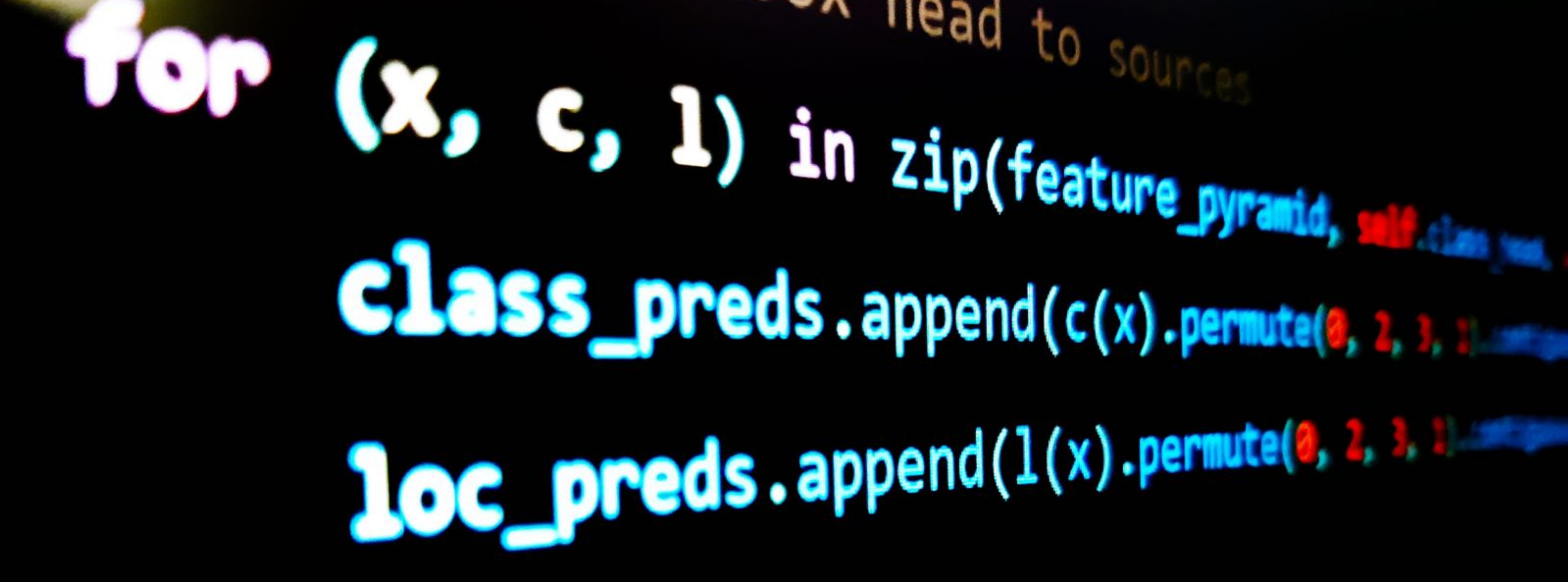


SEARCHING AND SORTING ARRAYS

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

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

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

```
int LinearSearch(int A[], int n, int x) {
    int i = 0;
    int index = -1;
    int found = FALSE;
    while (i < n && !found) {
        if(A[i] == x) {
            index = i;
            found = TRUE;
        }
        i++;
    }
    return index;
}
```

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

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

BINARY SEARCH

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

BINARY SEARCH

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

BINARY SEARCH

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1	3	7	17	19	21	23	30	37	40	55	68	78	79	83	88	92	94
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

BINARY SEARCH

```
int BinarySearch(int A[], int low, int high, int x) {
    int mid;
    int found = FALSE;
    while(low <= high && !found) {
        mid = (low + high) / 2;
        if(A[mid] == x)
            found = TRUE;
        else if(x < A[mid])
            high = mid - 1
        else if(x > A[mid])
            low = mid + 1
    }
    if(found)
        return mid;
    else
        return -1;
}
```

BINARY SEARCH

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

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 $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Solution: Sorting algorithms

SORTING ALGORITHMS

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;
    for(i = 0; i < size; i++)
        for(j = size – 1; j >= i + 1; j--)
            if (array[j] < array[j – 1])
                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.

