Merge sort

Dr Yuhua Li
School of Computer Science & Informatics

(Content adapted from slides by Dr Louise Knight)

Outline of lecture

- Divide-and-conquer
- Merge sort
- Sorting summary
- Stability in sorting
- Sorting records

Divide-and-conquer algorithms

- The divide-and-conquer approach solves a problem by
  - Dividing the data into parts,
  - Finding subsolutions to each part,
  - Constructing the final answer from the subsolutions.
- Often, but not always, the divide-and-conquer approach results in a recursive algorithm.

Recursion example

- Sum of squares

Algorithm SumSquares(n)
Input: An integer, n.
Output: The sum of squares 1..n.
if $n = 1$ **then return** (1)
else return $(n * n + \text{SumSquares}(n-1))$

- Note: recursion terminates when n = 1. This is handled using a conditional statement

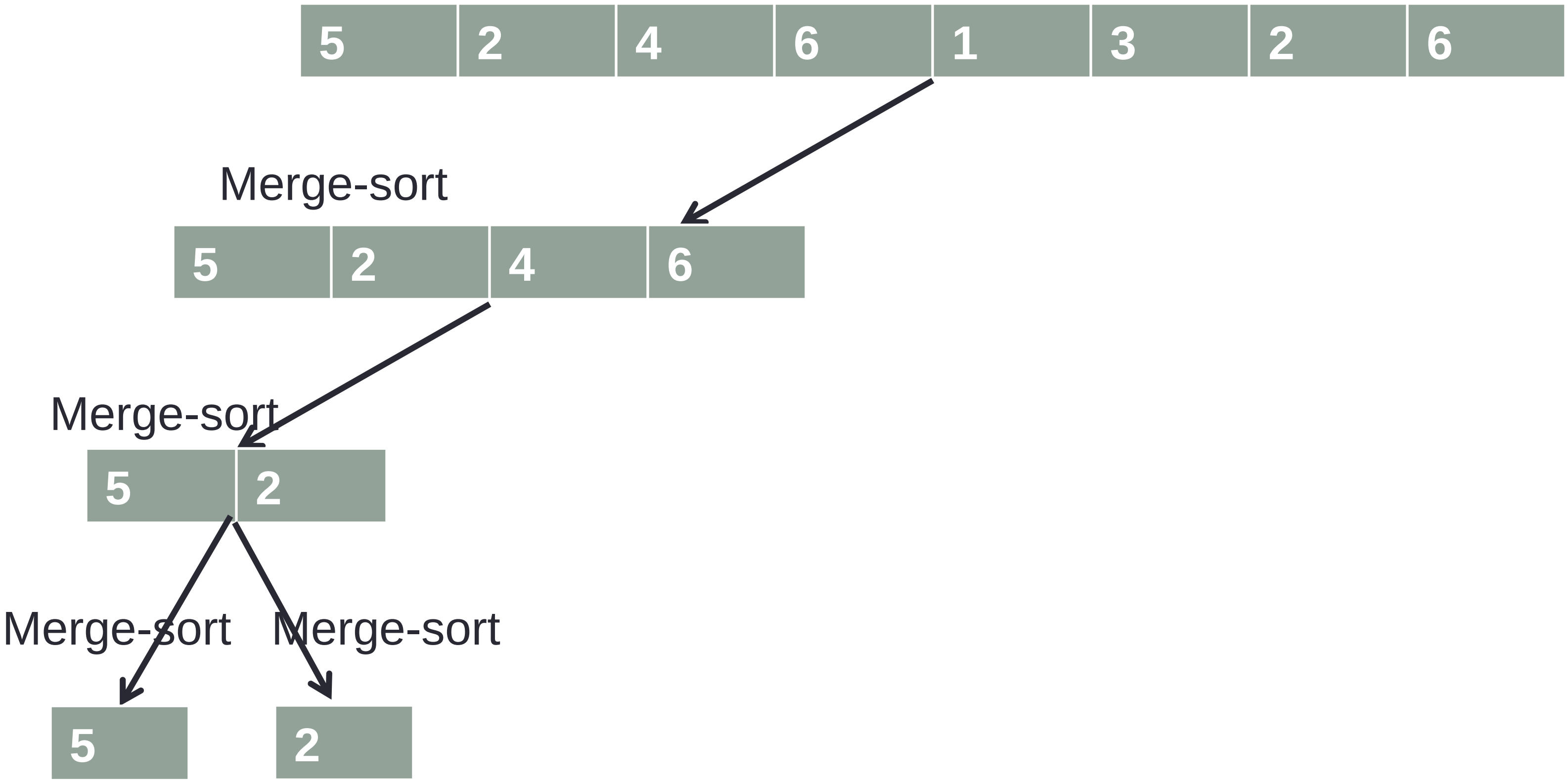
Merge sort

- This is an example of a divide-and-conquer algorithm
- S is a sequence
- **Divide:** if S has at least 2 elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S
- **Recurse:** recursively sort sequences S_1 and S_2
- **Conquer:** put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence

