# COMP2521 23T3
## Sorting Algorithms (II)

Kevin Luxa
cs2521@cse.unsw.edu.au

merge sort

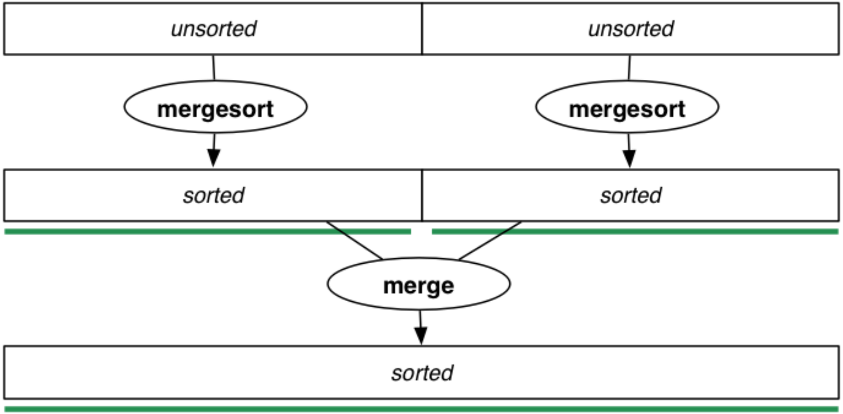
# Divide-and-Conquer Algorithms

divide-and-conquer algorithms
split a problem into smaller sub-problems,
solve the sub-problems recursively,
and then combine the results.

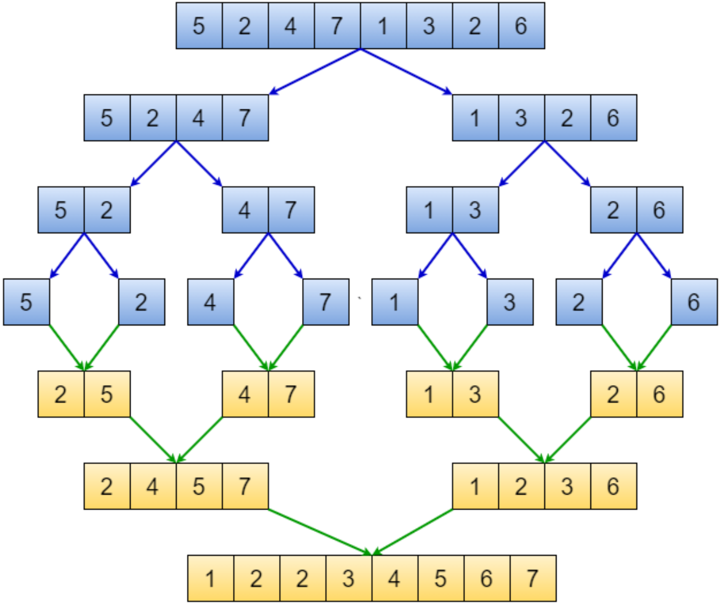
# Merge Sort

A divide-and-conquer sorting algorithm:

split the array into two roughly equal-sized parts.
recursively sort each of the partitions.
merge the two now-sorted partitions into a sorted array.



# Merge Sort


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

How do we split the array?

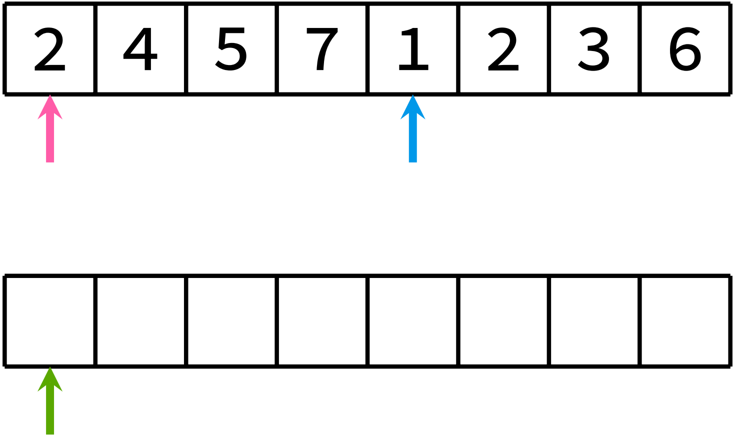
- We don't physically split the array
- We simply calculate the midpoint of the array
  - `mid = (lo + hi) / 2`
- Then recursively sort each half by passing in appropriate indices
  - Sort between indices `lo` and `mid`
  - Sort between indices `mid + 1` and `hi`
- This means the time complexity of splitting the array is $O(1)$

How do we merge two sorted subarrays?

- We merge the subarrays into a *temporary array*
- Keep track of the smallest element that has not been merged in each subarray
- Copy the smaller of the two elements into the temporary array
  - If the elements are equal, take from the left subarray
- Repeat until all elements have been merged
- Then copy from the temporary array back to the original array

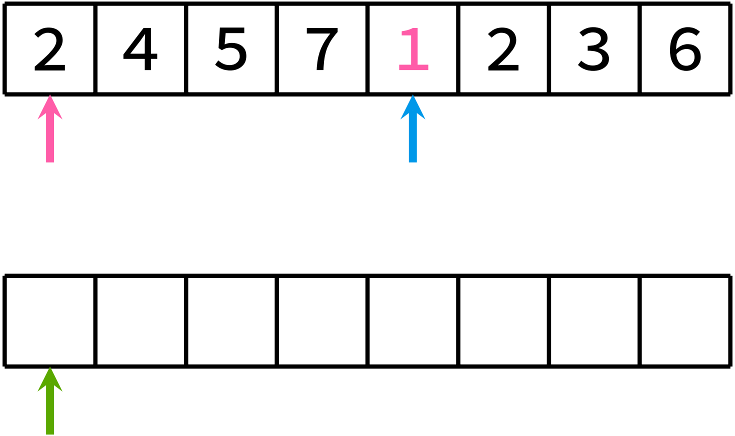
# Merge Sort

## Merging - Example 1



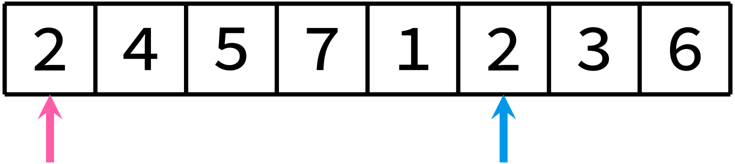
# Merge Sort

## Merging - Example 1

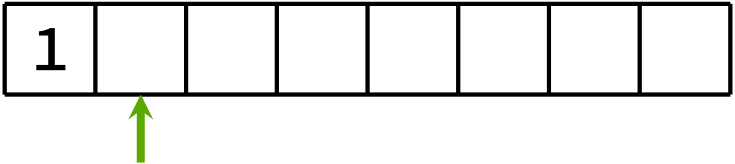


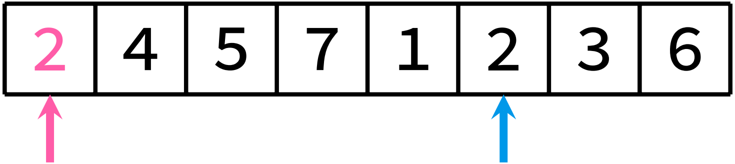
# Merge Sort

## Merging - Example 1

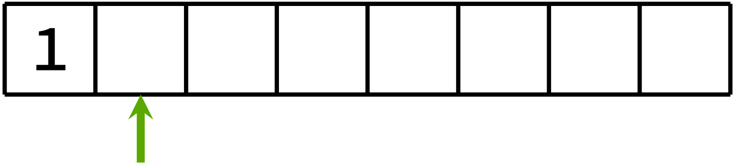



When items are equal, merge takes from the left subarray
(this ensures stability)