4	2	5	1	3
0	1	2	3	4

$i = 0$

$j = 4$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

4	2	1	5	3
0	1	2	3	4

SWAP 1 and 5

$i = 0$
$j = 3$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

4	1	2	5	3
0	1	2	3	4

SWAP 1 and 2

$i = 0$
$j = 2$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	4	2	5	3
0	1	2	3	4

SWAP 1 and 4

$i = 0$
$j = 1$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	4	2	3	5
0	1	2	3	4

SWAP 3 and 5

$i = 1$
$j = 4$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	4	2	3	5
0	1	2	3	4

$i = 1$

$j = 3$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	2	4	3	5
0	1	2	3	4

SWAP 2 and 4

$i = 1$

$j = 2$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	2	4	3	5
0	1	2	3	4

$i = 2$

$j = 4$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	2	3	4	5
0	1	2	3	4

SWAP 3 and 4

$i = 2$

$j = 3$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	2	3	4	5
0	1	2	3	4

$i = 3$

$j = 4$

BUBBLE SORT DEMO

Sort the array below using bubble sort.

1	2	3	4	5
0	1	2	3	4

$i = 4$

$j = 4$

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.

2	4	5	1	3
0	1	2	3	4

INSERT $\mathbf{2}$ at index $\mathbf{0}$

$i = 1$

INSERTION SORT DEMO

Sort the array below using insertion sort.

2	4	5	1	3
0	1	2	3	4

$i = 2$

INSERTION SORT DEMO

Sort the array below using insertion sort.

1	2	4	5	3
0	1	2	3	4

INSERT 1 at index $\mathbf{0}$

$i = 3$

INSERTION SORT DEMO

Sort the array below using insertion sort.

1	2	3	4	5
0	1	2	3	4

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.

1	2	5	4	3
0	1	2	3	4

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

$i = 0$

SELECTION SORT DEMO

Sort the array below using selection sort.

1	2	5	4	3
0	1	2	3	4

$i = 1$

SELECTION SORT DEMO

Sort the array below using selection sort.

1	2	3	4	5
0	1	2	3	4

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

$i = 2$

SELECTION SORT DEMO

Sort the array below using selection sort.

1	2	3	4	5
0	1	2	3	4

$i = 3$

SELECTION SORT DEMO

Sort the array below using selection sort.

1	2	3	4	5
0	1	2	3	4

$i = 4$

SELECTION SORT DEMO

Sort the array below using selection sort.

1	2	3	4	5
0	1	2	3	4

$i = 4$

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, A1 and A2, 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: $\mathrm{n} \log_2 n$

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++;
}
…
```

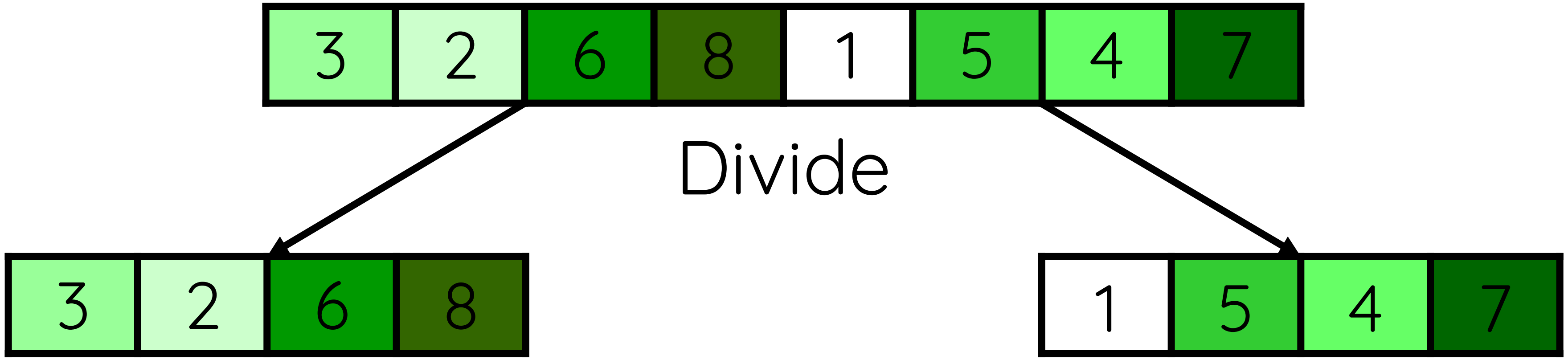
MERGE SORT

