MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

30 Sorting

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

April 6, 2022

Admin

- I will have office hour today at 15:00
  - Virtual and in-person at ITB-159

Overview

- Sorting: What we have seen so far?

Sorting Algorithm	Time Complexity	Properties
Insertion sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Selection sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Heap sort	O(nlogn)	• fast • in-place • Suitable for large datasets (1K - 1M)

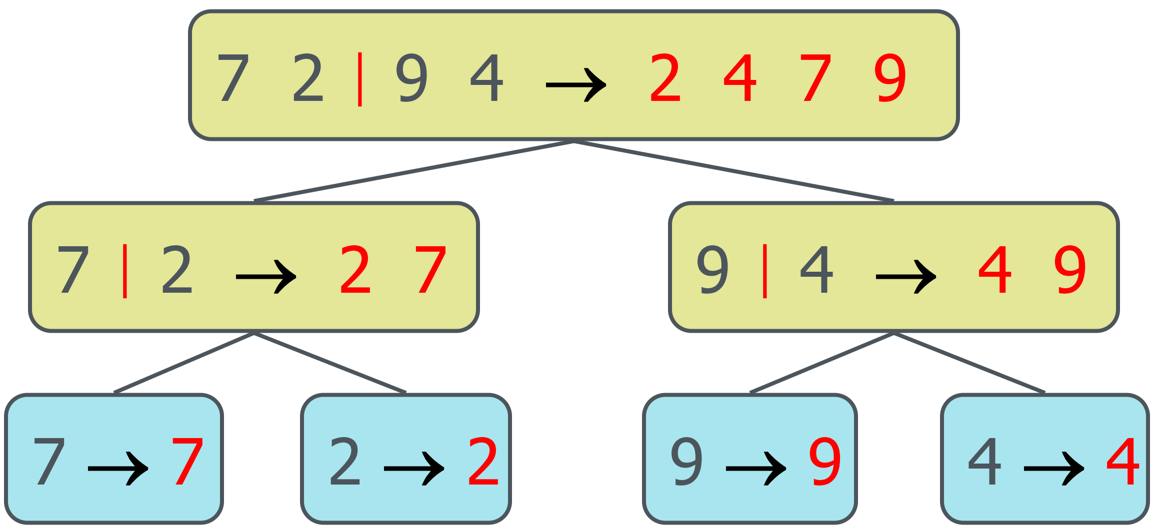
- We will talk about Merge sort and Quick sort
  - Both are very fast algorithms
  - are suitable for very large dataset (>1M)

Merge sort

- **Merge sort** is a sorting algorithm based on the divide-and-conquer paradigm
- Like Heap sort
  - It uses a **comparator**
  - It has $O(n \log n)$ running time (we will discuss this in more detail)
- Unlike Heap sort
  - It does **not** use an **auxiliary priority queue**
  - It accesses data in a sequential manner (suitable to sort data on a disk)
    - It is **not in-place**.
      - There are methods to implement it as an in-place sorting, but those methods are not in the scope of this course.

Merge sort - A quick overview

- An execution of merge-sort is depicted by a **binary tree**
  - each node represents a **recursive** call of merge-sort and shows
    - unsorted sequence before the execution and how we partition it
    - sorted sequence at the end of the execution
  - the **root** is the **initial** call
  - the **leaves** are calls on subsequences of size **0** or **1**



Divide-and-Conquer

- **Divide-and conquer** is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1

- Merge sort is a sorting algorithm based on the **divide-and-conquer** paradigm

Merge sort - Formal Algorithm

- **Merge sort** on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1, C)$

$mergeSort(S_2, C)$

$S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences (Conquer)

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time
- Similar to the addition of polynomials in Assignment 2

Algorithm *merge*(A, B)

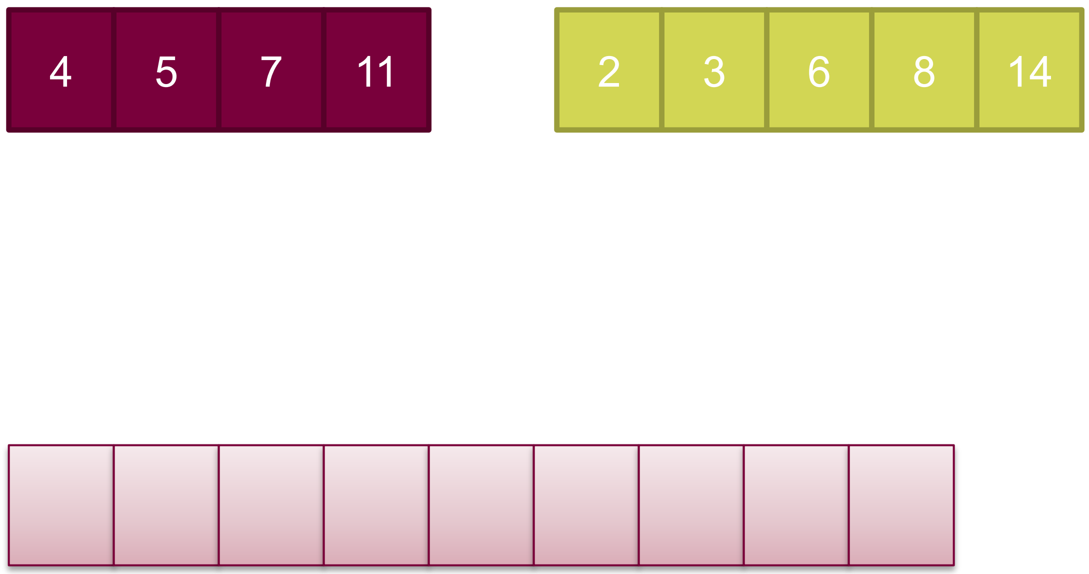
Input sequences A and B with $n/2$ elements each

Output sorted sequence of $A \cup B$

```
S ← empty sequence
while ¬A.empty() ∧ ¬B.empty()
    if A.front() < B.front()
        S.addBack(A.front()); A.eraseFront();
    else
        S.addBack(B.front()); B.eraseFront();
while ¬A.empty()
    S.addBack(A.front()); A.eraseFront();
while ¬B.empty()
    S.addBack(B.front()); B.eraseFront();
return S
```

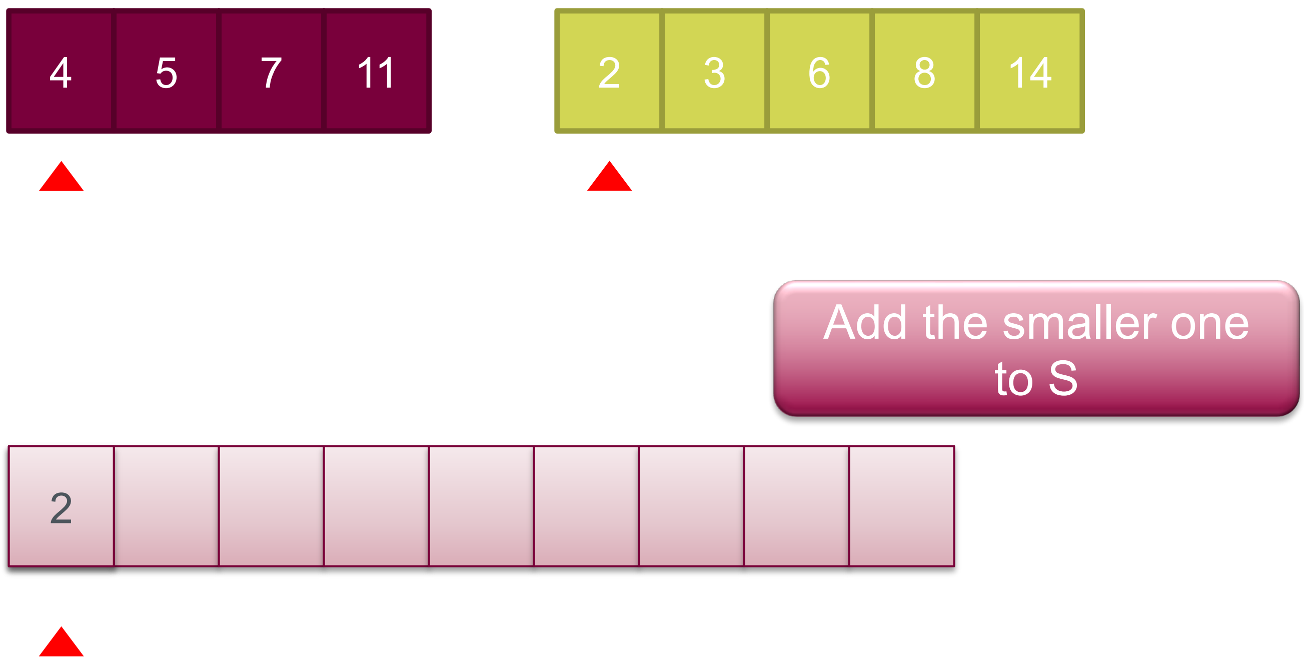
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