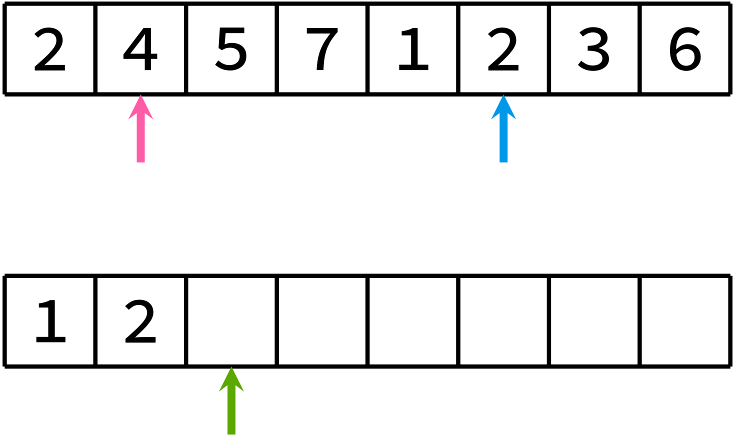


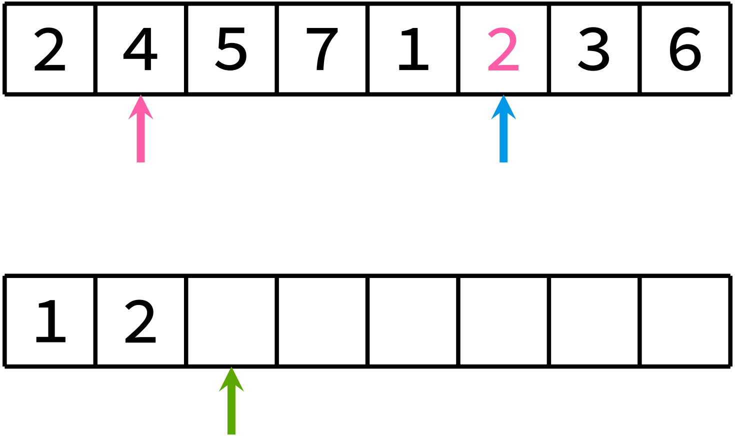

When items are equal, merge takes from the left subarray
(this ensures stability)