```
    /* 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++;
    }
}
```

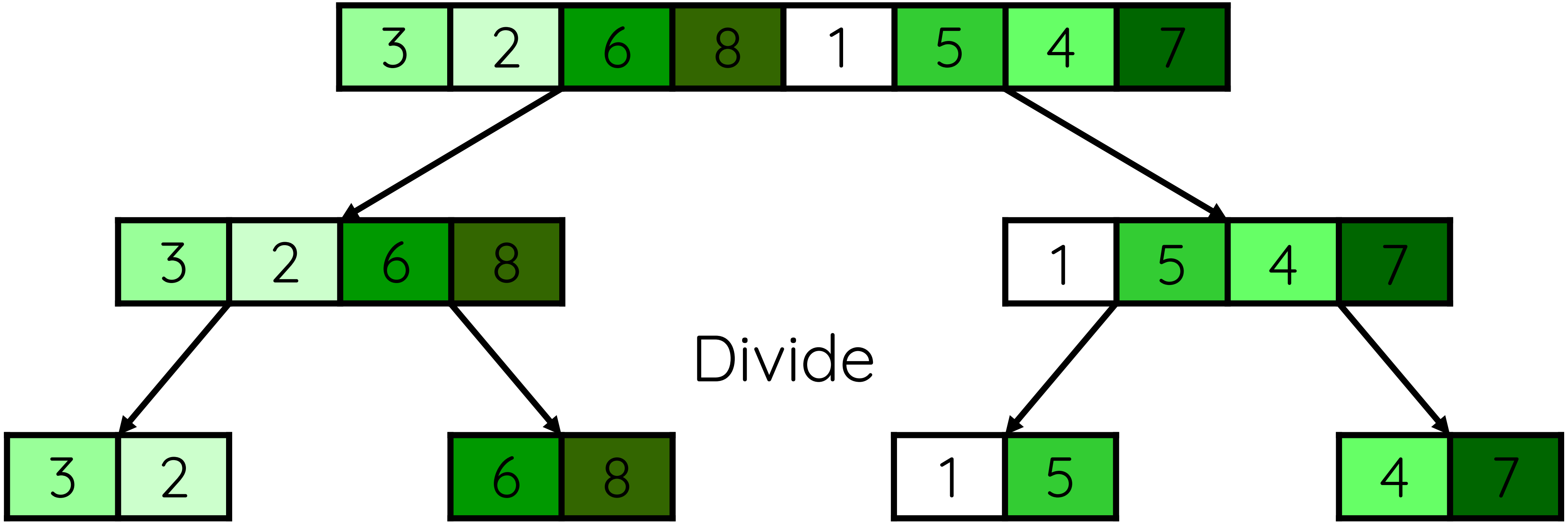
MERGE SORT DEMO