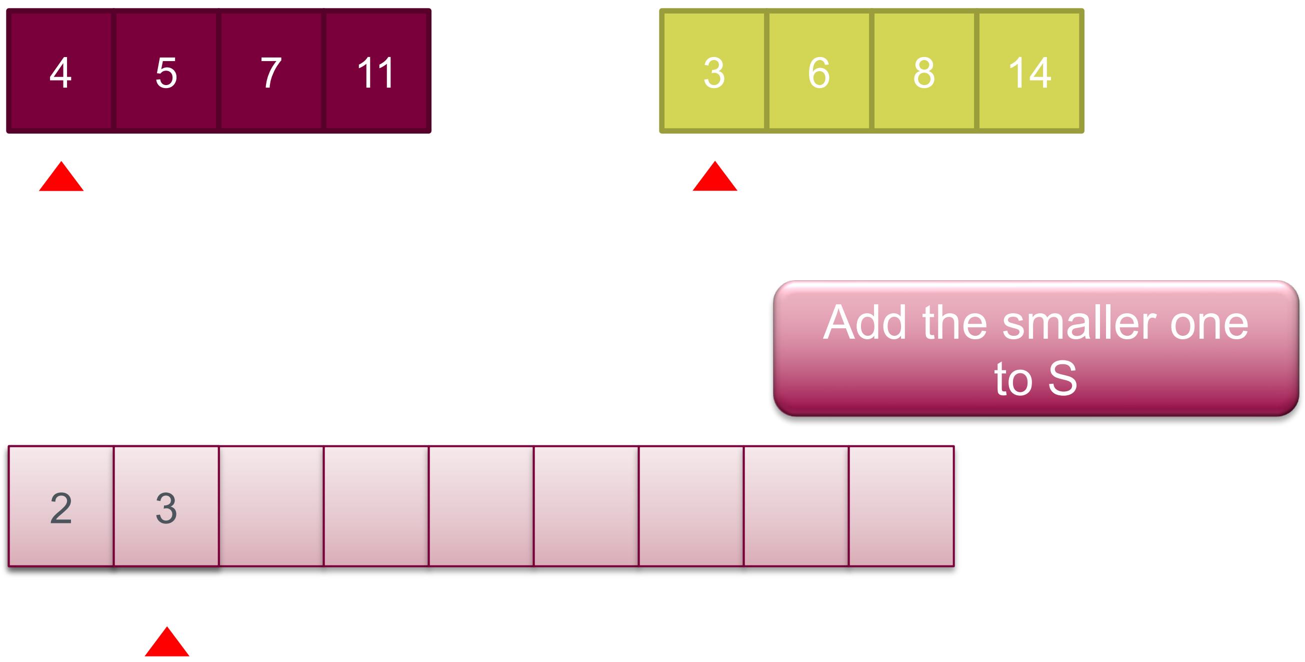
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



Merging Two Sorted Sequences (Conquer)

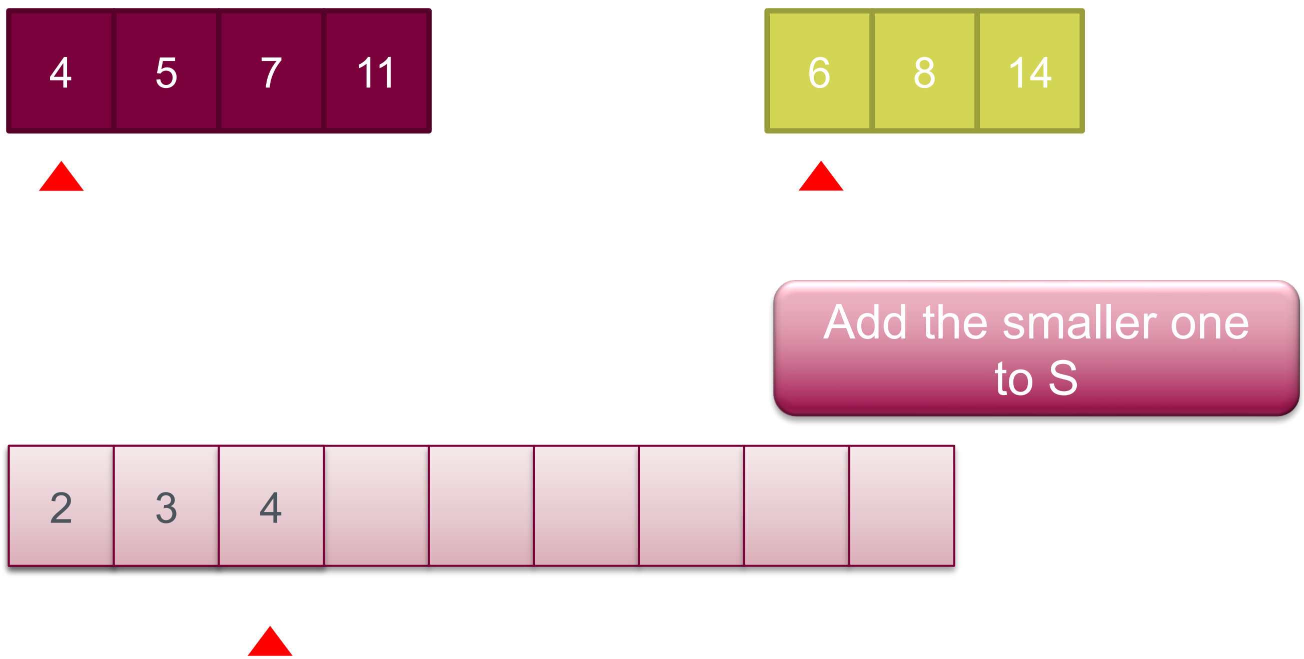
- How we merge two sorted sequences



McMaster University

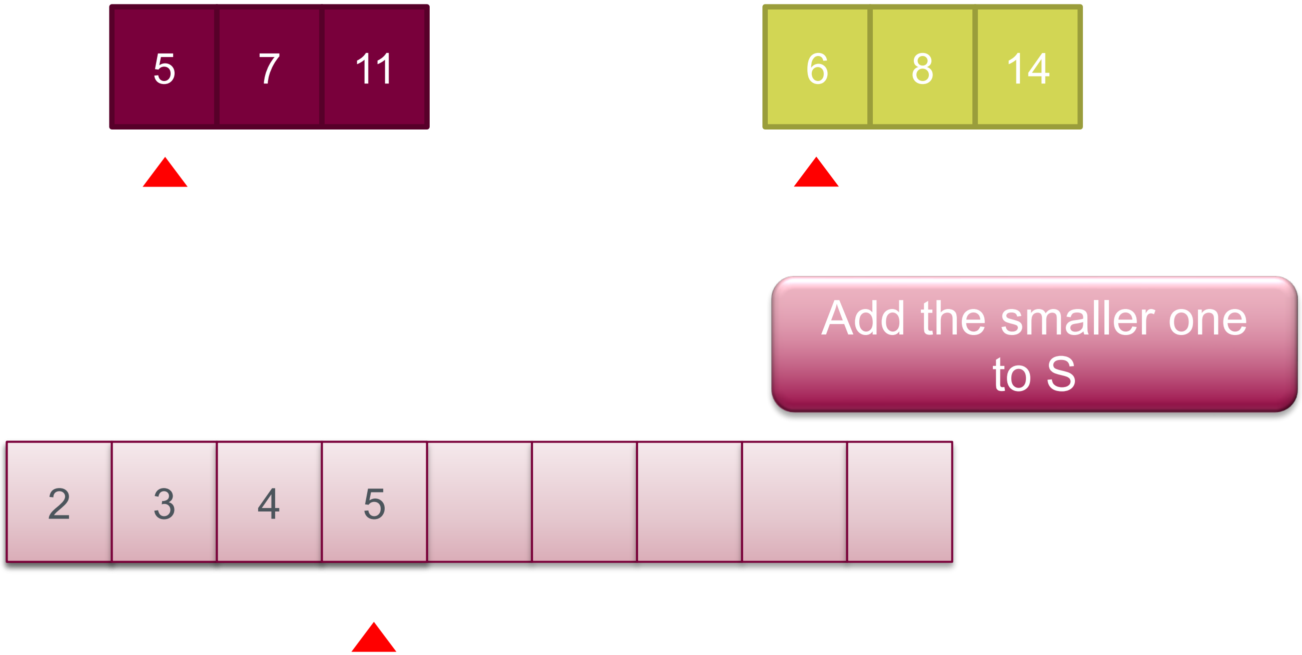
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