# Merge Sort

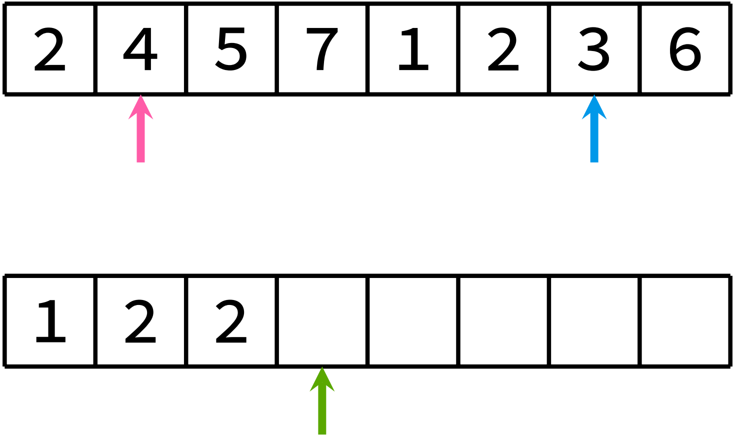
## Merging - Example 1

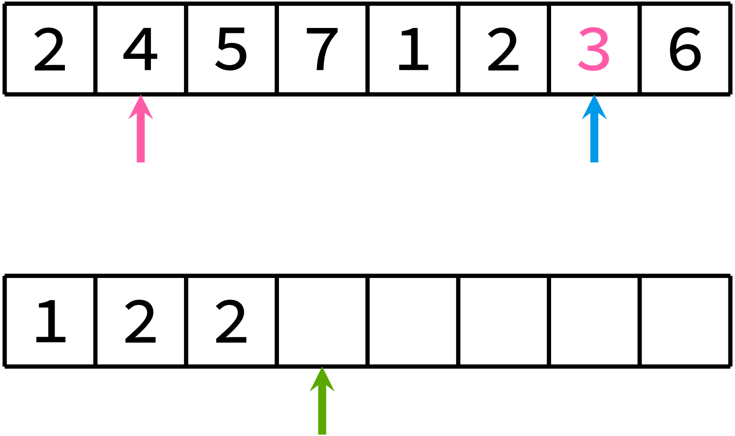



2
4
5
7
1
2
3
6
1
2

# Merge Sort

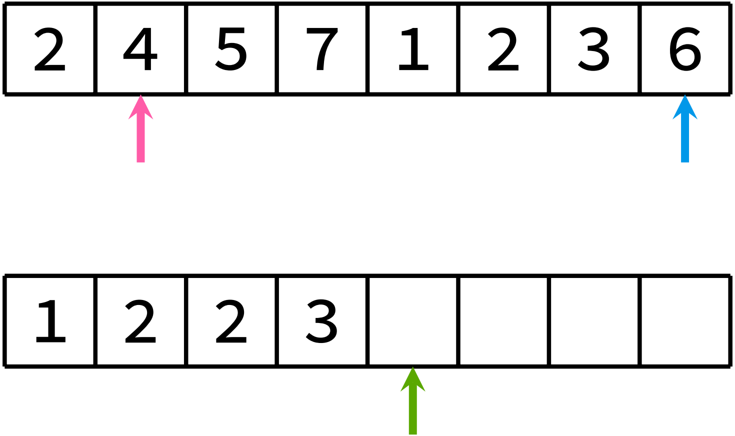
## Merging - Example 1




2 4 5 7 1 2 3 6
1 2 2

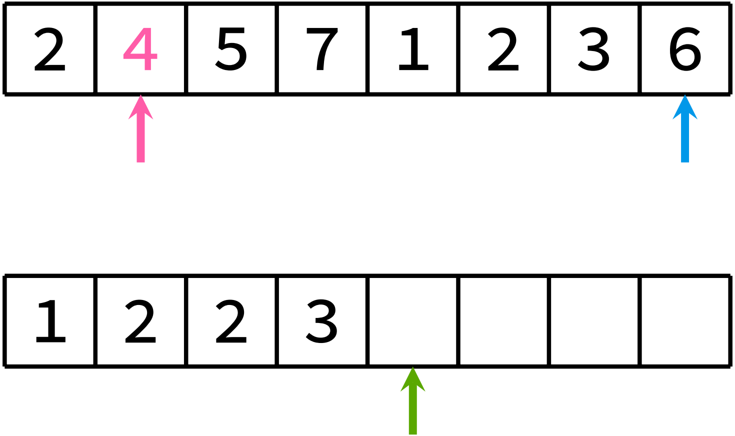
# Merge Sort

## Merging - Example 1



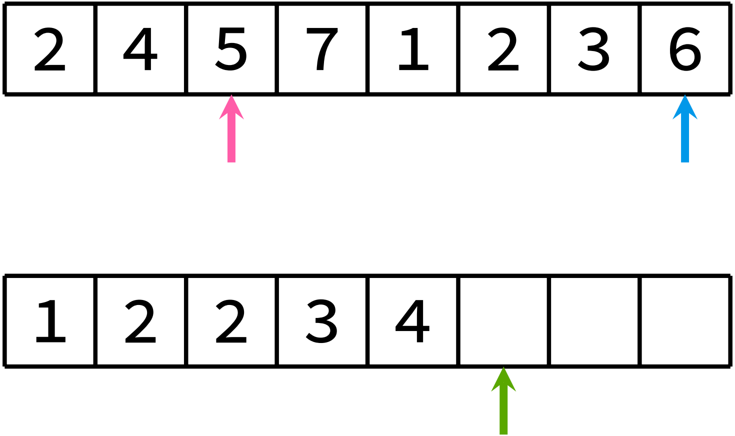
# Merge Sort

## Merging - Example 1



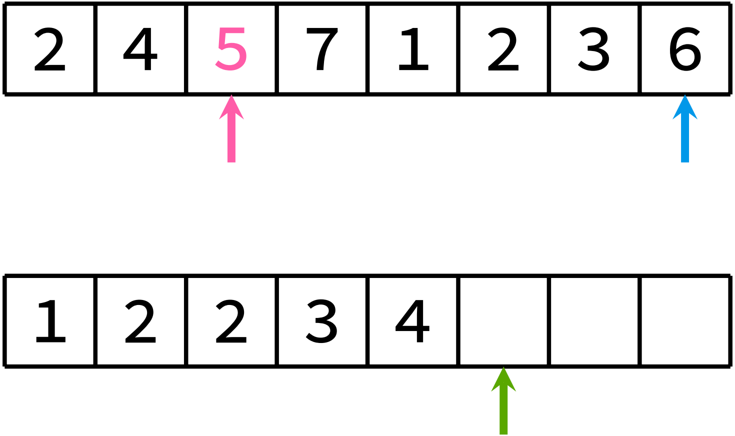
# Merge Sort

## Merging - Example 1

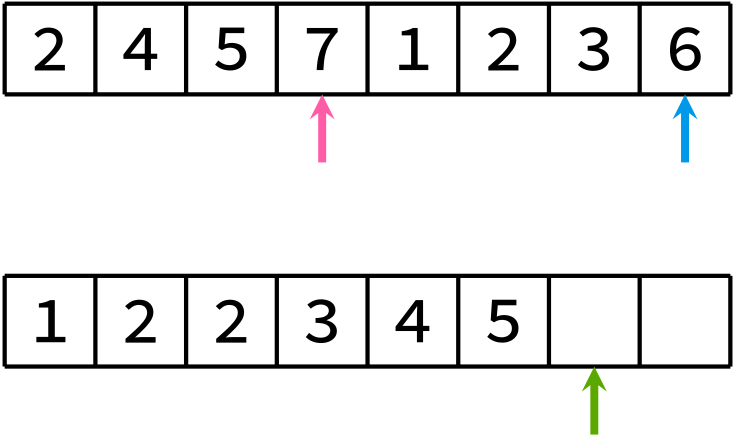


# Merge Sort

## Merging - Example 1

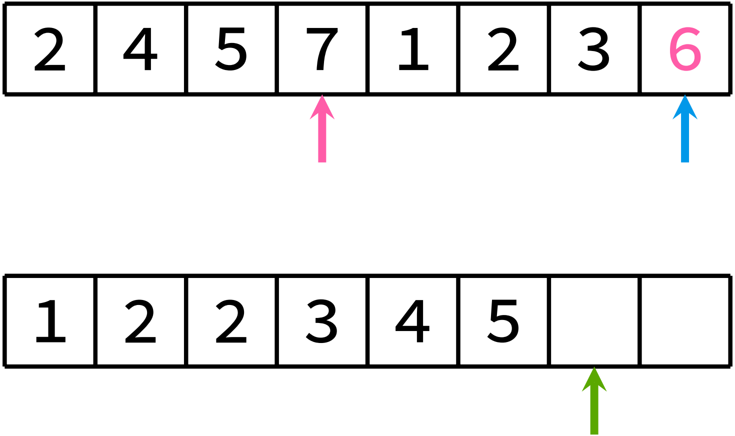


# Merge Sort
## Merging - Example 1



# Merge Sort

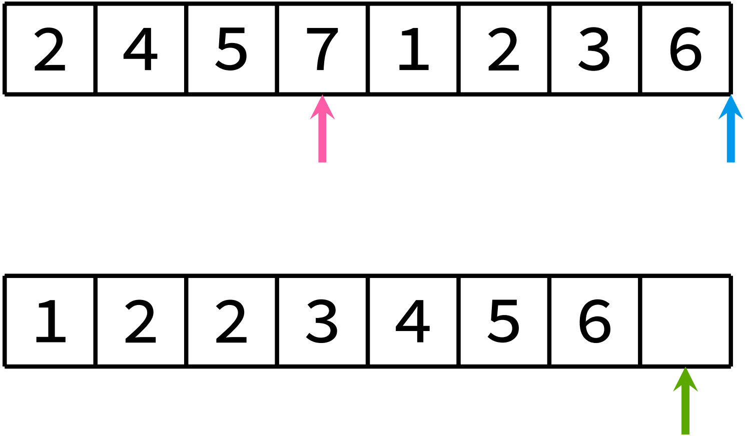
## Merging - Example 1



# Merge Sort

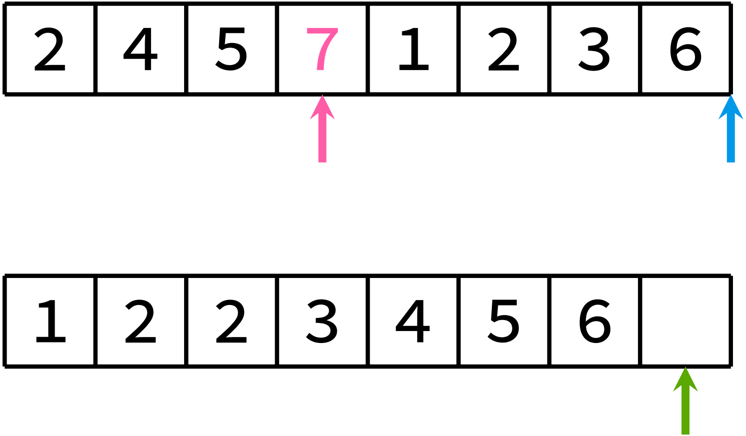
## Merging - Example 1





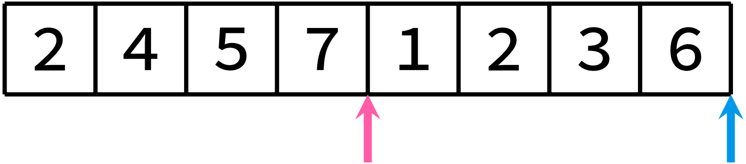
# Merge Sort

## Merging - Example 1

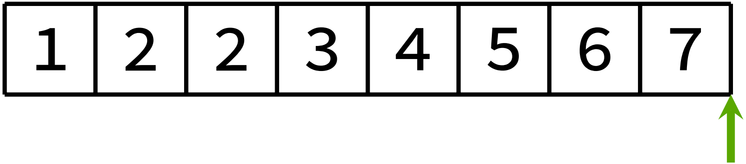


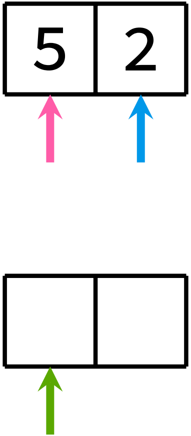
# Merge Sort

## Merging - Example 1




Now copy back to original array

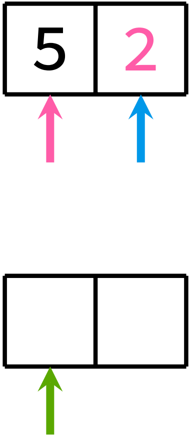


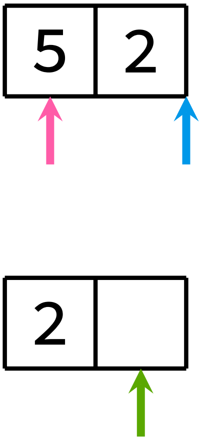

5
2

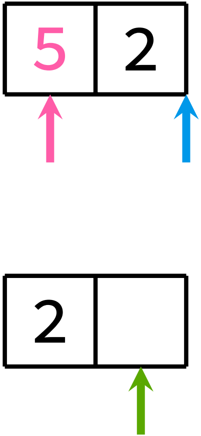
# Merge Sort

## Merging - Example 2





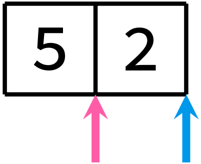

5
2
2


5
2
2

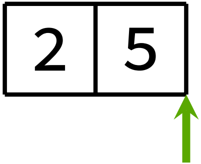
# Merge Sort

## Merging - Example 2






Now copy back to original array



- The time complexity of merging two sorted subarrays is $O(n)$, where $n$ is the total number of elements in both subarrays
- Therefore:
  - Merging two subarrays of size 1 takes 2 “steps”
  - Merging two subarrays of size 2 takes 4 “steps”
  - Merging two subarrays of size 4 takes 8 “steps”
  - ...

# Merge Sort

## C Implementation: Sort

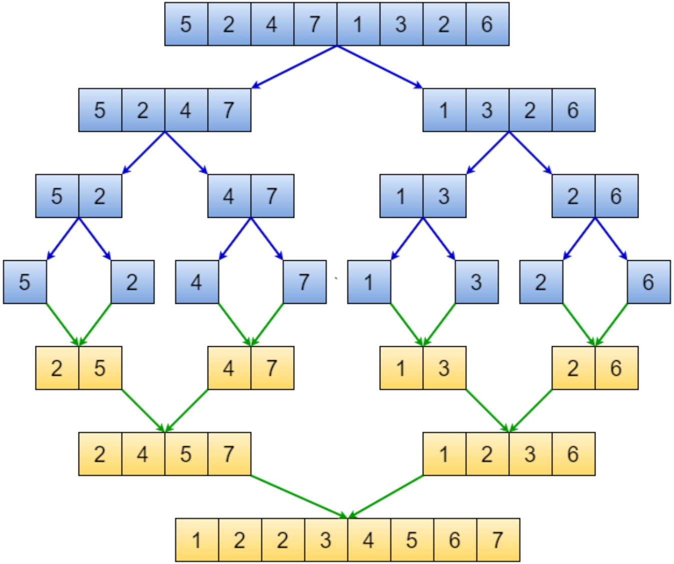
```c
void mergeSort(Item items[], int lo, int hi) {
    if (lo >= hi) return;
    int mid = (lo + hi) / 2;
    mergeSort(items, lo, mid);
    mergeSort(items, mid + 1, hi);
    merge(items, lo, mid, hi);
}
```