3	2	6	8	1	5	4	7

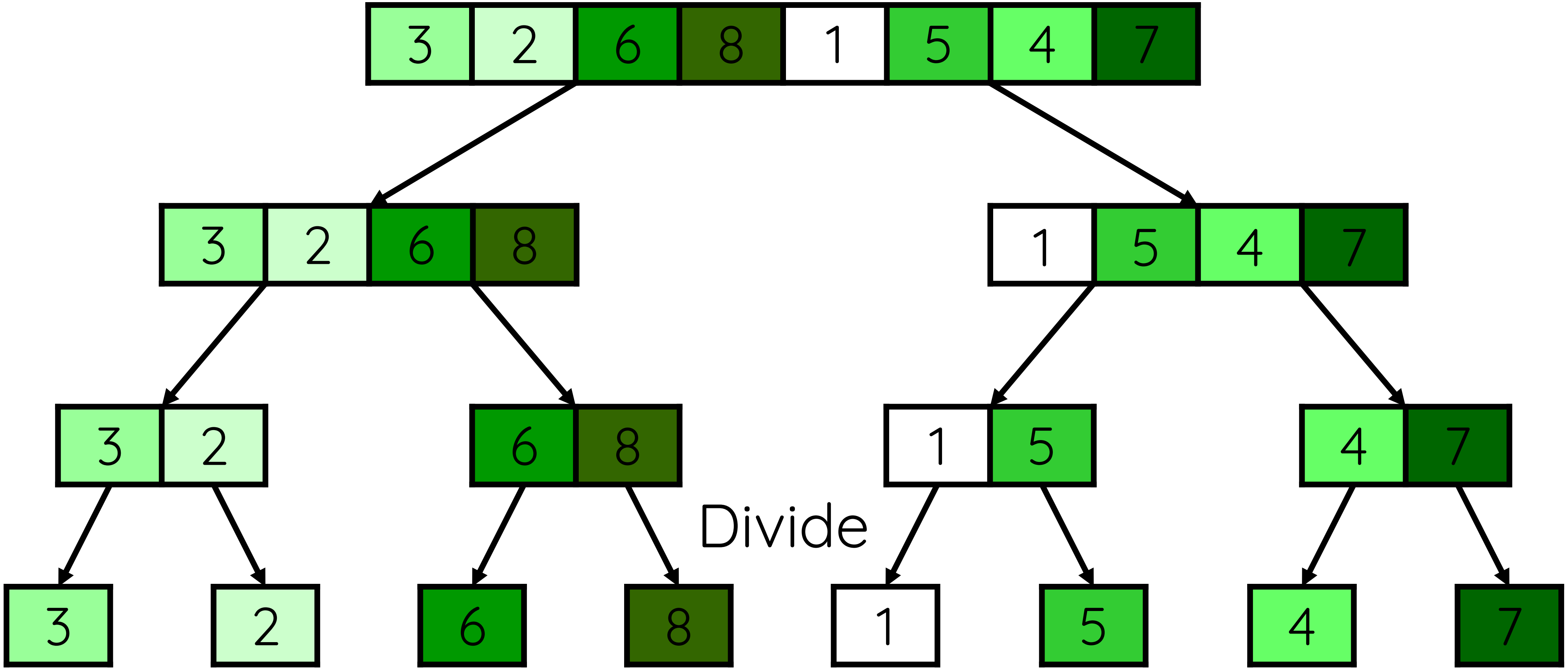
MERGE SORT DEMO



MERGE SORT DEMO



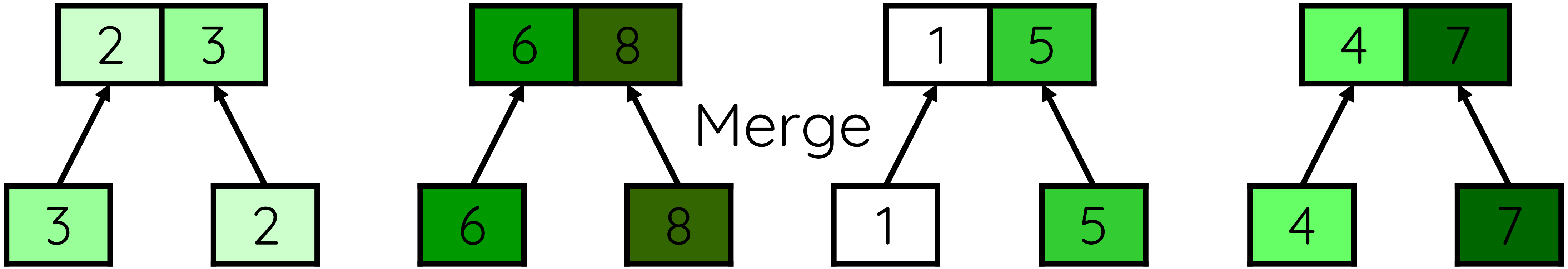