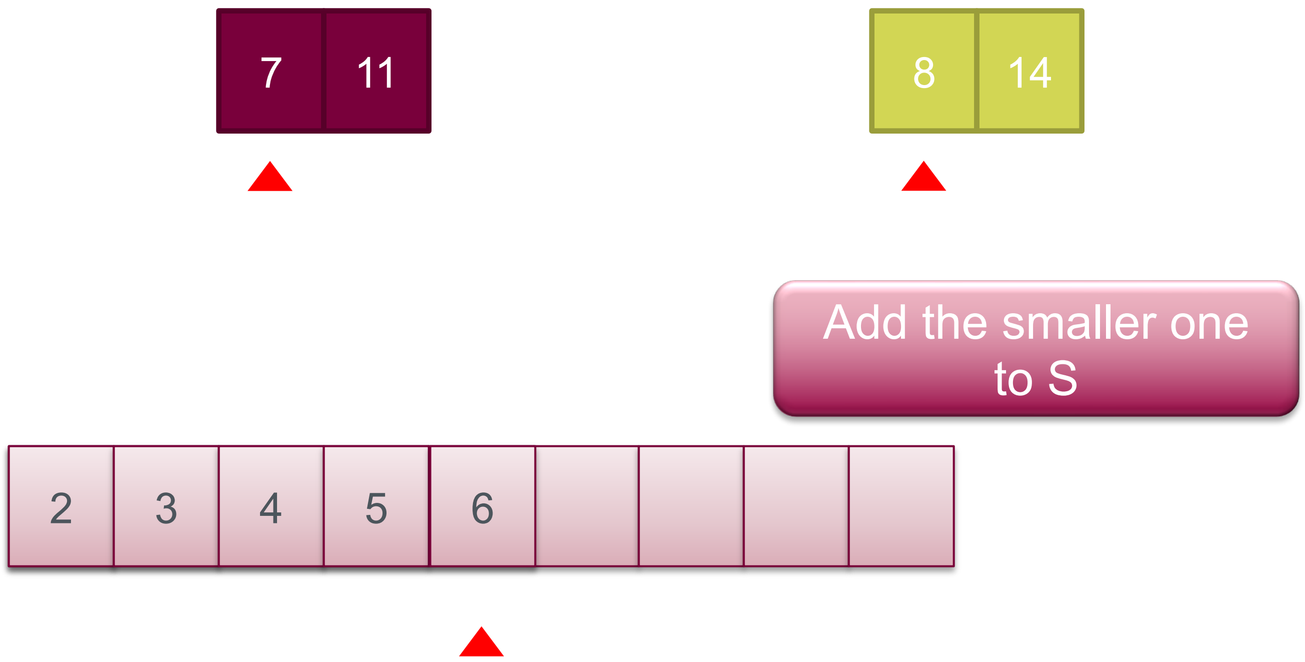
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



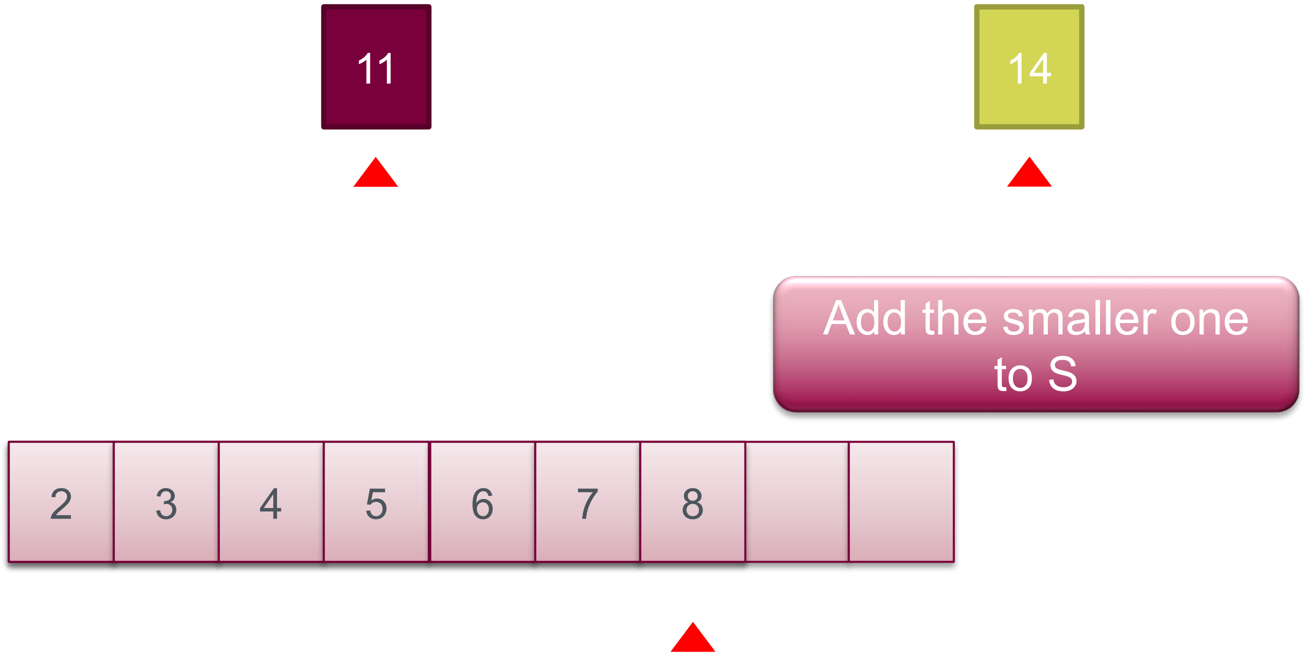
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