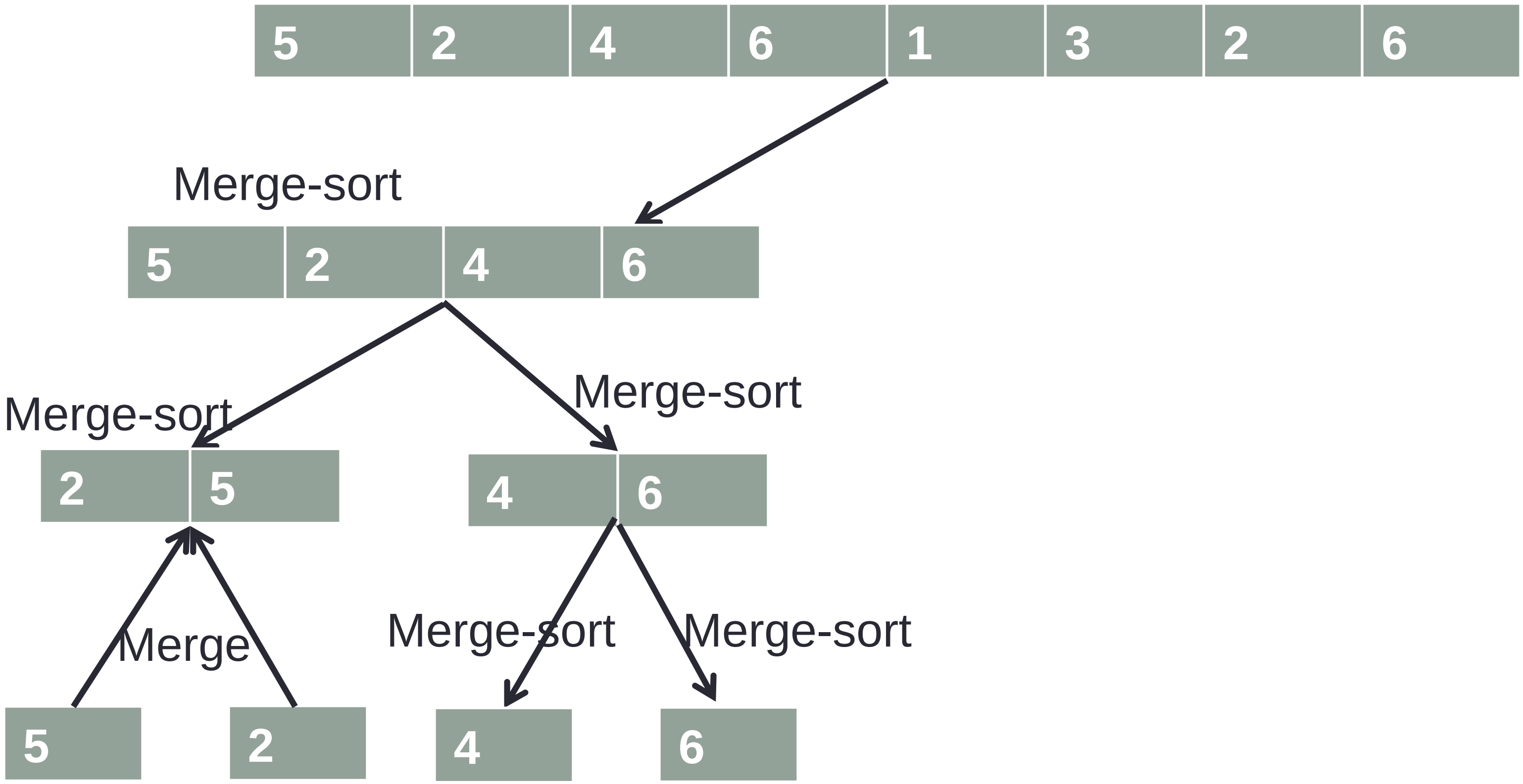
Merge sort pseudocode outline

Algorithm merge-sort(S)
    Input: An array, S.
    Output: The array, S sorted
    {**if** more than one element **then**}
        {divide into two subsets}
        {merge-sort left subset}
        {merge-sort right subset}
        {*merge together left and right subsets*}