MERGE SORT DEMO



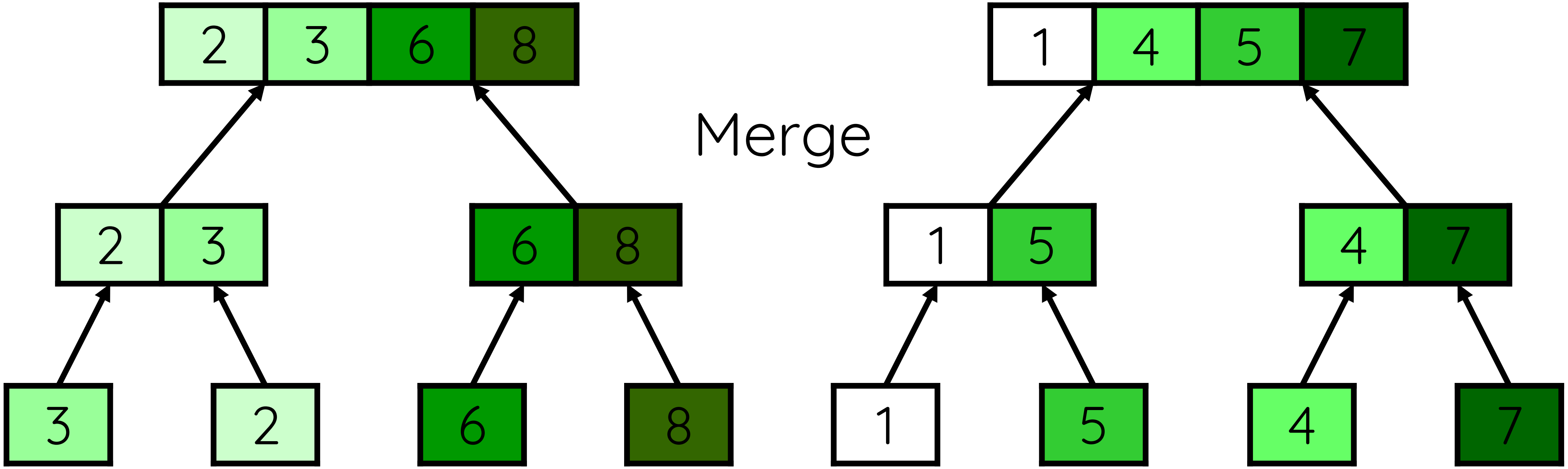
MERGE SORT DEMO

3	2	6	8	1	5	4	7

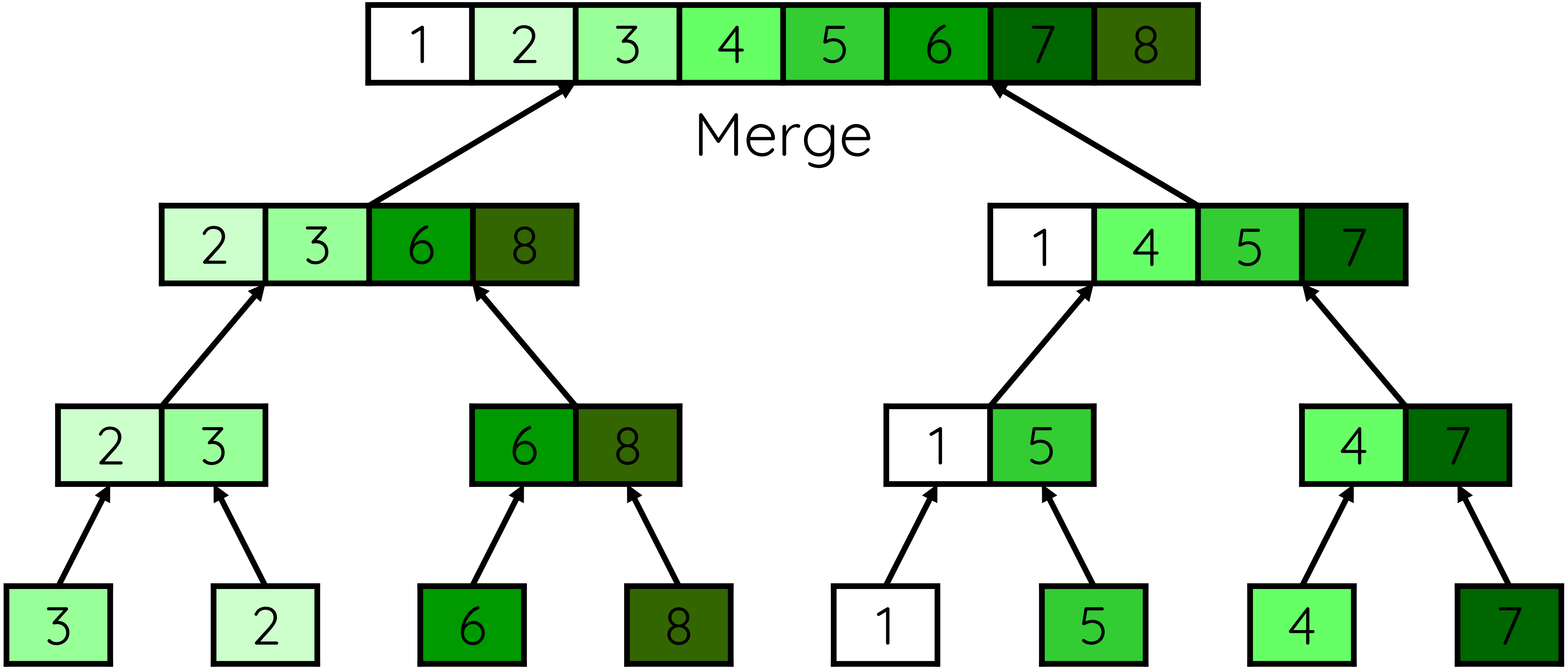
MERGE SORT DEMO