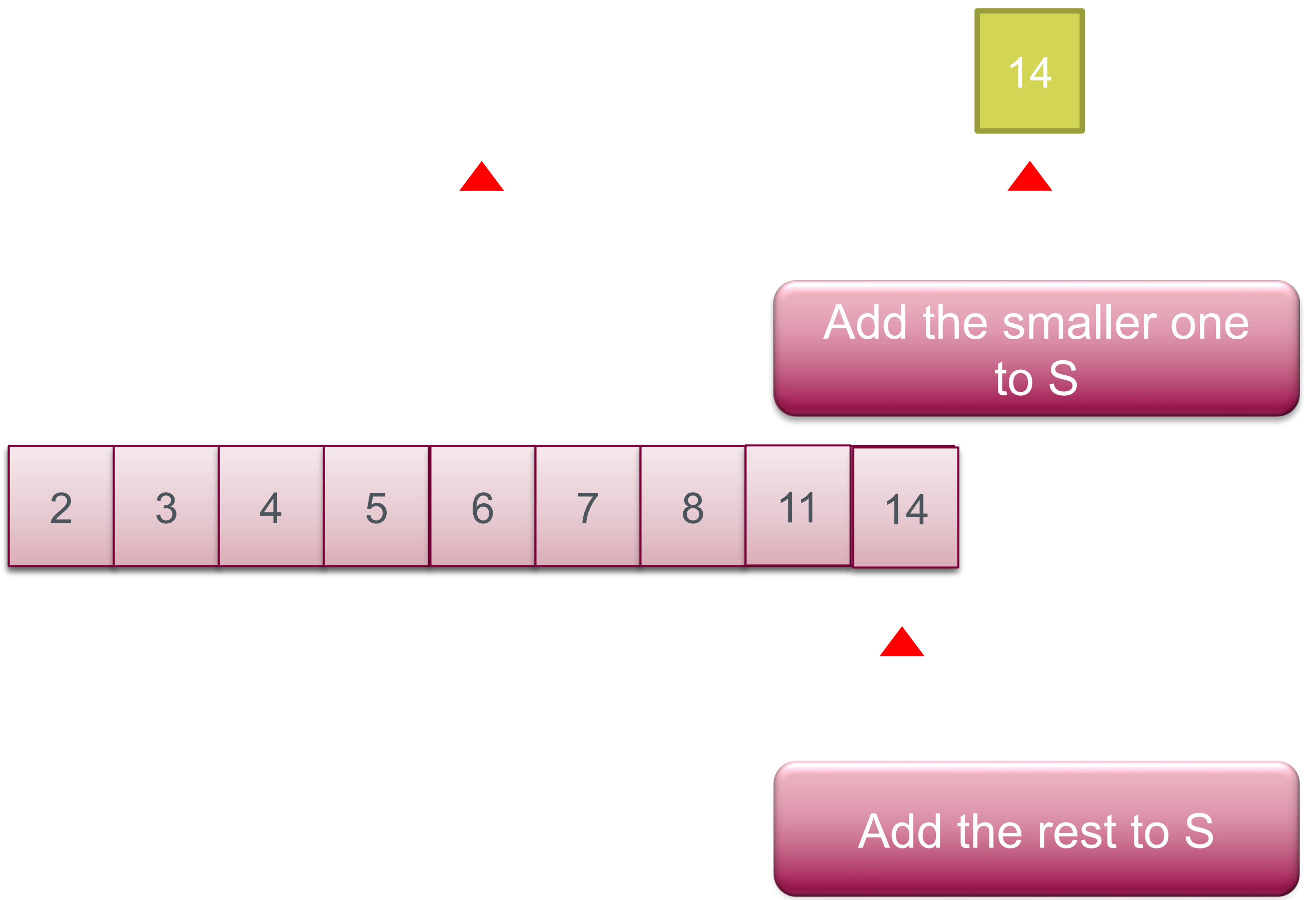
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



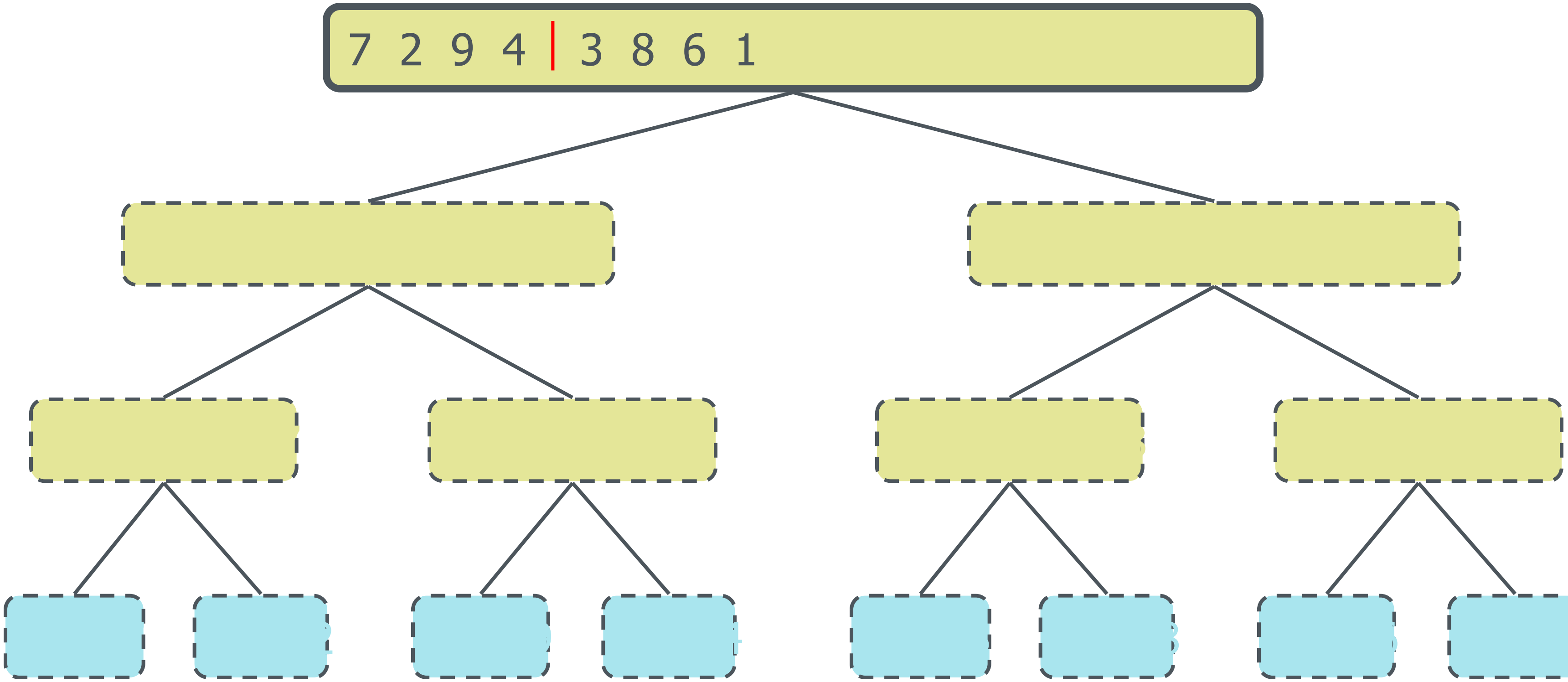
Merging Two Sorted Sequences (Conquer)