# Merge Sort

## C Implementation: Merge



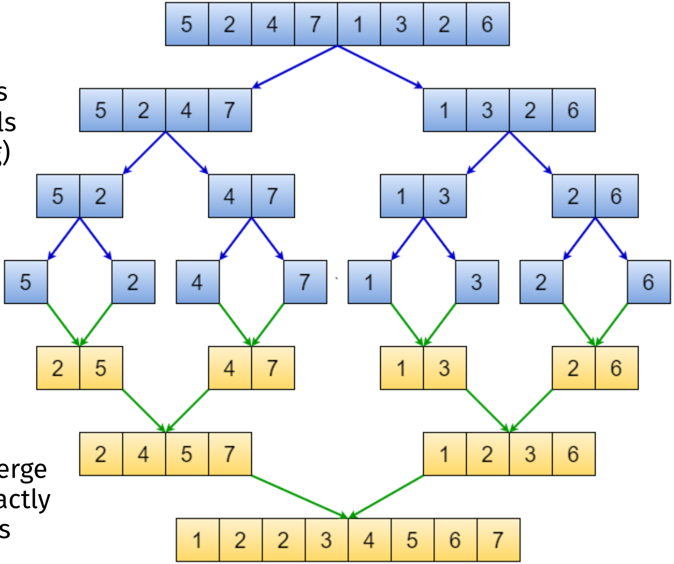
```c
void merge(Item items[], int lo, int mid, int hi) {
    Item *tmp = malloc((hi - lo + 1) * sizeof(Item));
    int i = lo, j = mid + 1, k = 0;

    // Scan both segments, copying to `tmp'.
    while (i <= mid && j <= hi) {
        if (le(items[i], items[j])) {
            tmp[k++] = items[i++];
        } else {
            tmp[k++] = items[j++];
        }
    }

    // Copy items from unfinished segment.
    while (i <= mid) tmp[k++] = items[i++];
    while (j <= hi) tmp[k++] = items[j++];

    // Copy `tmp' back to main array.
    for (i = lo, k = 0; i <= hi; i++, k++) {
        items[i] = tmp[k];
    }

    free(tmp);
}
```


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

# Merge Sort
## Analysis

**Split**

$n-1$ splits
($\log_2 n$ levels of splitting)




$O(n)$

**Merge**

We have to merge $n$ numbers exactly $\log_2 n$ times

$O(n \log n)$

Analysis:

- Merge sort splits the array into equal-sized partitions halving at each level $\Rightarrow \log_2 n$ levels
- The same operations happen at every recursive level
- Each ‘level’ requires $\leq n$ comparisons

Therefore:

- The time complexity of merge sort is $O(n \log n)$
  - Best-case, average-case, and worst-case time complexities are all the same

Note: Not required knowledge in COMP2521!

Let $T(n)$ be the time taken to sort $n$ elements.

Splitting arrays into two halves takes constant time.
Merging two sorted arrays takes $n$ steps.

So we have that:

$$T(n) = 2T(n/2) + n$$

Then the Master Theorem (see COMP3121) can be used to show that the time complexity is $O(n \log n)$.

# Merge Sort

## Properties

**Stable**

Due to taking from left subarray if items are equal during merge

**Non-adaptive**

$O(n \log n)$ best case, average case, worst case

**Not in-place**

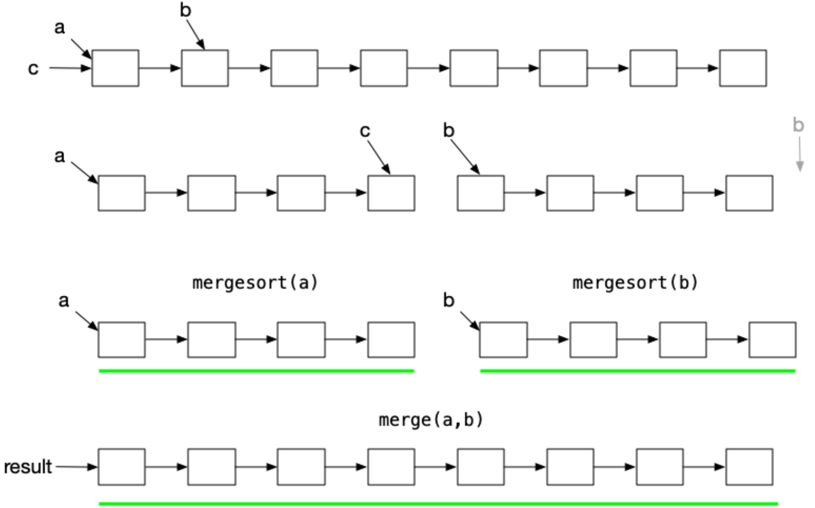
Merge uses a temporary array of size up to $n$

Note: Merge sort also uses $O(\log n)$ stack space

# Merge Sort on Lists



It is possible to apply merge sort on linked lists.



