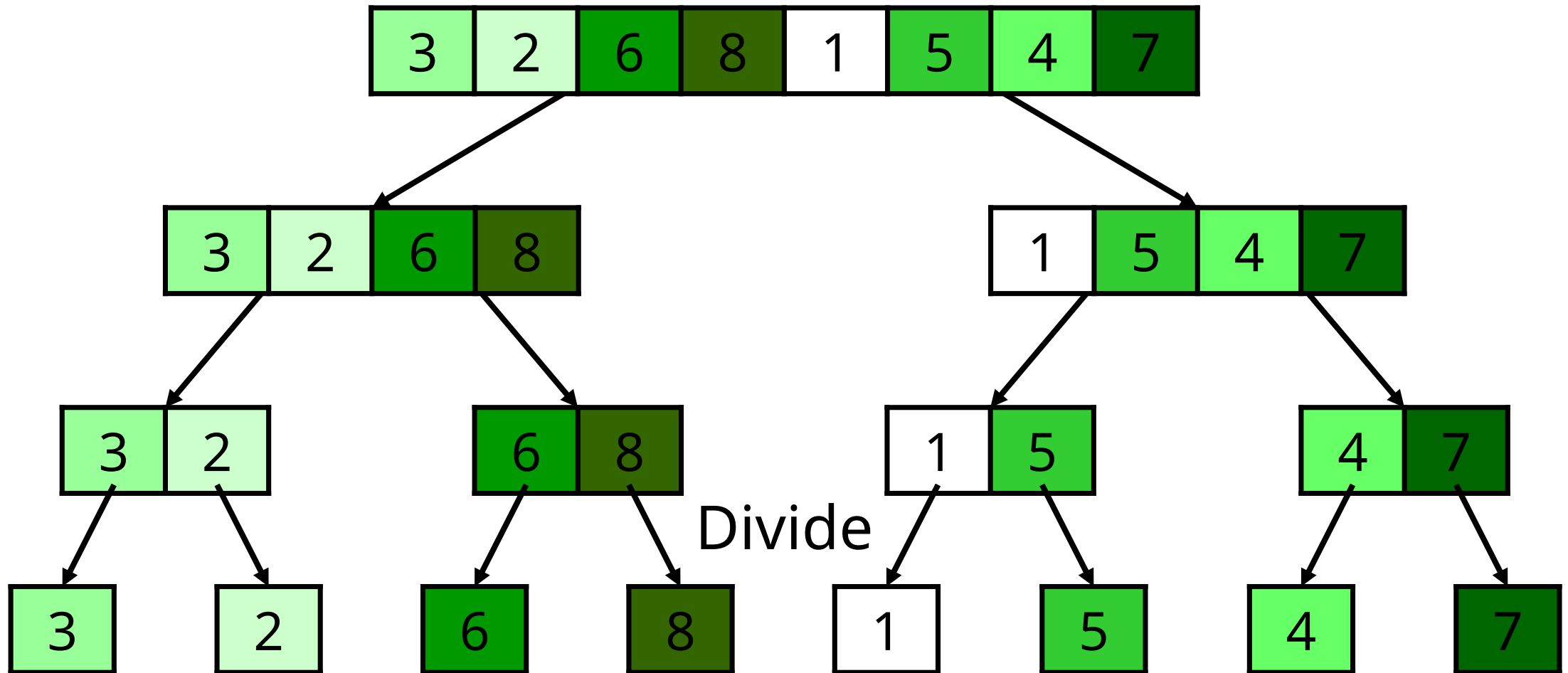
# MERGE SORT DEMO



# MERGE SORT DEMO



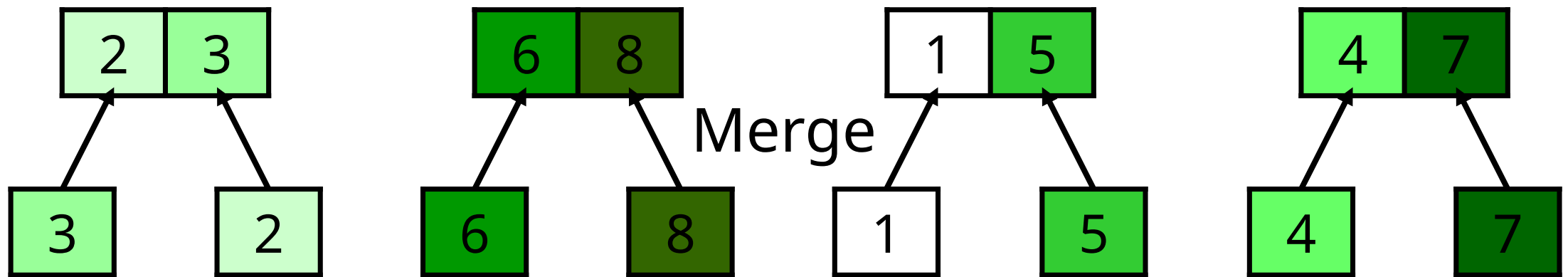
# MERGE SORT DEMO



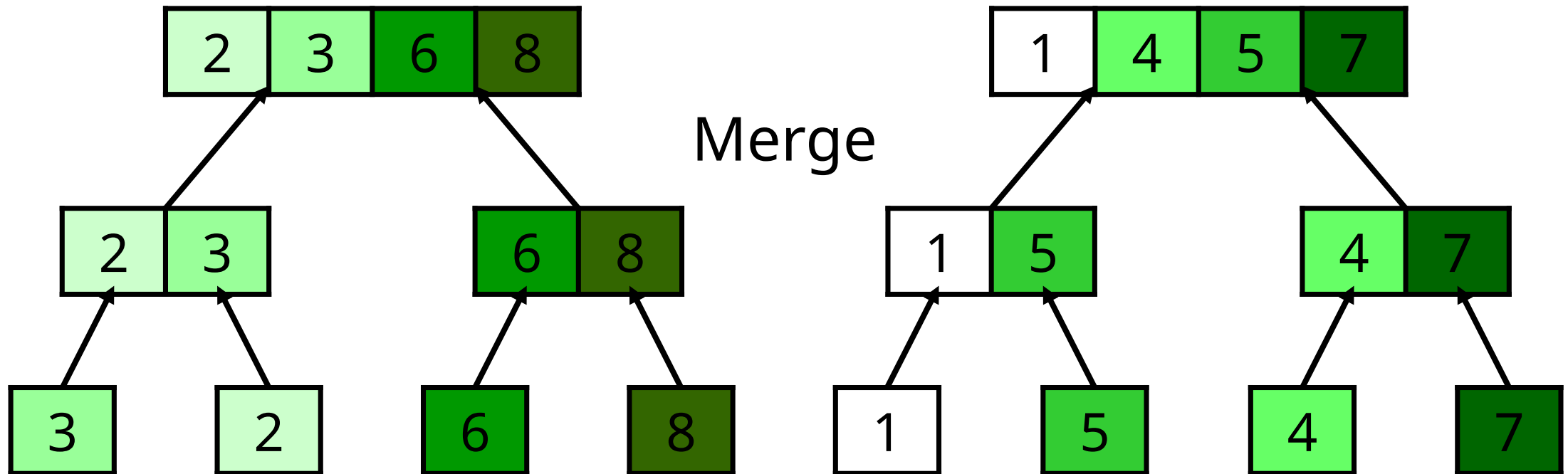
# MERGE SORT DEMO



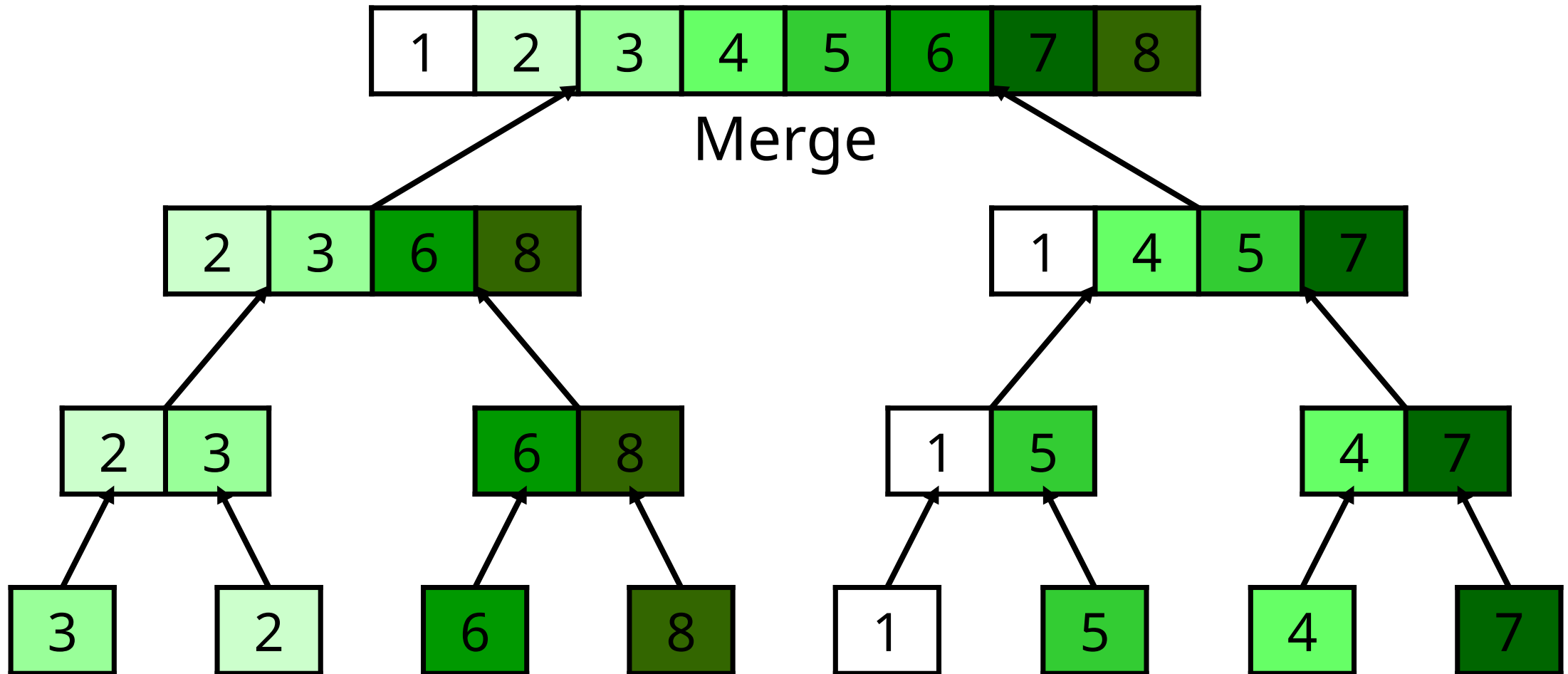
# MERGE SORT DEMO



# MERGE SORT DEMO



# MERGE SORT DEMO



## YT links for Sorting Algorithms in C

### Merge Sort

<https://youtu.be/VZCJ4qDsd2Q>

<https://youtu.be/9vLuZ2SSLII>

### Bubble Sort

<https://youtu.be/YqzNgaFQEh8>

### Selection Sort

<https://youtu.be/YepJ7fDmyjI>

### Insertion Sort

<https://youtu.be/Tz7vBodZqo8>



```
for (x, c, l) in zip(feature_pyramid, self.class_pred, self.loc_pred):  
    class_preds.append(c(x).permute(0, 2, 3, 1))  
    loc_preds.append(l(x).permute(0, 2, 3, 1))
```

# SEARCHING AND SORTING ARRAYS

CCDSALG T2 AY 2020-2021