- How we merge two sorted sequences



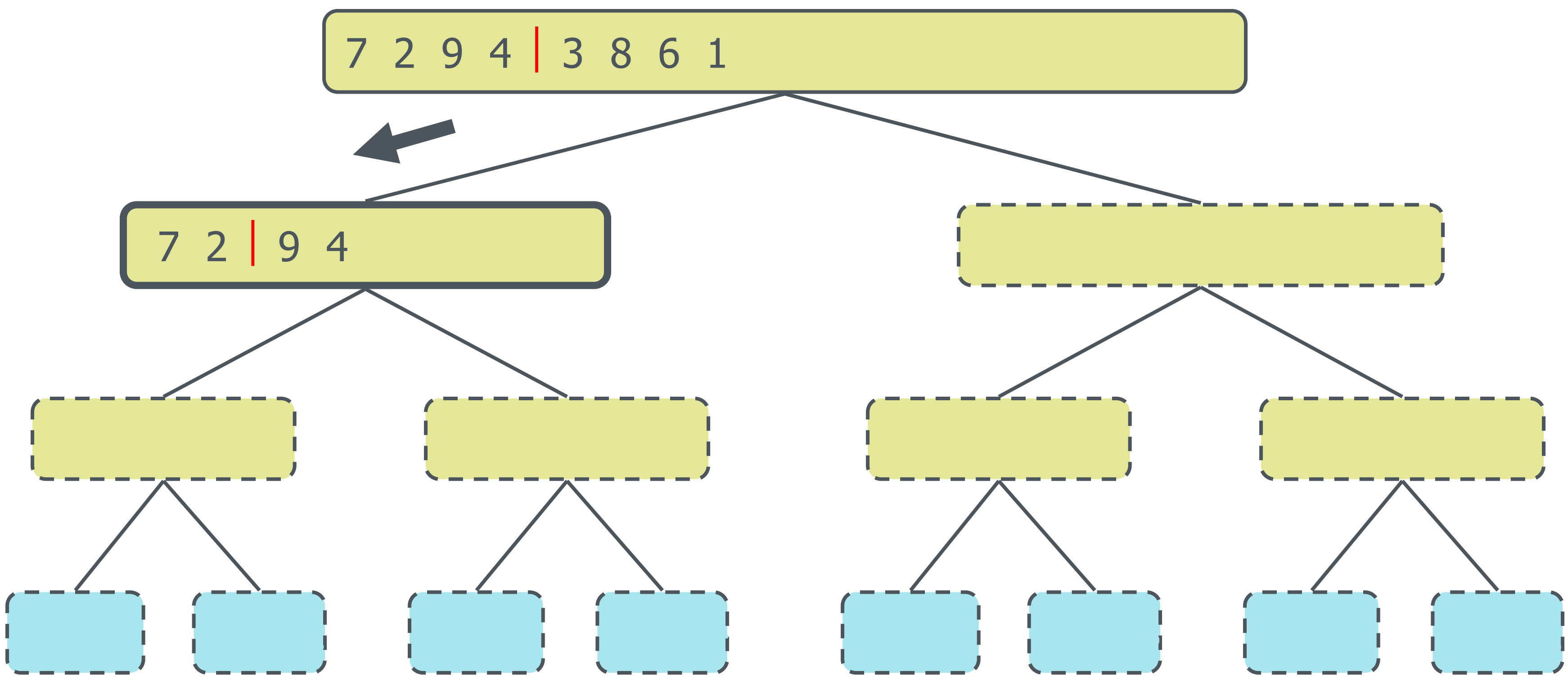
Execution Example

- **Partition**



Execution Example

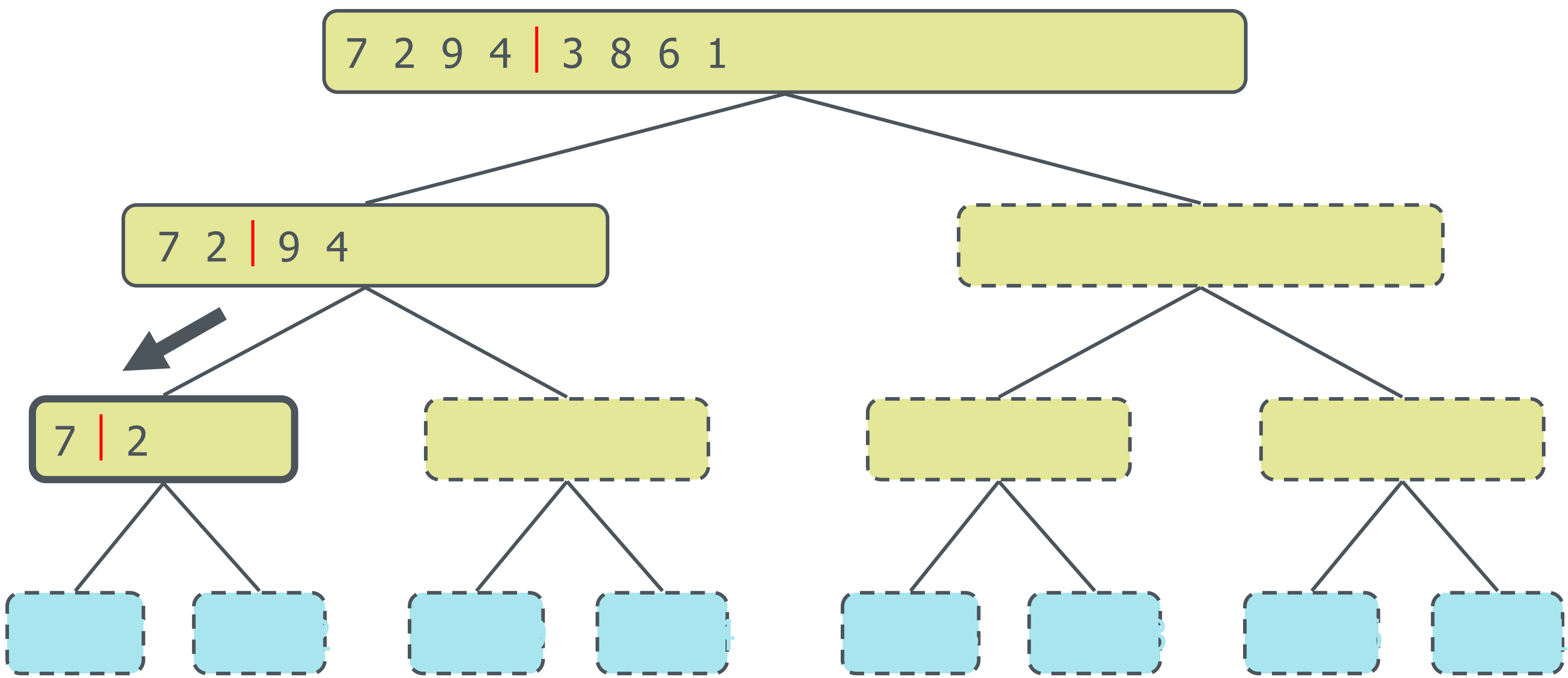
- **Recursive call, partition**



McMaster University

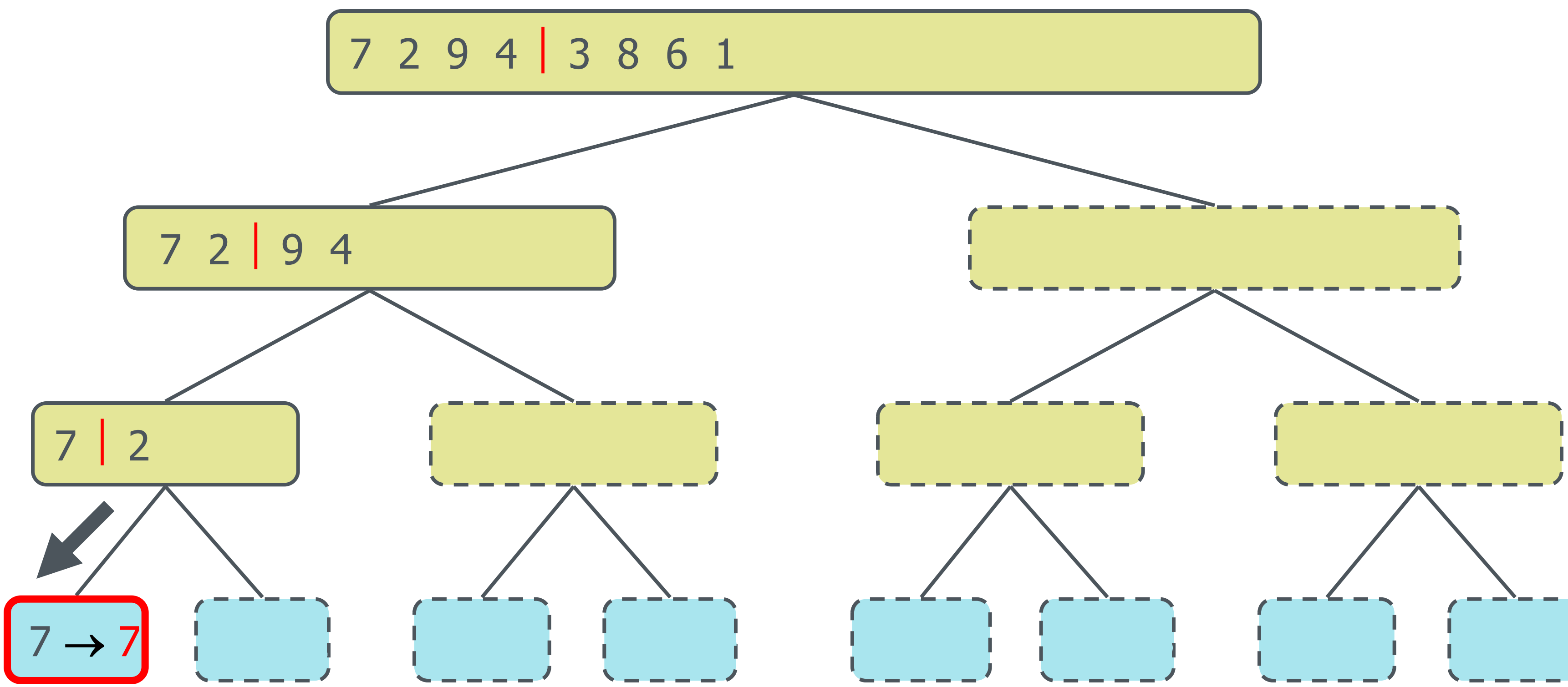
Execution Example

- **Recursive call, partition**



Execution Example

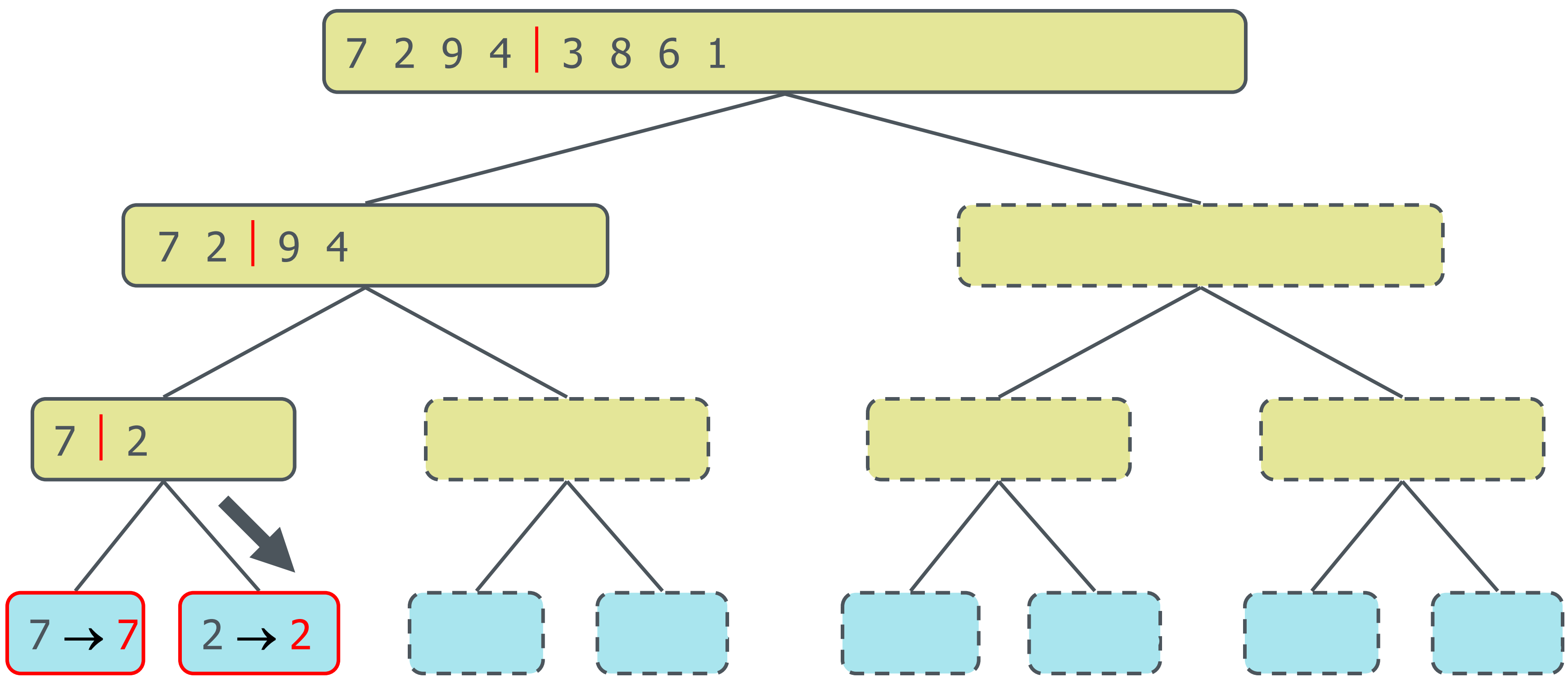
- **Recursive call, base case**



McMaster University

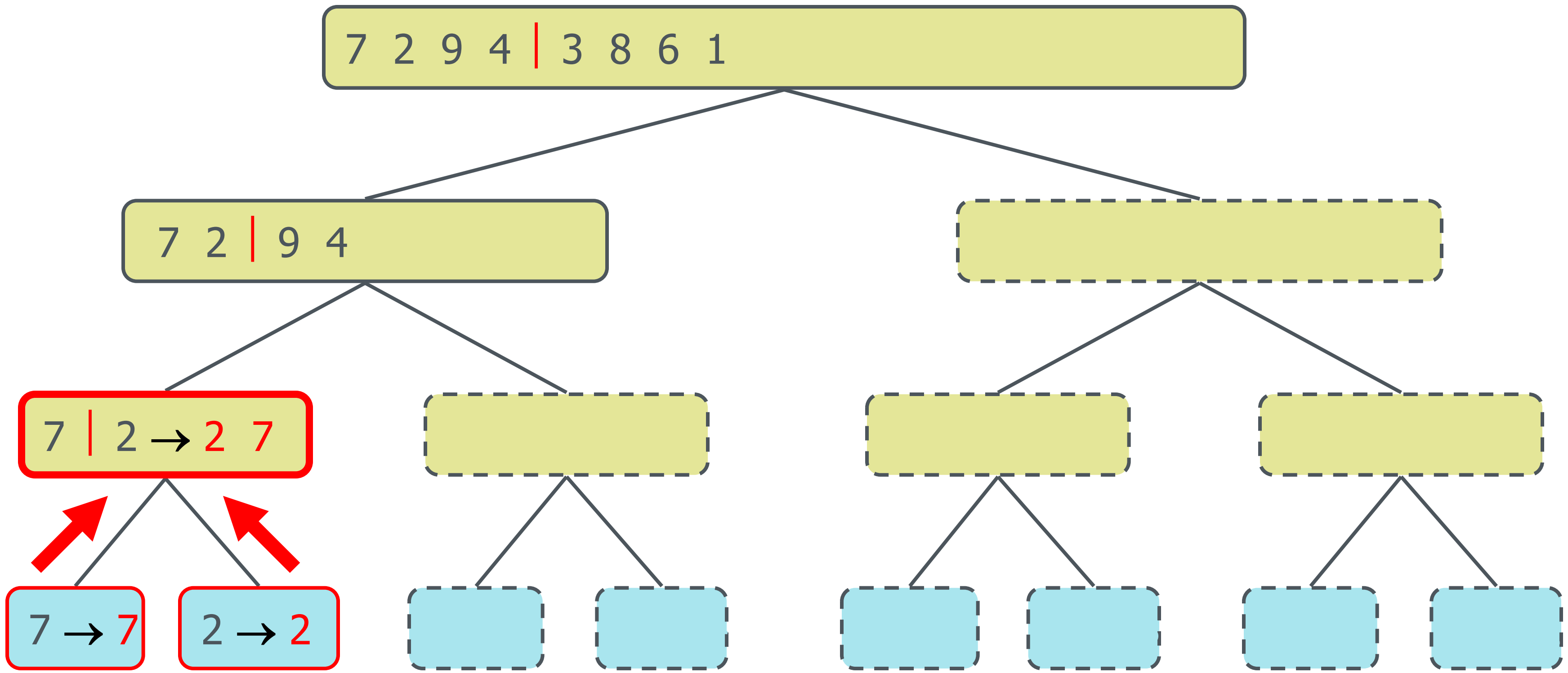
Execution Example

- **Recursive call, base case**



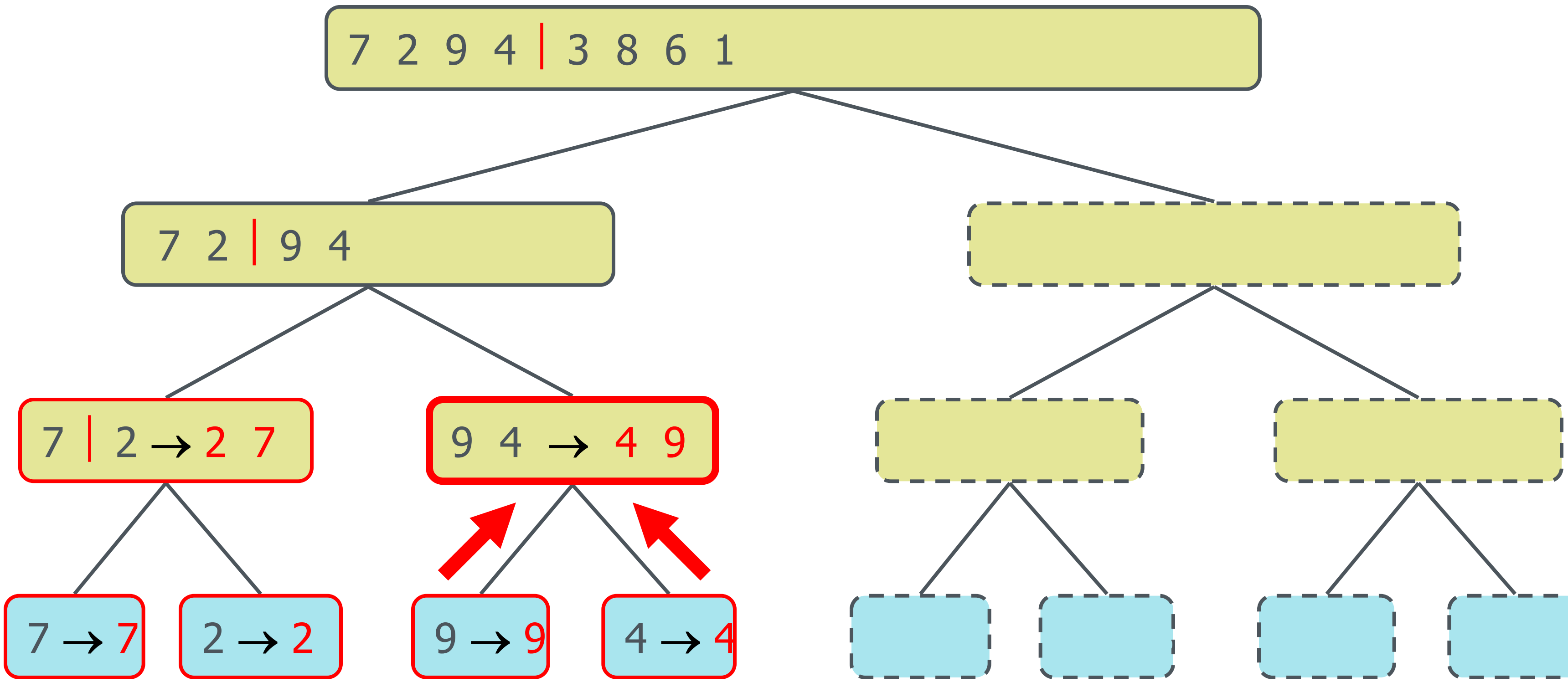