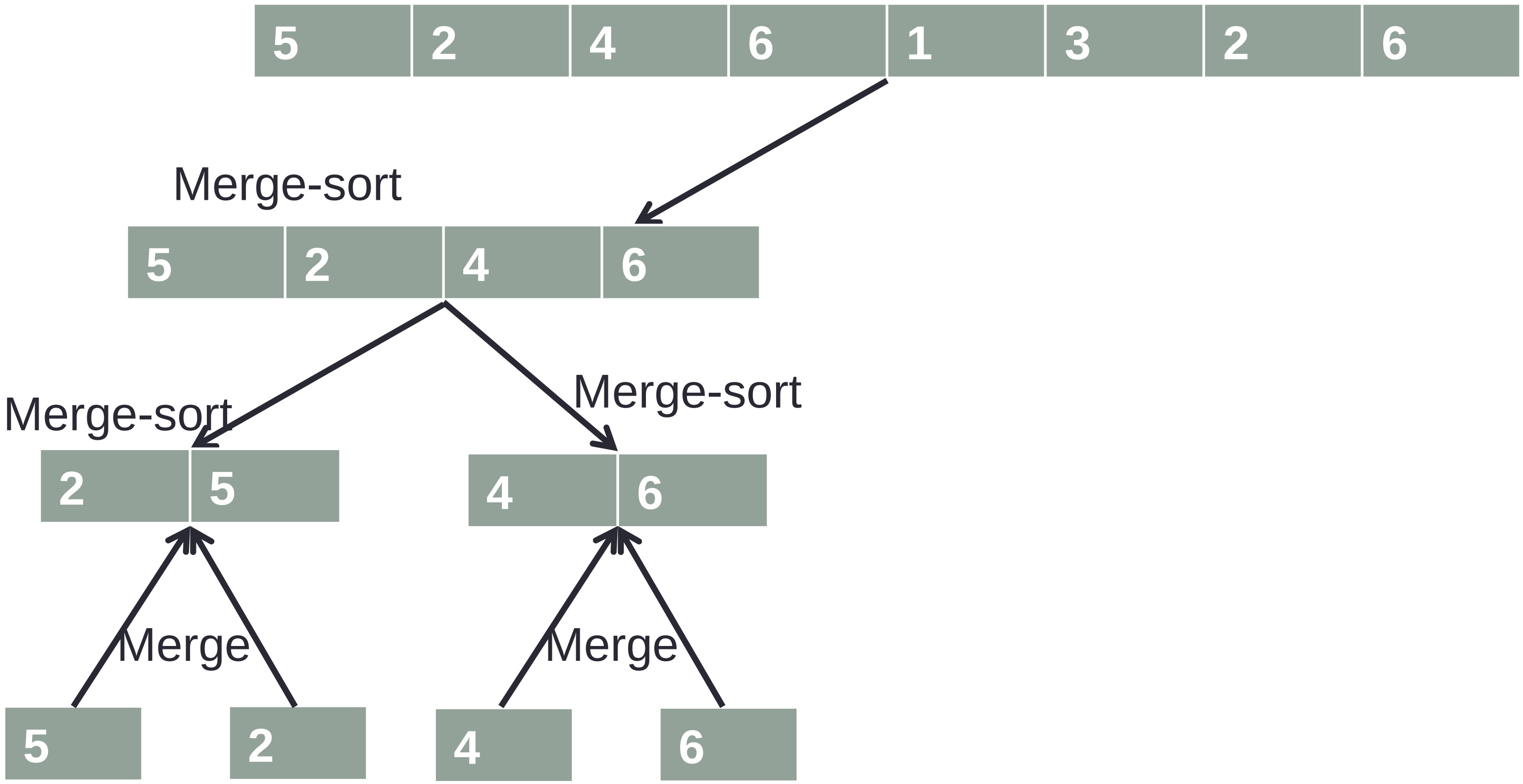
Merge sort example



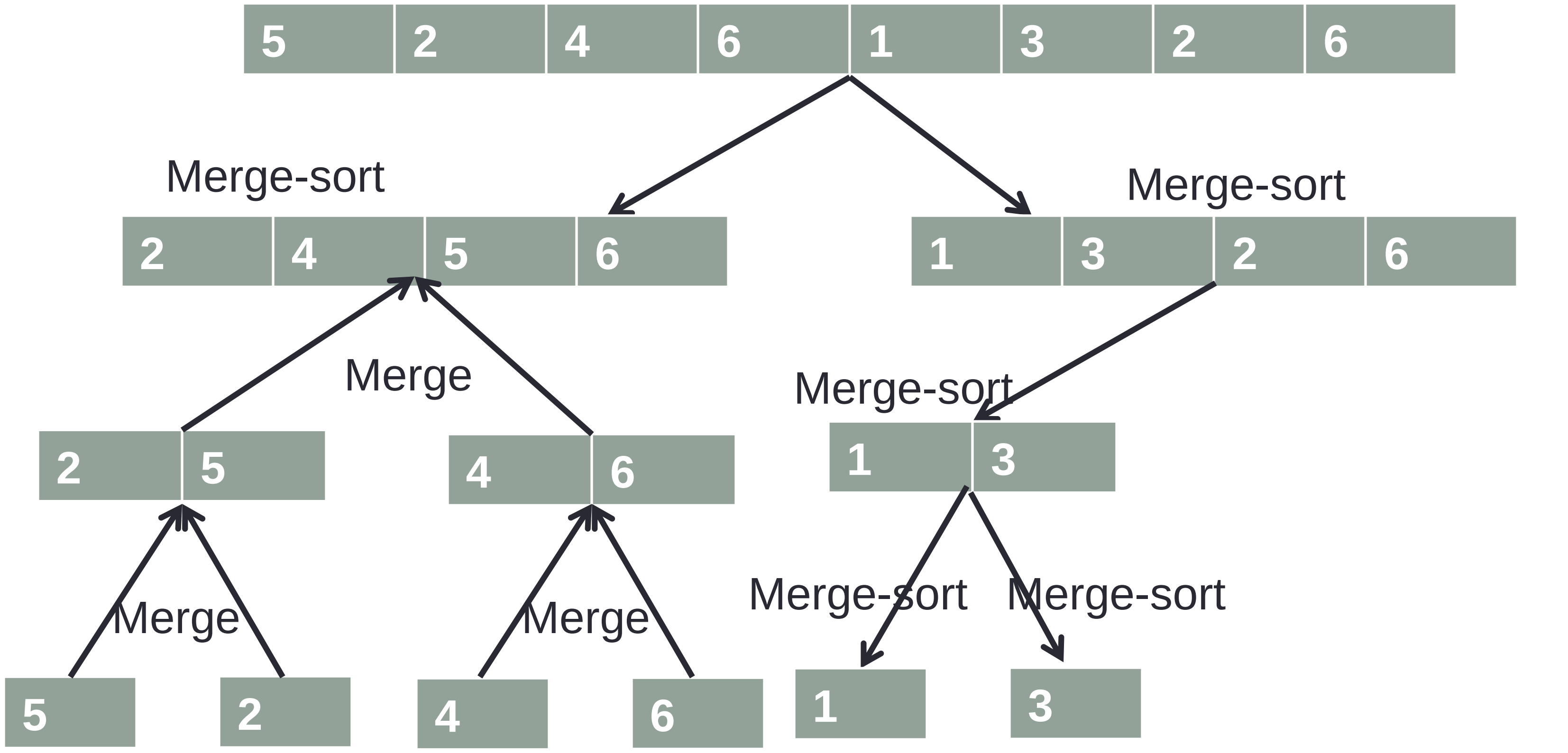
Merge sort example



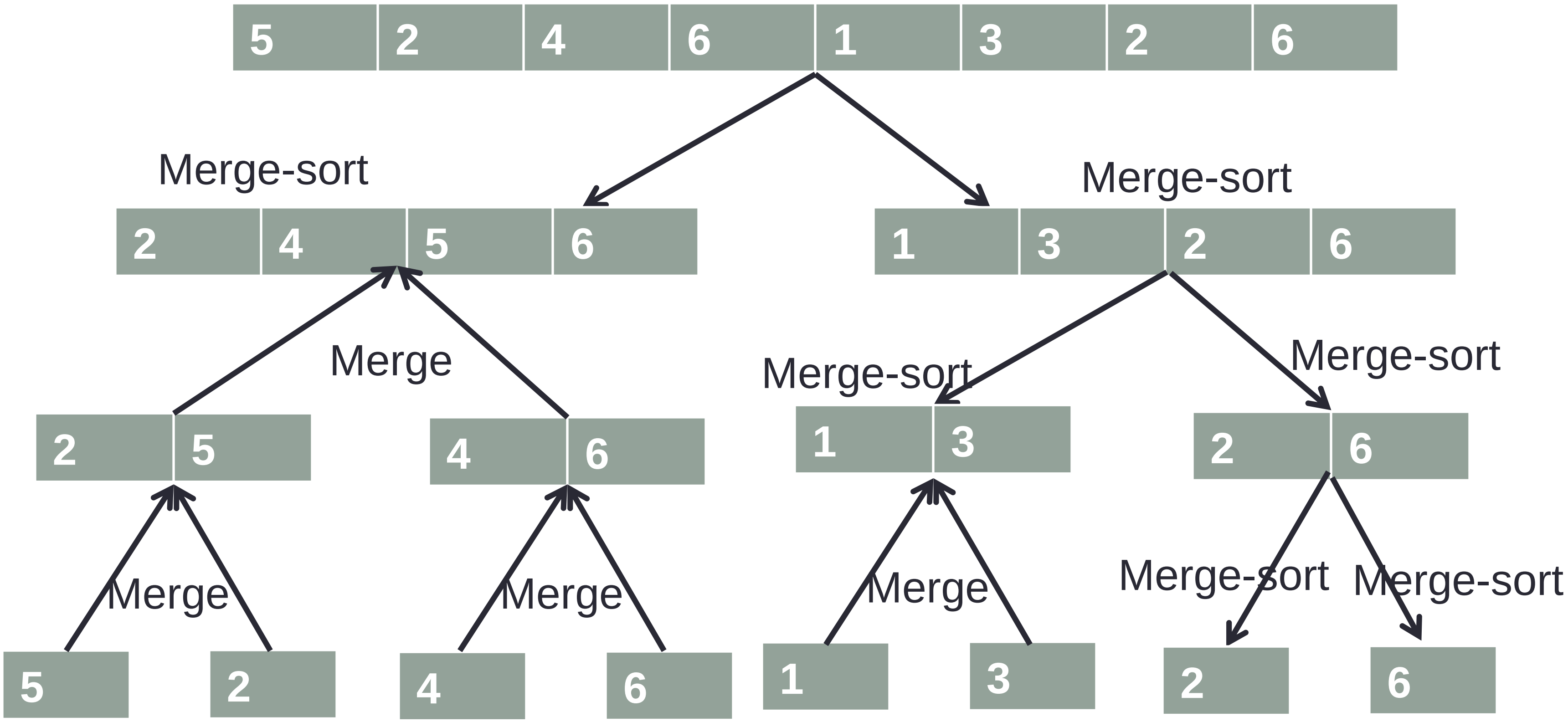