# Bottom-Up Merge Sort



An approach that works non-recursively!

- on each pass, our array contains sorted *runs* of length $m$.
- initially, $n$ sorted runs of length $1$.
- The first pass merges adjacent elements into runs of length $2$.
- The second pass merges adjacent elements into runs of length $4$.
- ... continue until we have a single sorted run of length $n$.

Can be used for *external* sorting;
*e.g.*, sorting disk-file contents

# Bottom-Up Merge Sort

## Example

Method
Splitting
Merging
Implementation
Analysis
Properties
Sorting Lists
Bottom-Up
Implementation
Appendix

| | [0] | [1] | [2] | | | | | | ..... | | | | | | | [15] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Original* | A | S | O | R | T | I | N | G | E | X | E | M | P | L | A | R |
| *After 1st pass* *sorted slices of length 2* | A | S | O | R | I | T | G | N | E | X | E | M | L | P | A | R |
| *After 2nd pass* *sorted slices of length 4* | A | O | R | S | G | I | N | T | E | E | M | X | A | L | P | R |
| *After 3rd pass* *sorted slices of length 8* | A | G | I | N | O | R | S | T | A | E | E | L | M | P | R | X |
| *After 4th pass* *sorted slice of length 16* | A | A | E | E | G | I | L | M | N | O | P | R | R | S | T | X |

# Bottom-Up Merge Sort

## C Implementation

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))