Execution Example

- **merge**



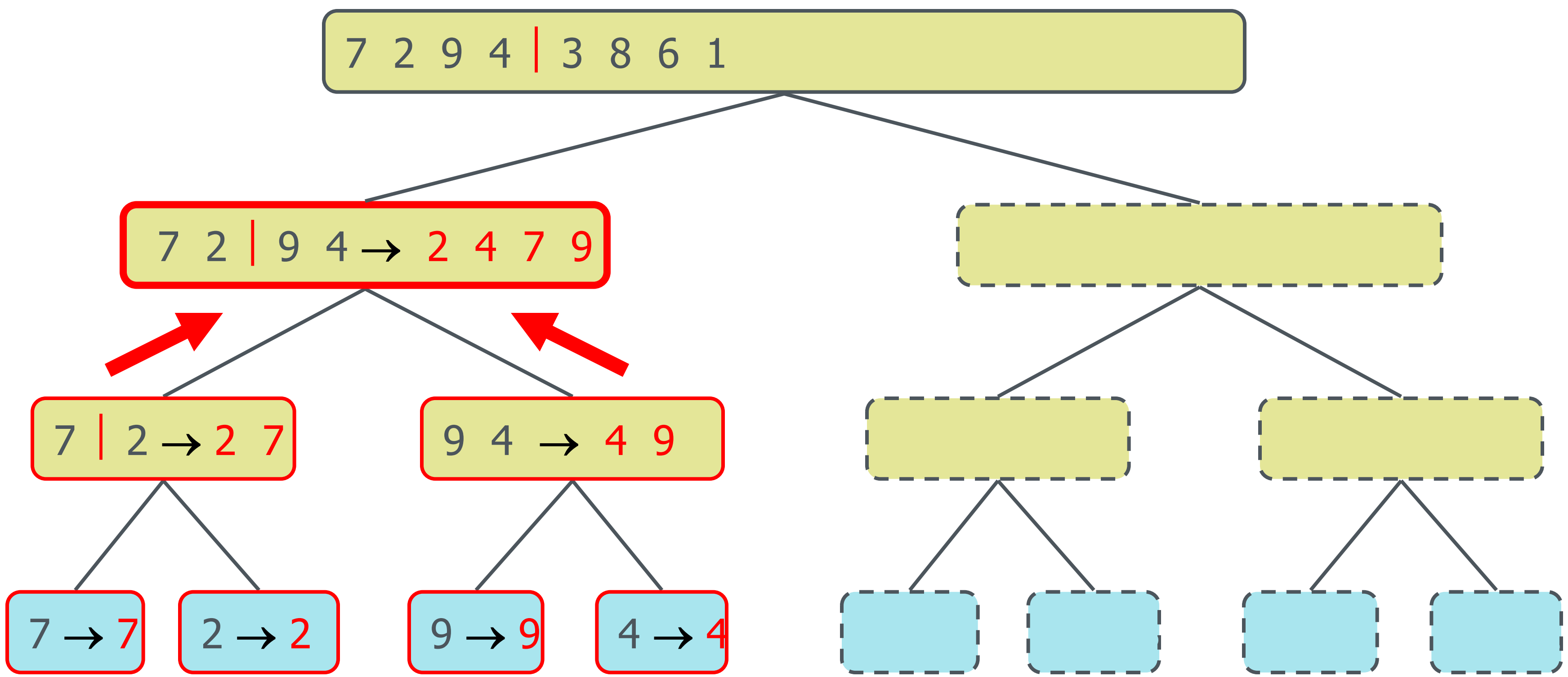
Execution Example

- **Recursive call, …, base case, merge**



Execution Example

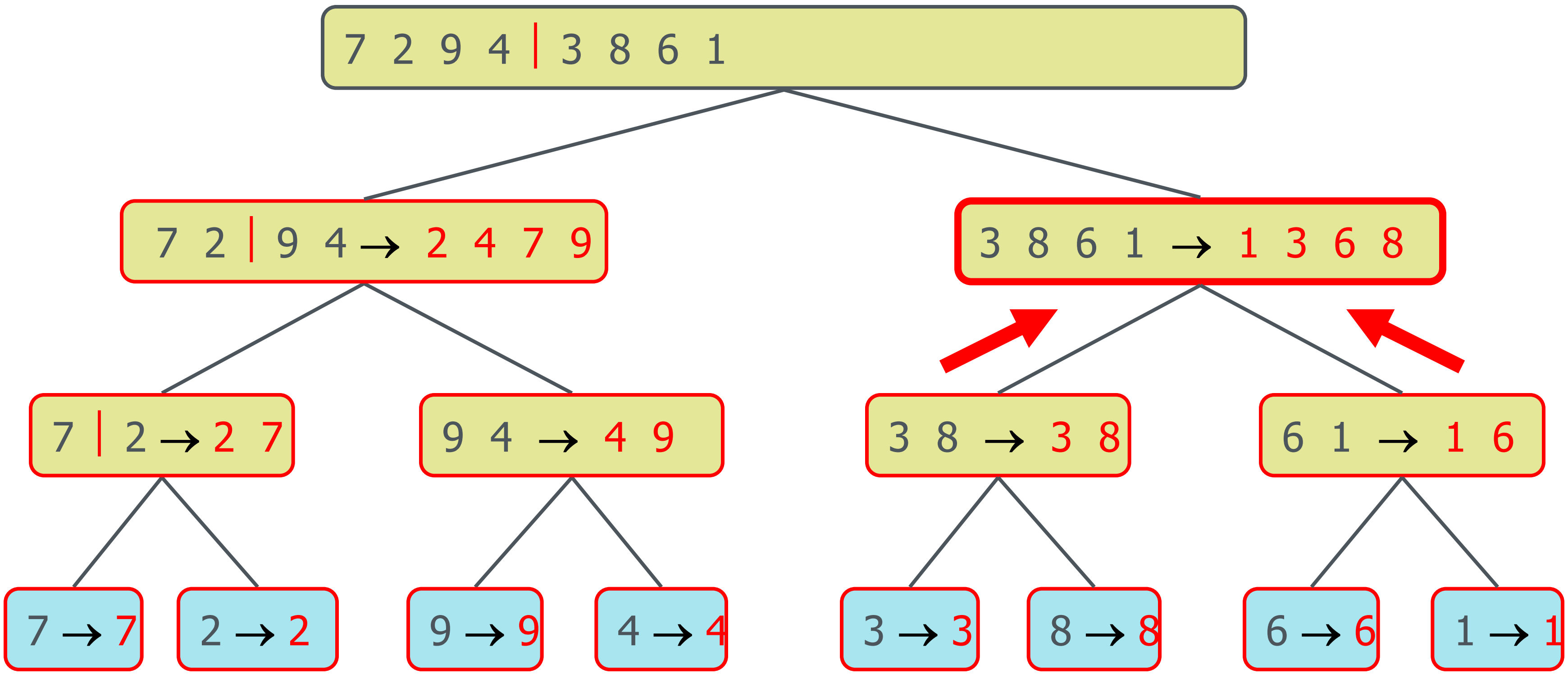
- **merge**



McMaster University

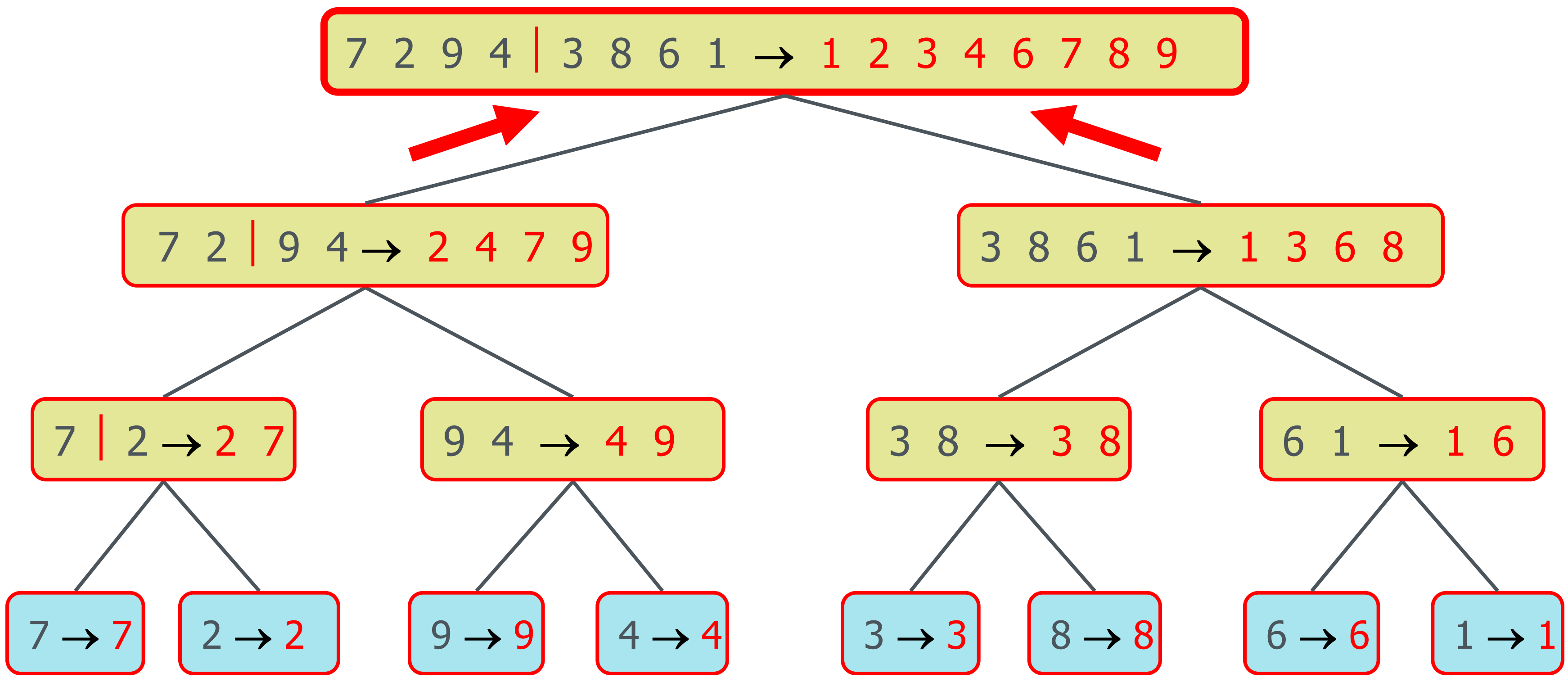
Execution Example

- **Recursive call, …, merge, merge**



Execution Example

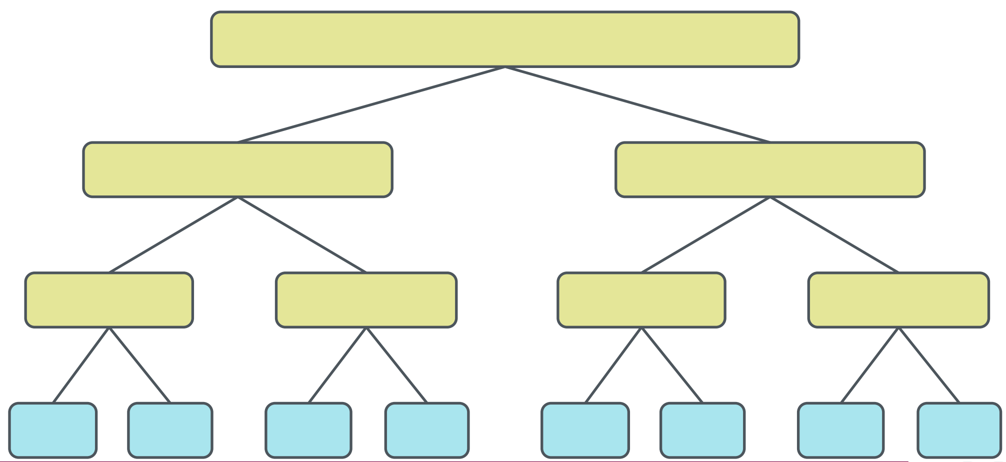
- **merge**



Analysis of Merge-Sort

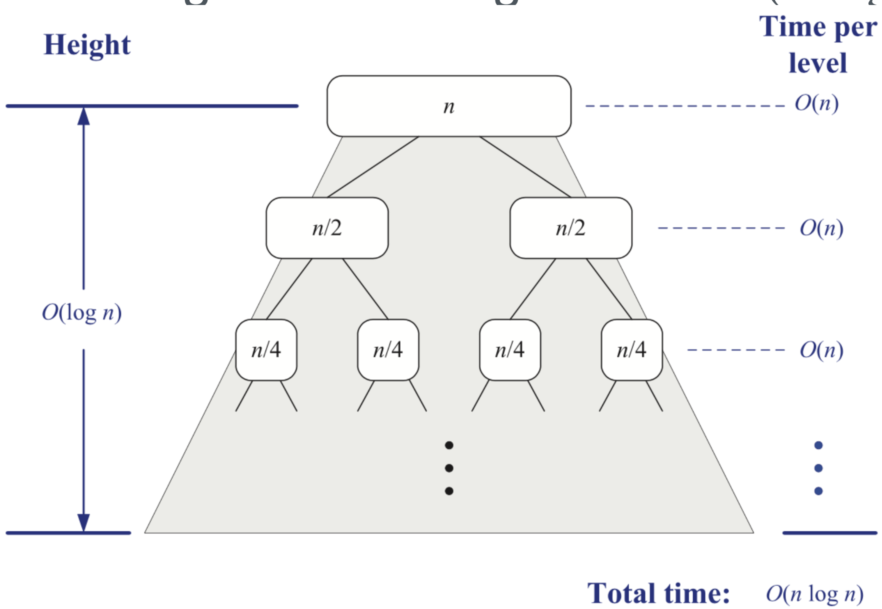
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Analysis of Merge-Sort

- The height $\boldsymbol{h}$ of the merge-sort tree is $\boldsymbol{O}(\log \boldsymbol{n})$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth $\boldsymbol{i}$ is $\boldsymbol{O}(\boldsymbol{n})$
  - we partition and merge 2^i sequences of size $\boldsymbol{n}/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$



Analysis of Merge-Sort - Using Recurrence Relations

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Analysis of Merge-Sort - Using Recurrence Relations

- Iterative Substitution:
- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}
$$

- Note that base, T(n)=b, case occurs when 2^i=n. That is, i = log n.
- So, $$T(n) = bn + bn\log n$$
- Thus, T(n) is O(n log n).

Recall

Sorting Algorithm	Time Complexity	Properties
Insertion sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Selection sort	$O(n^2)$	• slow • in-place • Suitable for small datasets (< 1K)
Heap sort	O(nlogn)	• fast • in-place • Suitable for large datasets (1K - 1M)
Merge sort	O(nlogn)	• fast • sequential data access • Suitable for for huge datasets (>1M)

Questions?

Please evaluate this course!
https://evals.mcmaster.ca/
Thank you