Merge sort example



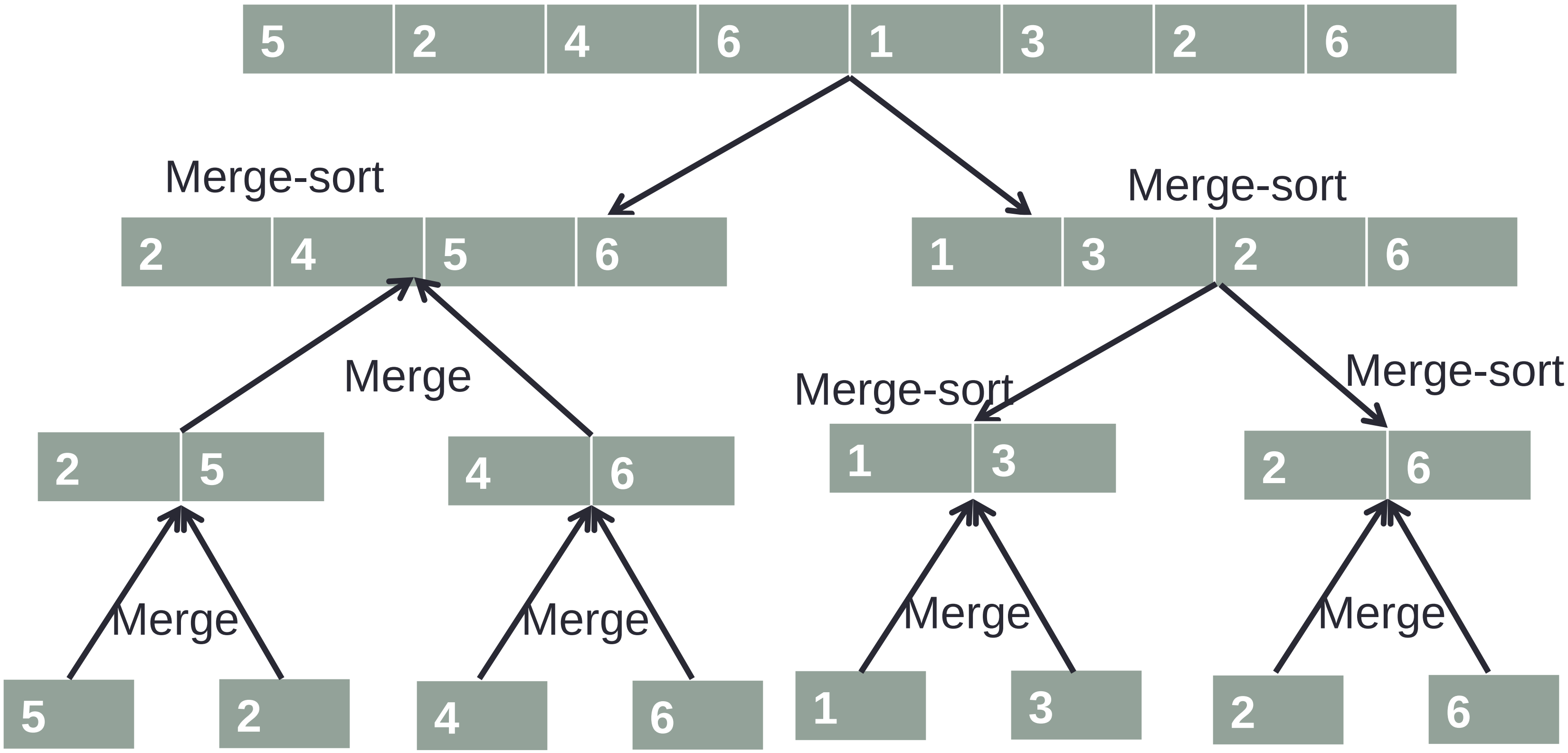
Merge sort example



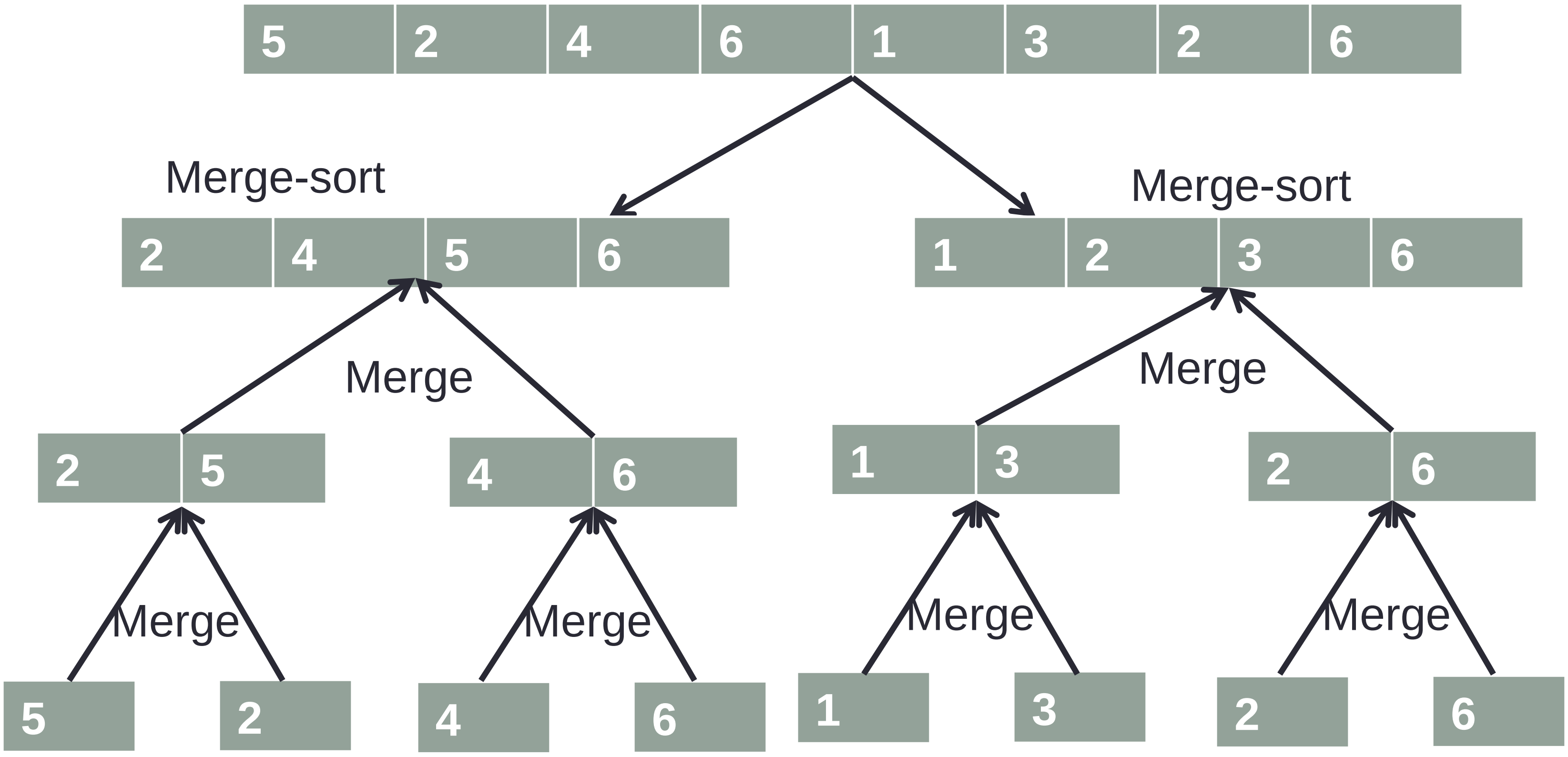
Merge sort example



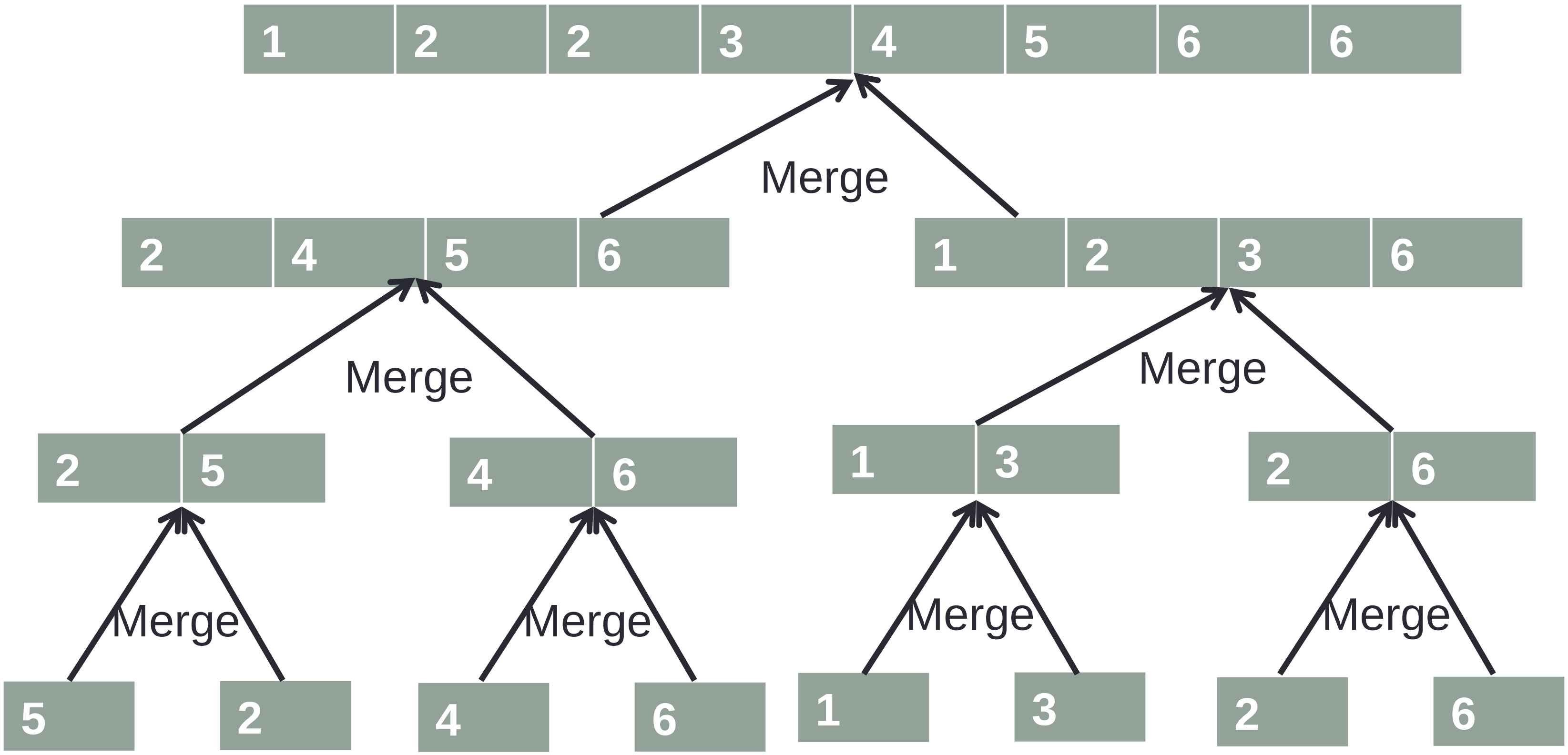
Merge sort example



Merge sort example



Merge sort example



Merge sort detailed pseudocode

Algorithm merge-sort(S, p, r)
{Sorts elements in subarray $S[p..r]$}
    Input: An array, S.
    Input: Index of first, p, element in subarray
    Input: Index of last, r, element in subarray
    Output: The array, S sorted
    if $p < r$ **then**
        $q \leftarrow \lfloor (p + r)/2 \rfloor$
        merge-sort(S, p, q)
        merge-sort(S, $q + 1$, r)
        merge($S[p \dots q]$, $S[(q + 1) \dots r]$, S)