void mergeSortBottomUp(Item items[], int lo, int hi) {
    for (int m = 1; m <= lo - hi; m *= 2) {
        for (int i = lo; i <= hi - m; i += 2 * m) {
            int end = MIN(i + 2 * m - 1, hi);
            merge(items, i, i + m - 1, end);
        }
    }
}
```

# Feedback

https://forms.office.com/r/aPF09YHZ3X

# Appendix

# Merge Sort

## Demo (I)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 1 | 7 | 3 | 8 | 6 | 5 | 2 |

# Merge Sort

## Demo (I)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 1 | 7 | 3 | 8 | 6 | 5 | 2 |

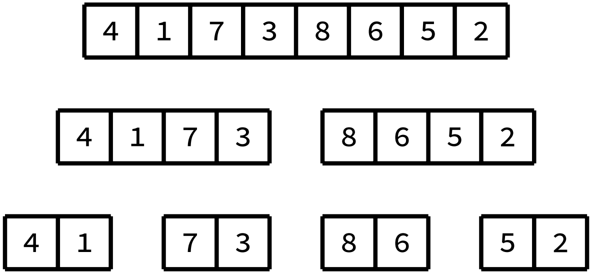
| | | | |
|---|---|---|---|
| 4 | 1 | 7 | 3 |

| | | | |
|---|---|---|---|
| 8 | 6 | 5 | 2 |

# Merge Sort

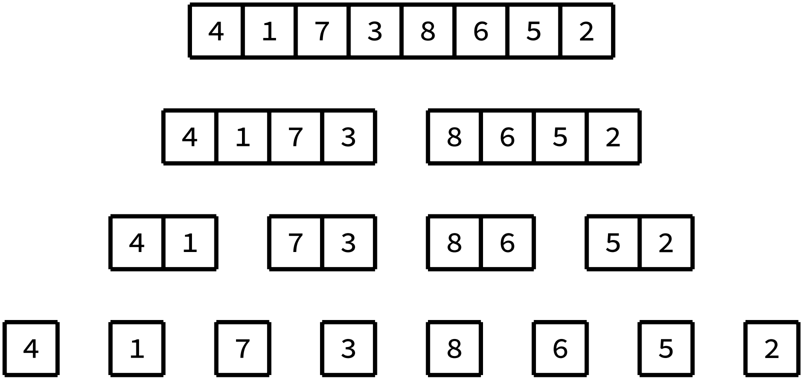
## Demo (I)





# Merge Sort

## Demo (I)



# Merge Sort

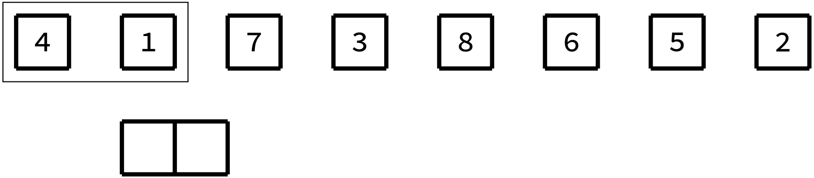
## Demo (II)



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 1 | 7 | 3 | 8 | 6 | 5 | 2 |

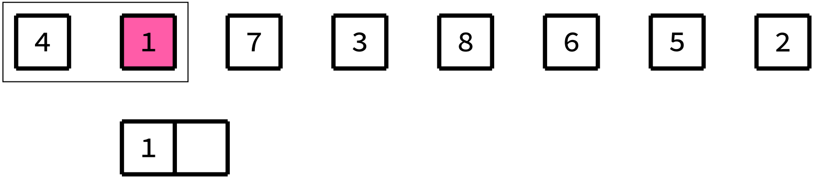
# Merge Sort

## Demo (II)



# Merge Sort

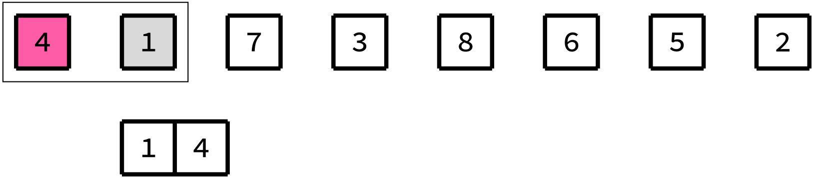
