Last Lecture

- We saw how to ...
  - Define the best, worst, and average case running time and space efficiency of
    - insertion sort
    - selection sort
  - Considering the “simple” sorting algorithms, identify the most efficient one and explain why it is the most efficient
Learning Outcomes

- At the end of the next few lectures, a student will be able to:
  - describe the behaviour of and implement simple sorting algorithms:
    - insertion sort
    - selection sort
  - describe the behaviour of and implement more efficient sorting algorithms:
    - quick sort
    - merge sort
  - analyze the best, worst, and average case running time (and space) of these sorting algorithms
Today’s menu

- Looking at
  - Merge Sort
- Analyze its best, worst, and average case running time and space efficiency
How Merge Sort works

- **Divide and conquer** algorithm
- Recursive in nature
- Array has \( n \) elements

1. **Partitioning**: we partition the array until we can’t partition anymore
2. **Sorting**: Then we merge the partitions by sorting their elements until the whole array is sorted

Sorting done by merging elements
How Merge Sort works – Partitioning step

• **Problem statement** -> sort this array:
How Merge Sort works – Sorting step

5 9 7 3 2 11 4
Let’s have a look at Merge Sort - again
Merge Sort Algorithm -> mergeSort

mergeSort(arr, low, high)

if ( low < high )
    mid = (low + high) / 2
    mergeSort(arr, low, mid) // partitioning
    mergeSort(arr, mid + 1, high) // partitioning
    merge(arr, low, mid, mid + 1, high) // sorting
Merge Sort Algorithm -> merge

merge(arr, low, mid, mid + 1, high)

initialize indexes
create temp array of size high - low + 1
while both subarrays contain unmerged elements
    if subarray1’s current element is less than subarray2’s
        insert subarray1’s current element in temp
        increment subarray1 and temp’s indexes
    else
        insert subarray2’s current element in temp
        increment subarray2 and temp’s indexes
while subarray1 contains unmerged items
    insert subarray1’s current element in temp
    increment subarray1 and temp’s indexes
while subarray2 contains unmerged items
    insert subarray2’s current element in temp
    increment subarray2 and temp’s indexes
copy temp back to the original (sub)array low ... high
Each time we “mergeSort” …
- We divide in half until the partitions have size 1
- How many times does \( n \) have to be divided in half before the result is 1?
- Answer: \( \log_2(n) \) times

When we “merge” …
- We merge \( n \) elements at each level
- \( n - 1 \) comparisons are made at each level
- Merge sort performs around \( n \times \log_2(n) \) operations

For example:
```
8
  4  4
  2  2  2  2
1 1 1 1 1 1 1 1
```

3
When we “merge” …

Section 1  1 12 22 99  
Section 2  3 5 23 42  

output

- We have \( n \) elements
- \( \textbf{Merge} \) compares ______ times
Time Efficiency Analysis of Merge Sort - 2

- Does the organization of the data in the array to be sorted affect the amount of work done by merge sort?

- Time efficiency of
  - Best case:
  - Average case:
  - Worst case:
Space Efficiency Analysis of Merge Sort

- Not in-place algorithm

- How much space (memory) does merge sort require to execute?

- Therefore, its space efficiency is -
Is merge sort stable?
Learning Check

- We can now ...
  - Define the best, worst, and average case running time and space efficiency of
    - merge sort
Next Lectures

- Another efficient sorting algorithm -> QuickSort