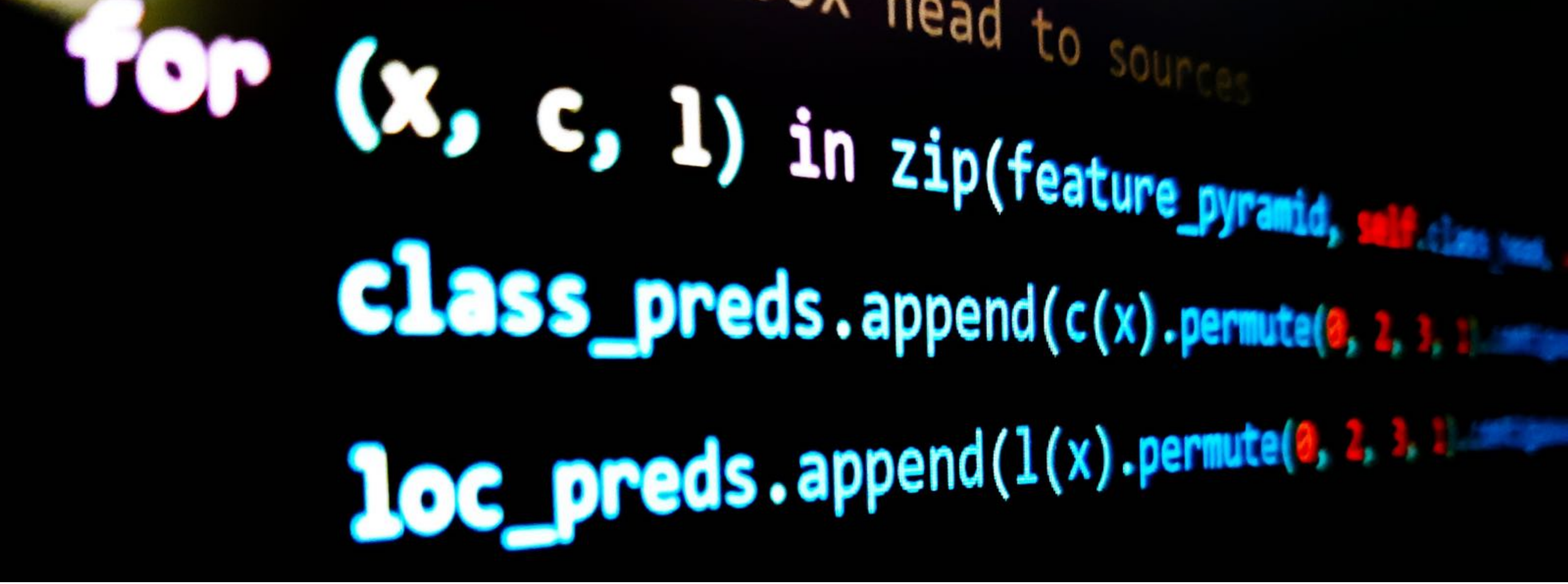


MERGE SORT DEMO



MERGE SORT DEMO





SEARCHING AND SORTING ARRAYS

CCDSALG T2 AY 2020-2021