## Demo (II)



# Merge Sort

## Demo (II)

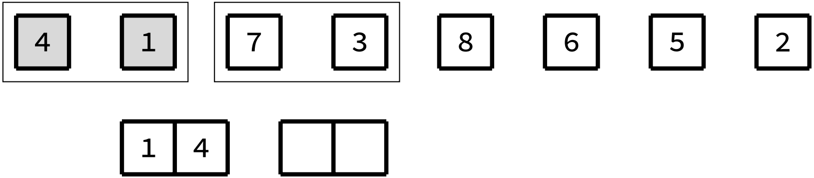




# Merge Sort

## Demo (II)

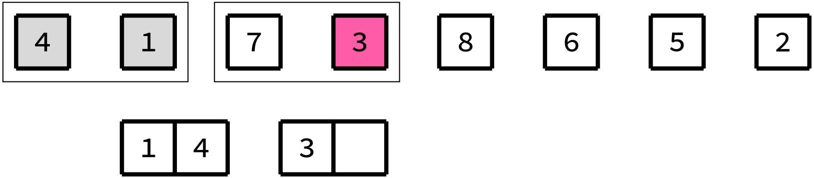




# Merge Sort

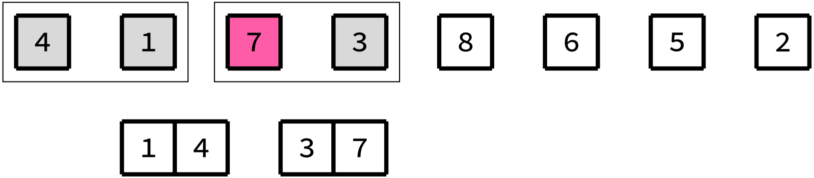
## Demo (II)





# Merge Sort

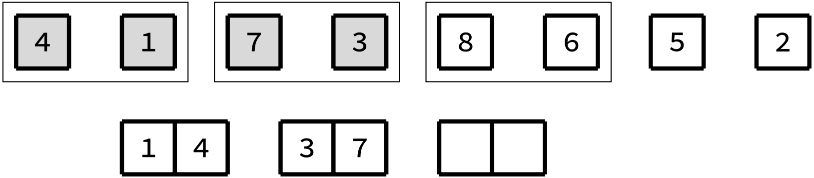
## Demo (II)



# Merge Sort

## Demo (II)

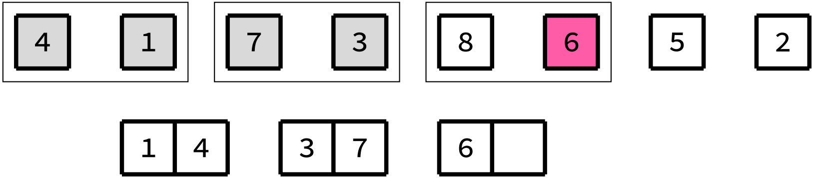




# Merge Sort

## Demo (II)

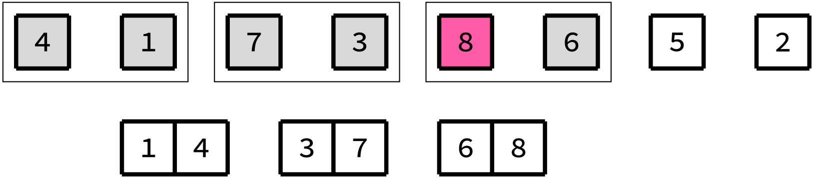




# Merge Sort

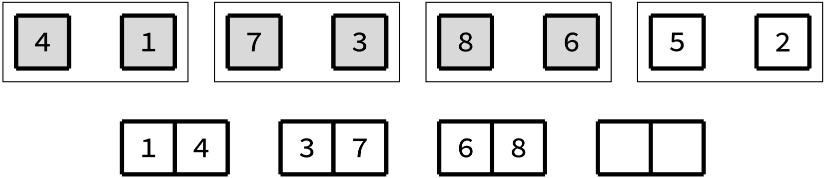
## Demo (II)



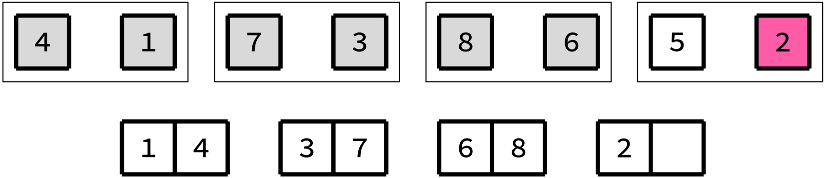


# Merge Sort

## Demo (II)

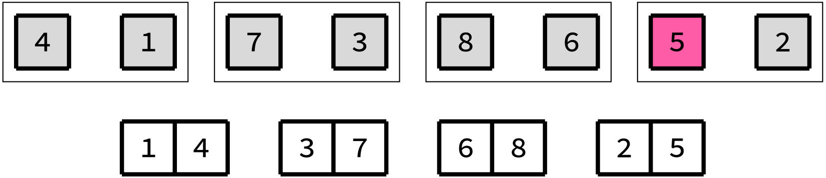


# Merge Sort
## Demo (II)



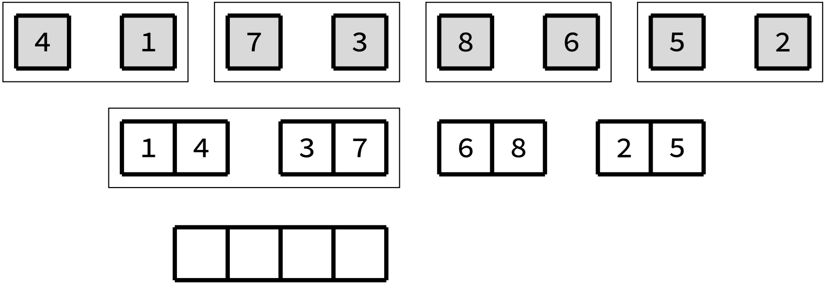
# Merge Sort

## Demo (II)



# Merge Sort
## Demo (II)

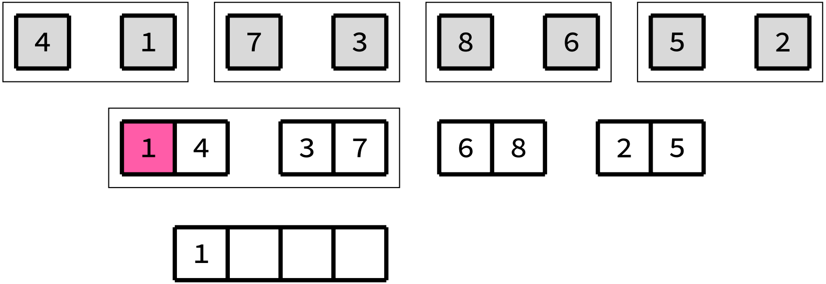




# Merge Sort

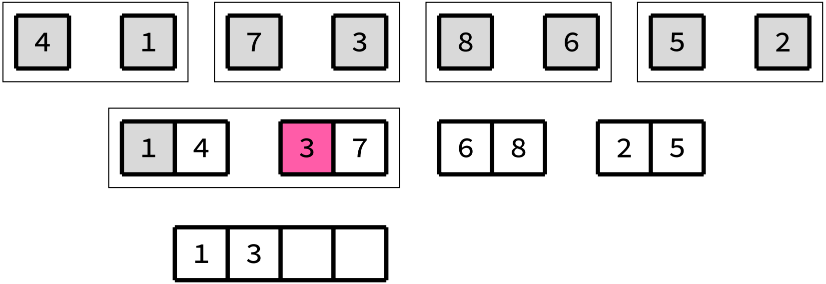
## Demo (II)





# Merge Sort
## Demo (II)

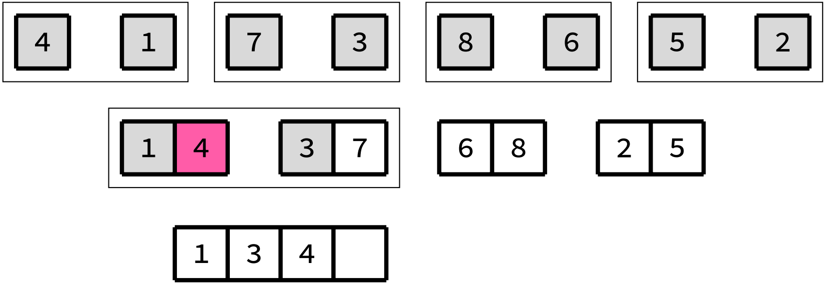




# Merge Sort

## Demo (II)

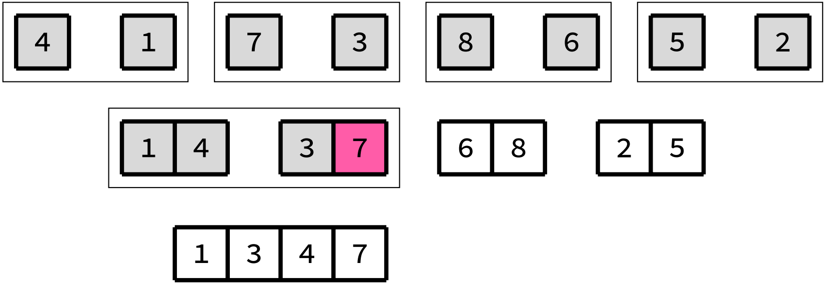