Merge sort exercise

Show the operation of merge sort on the array

{6, 7, 3, 1, 4, 2}

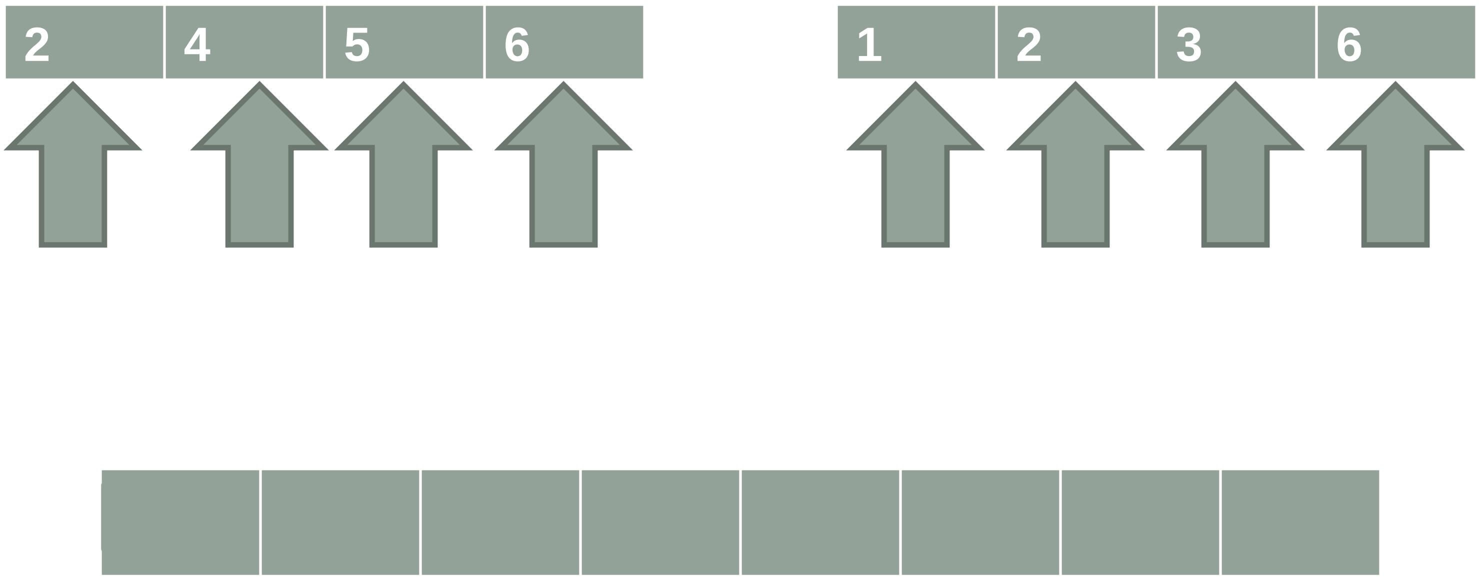
in TWO stages: the divide stage, followed by the merge stage.

Looking for two drawings: one with arrows going down (divide), and another where the arrows point up (merge).

Merge phase pseudocode

Algorithm merge(S_1, S_2, S)
{Merges two sorted sequences into a unique sorted sequence}
Input: Arrays, S_1 and S_2 sorted into non-decreasing order.
Input: An empty array, S
Output: The array, S sorted
while S_1 not empty and S_2 not empty **do**
  if next element in $S_1 \leq$ next element in S_2
    move the next element of S_1 to the next free position in S
  else
    move the next element of S_2 to the next free position in S
while S_1 not empty **do**
  move the remaining elements of S_1 to S
while S_2 not empty **do**
  move the remaining elements of S_2 to S

Merge phase example



Merge sort complexity

- Number of “levels” = $\log_2 n$
- How many comparisons per level? n
- $T(n) = O(n \log_2 n)$

Sorting

- We have covered:
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
- What kinds of characteristics differ between these?
  - Time complexity
  - Whether they sort in-place/need extra storage (merge sort needs extra storage)
- How are they all similar?
  - All comparison sorts
  - There are others which do not fall into this category (e.g. counting sort)

Time complexity exercise 1

```
method1(arr)
        prod = 1;
        indx = 0;
        while indx < size(arr),
                prod = prod * arr(indx);
                indx = indx + 1;
        end
```

Time complexity exercise 2

```
method2(arr)
    prod = 1;
    for i = 1:size(arr),
        for j = 1:size(arr),
            prod = prod * arr(i) * arr(j);
        end
    end
```

Stability in sorting

- In many real applications we need to sort records according to the value of a particular field or *key* (not just lists of numbers)
- E.g. sorting records of personal information, key might be the person's name
- Another example involves sorting exam results. We might initially have a set of records each consisting of a person's name and their mark in the exam. Suppose we want to sort these records according to the exam mark. In this case the exam mark would be the key

Stability: sorting exam results

- What if the records have equal keys (equal exam results)?
- The sorting algorithm may or may not preserve the alphabetical order for equal keys, depending on what algorithm is used
- A sorting method is **stable** if it preserves the relative order of equal keys in the input data

Example

Alphabetic list

Name	Mark
Adams	45
Barker	50
Cousins	78
Jones	50

Stable sort

Name	Mark
Adams	45
Barker	50
Jones	50
Cousins	78

Unstable sort

Name	Mark
Adams	45
Jones	50
Barker	50
Cousins	78

Sorting records

- Most sorting algorithms can be described in terms of two basic operations
  1. Compare two records
  2. Exchange two records
- Comparison of two records involves examining their respective keys
- Exchanging two records can involve a lot of work if the records are very long

Manipulating indices

- Sorting algorithms often sort records by manipulating an array of indices, rather than moving records
- On completion of the sort algorithm the auxiliary index array contains the order of the sorted records

Records to be sorted

45	Adams, John
53	Barrett, Jane
49	Dodgeson, Peter
72	Singleton, Valerie
66	Vicks, David

Index array

Before	After
1	4
2	5
3	2
4	3
5	1