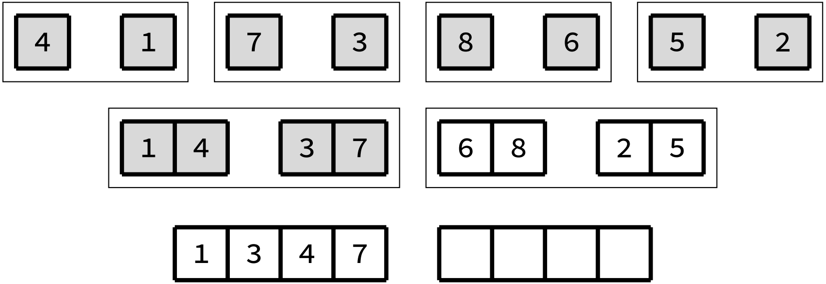
# Merge Sort

## Demo (II)





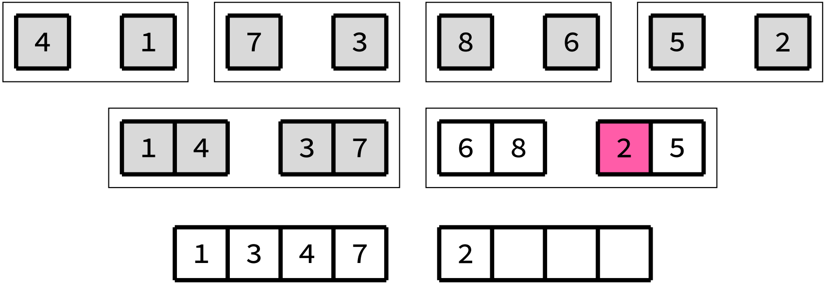
# Merge Sort
## Demo (II)



# Merge Sort

## Demo (II)

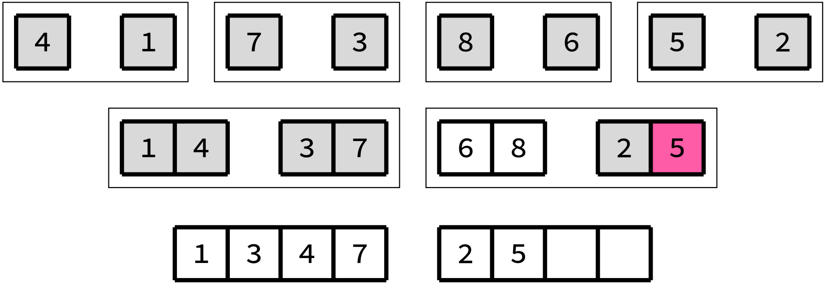




# Merge Sort

## Demo (II)

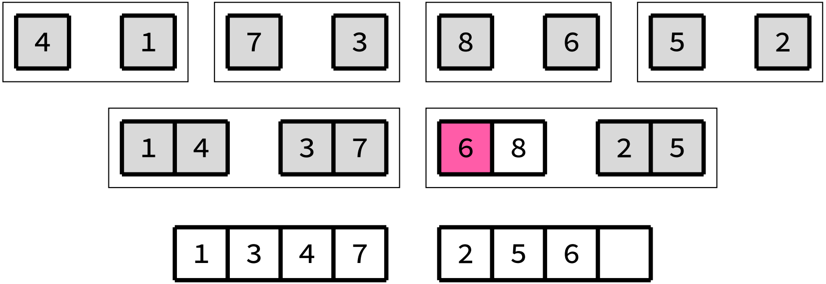




# Merge Sort

## Demo (II)

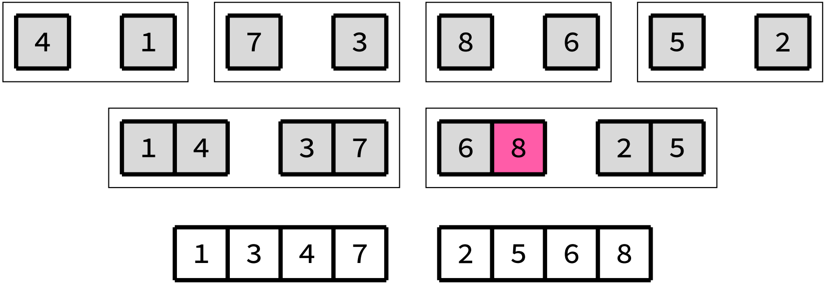




# Merge Sort

## Demo (II)

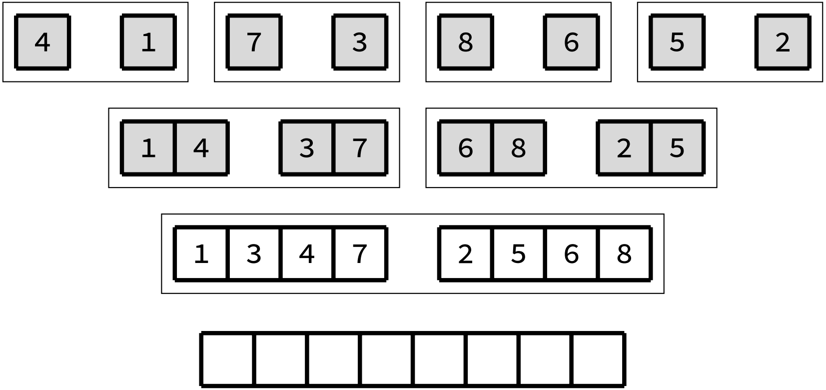




# Merge Sort

## Demo (II)

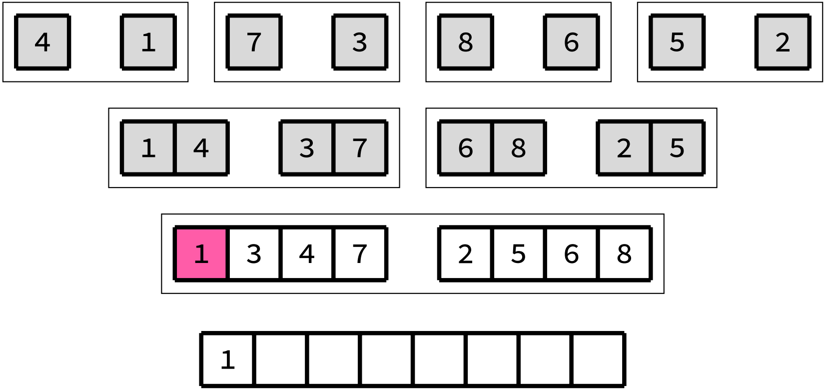




# Merge Sort

## Demo (II)

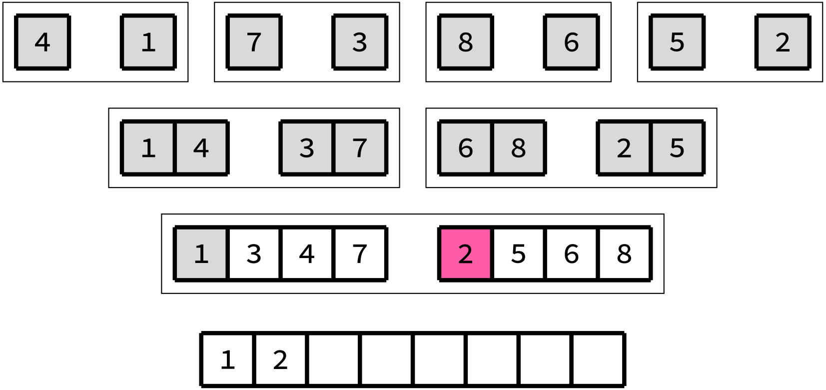




# Merge Sort

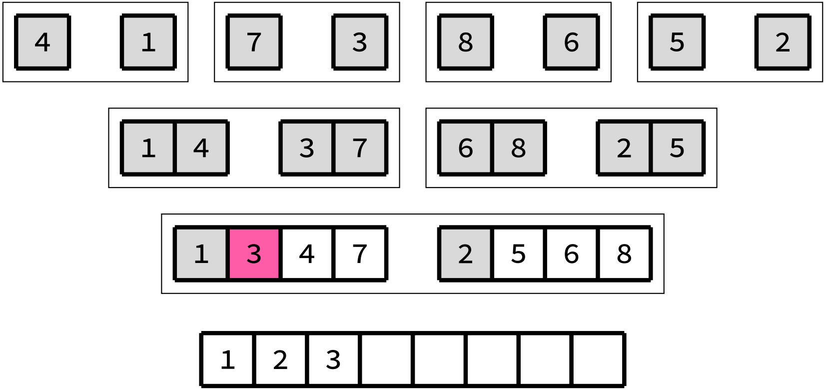
## Demo (II)





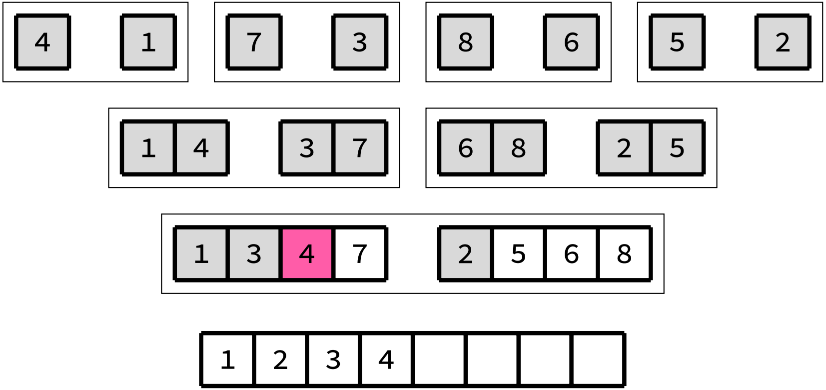
# Merge Sort

## Demo (II)



# Merge Sort

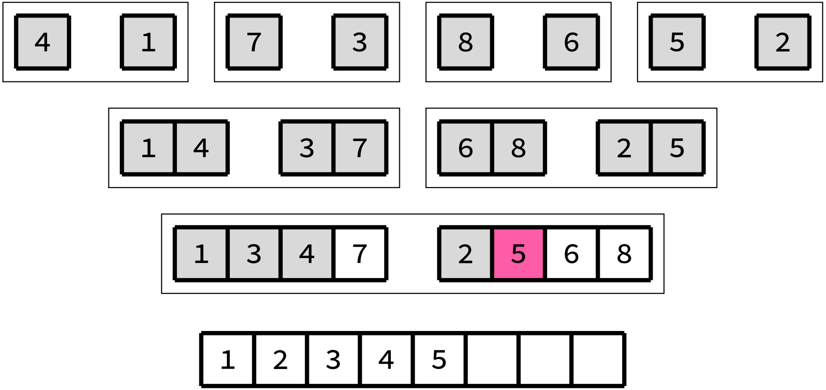
## Demo (II)



# Merge Sort

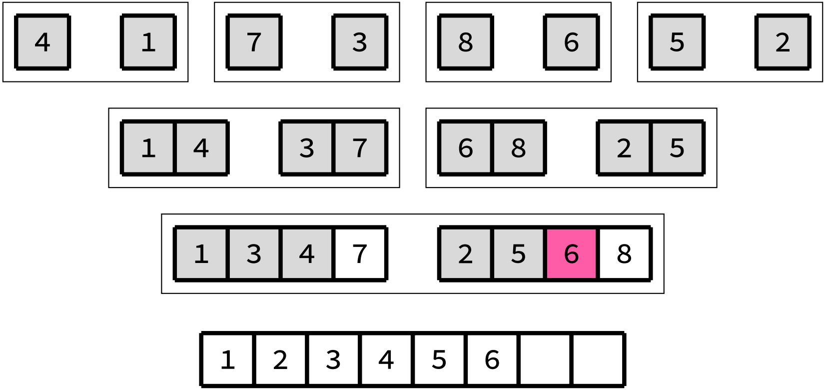
## Demo (II)





# Merge Sort
## Demo (II)

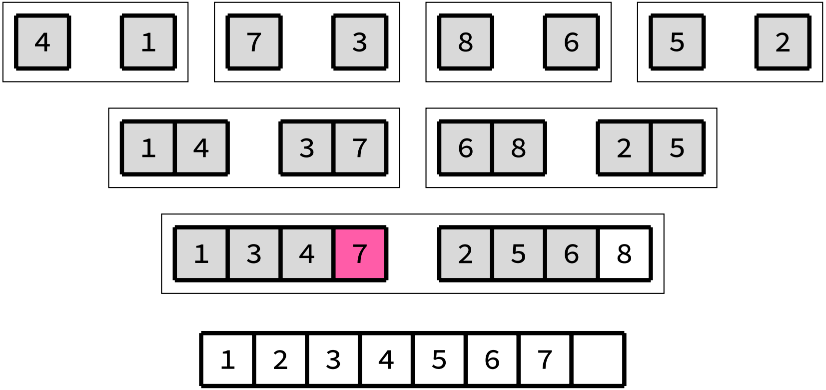




# Merge Sort

## Demo (II)

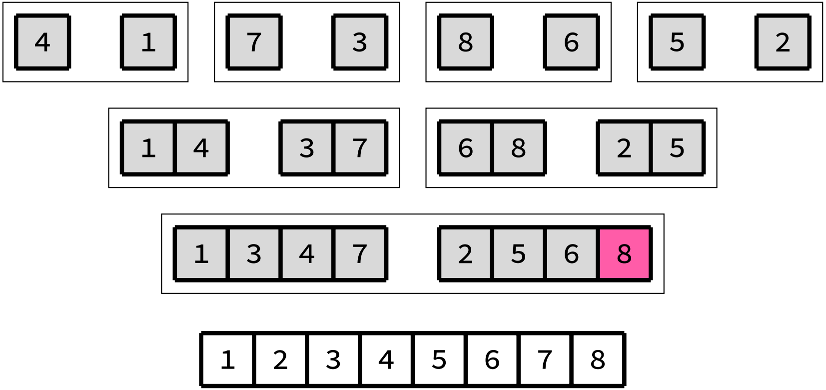




# Merge Sort

## Demo (II)





# Merge Sort

## Demo (II)