Algorithms and Data Structures

5th Lecture: Divide and Conquer

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
S. Semba, Deepika Saxena, Yinghu Zhou, Akila Siriweera

University of Aizu

Last Updated: 2024/12/16

Outline

- Recursion
- Divide and Conquer Approach
- Merge Sort

Recursion

- A recursive function is a function which calls itself recursively.
- The recursive function should be terminated by a specified condition.

Recursive Function: An Example

```
toBinary(x)
    if x > 0
        toBinary(x/2)
        print x%2


call toBinary(input)
```

toBinary(1)	$\rightarrow$	1
toBinary(2)	$\rightarrow$	0
toBinary(5)	$\rightarrow$	1
toBinary(10)	$\rightarrow$	0
toBinary(21)	$\rightarrow$	1
toBinary(43)	$\rightarrow$	1
toBinary(86)	$\rightarrow$	0

Input	**Output**
$86_{(10)}$	$1010110_{(2)}$

Divide and Conquer Approach

- Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.
- These algorithms typically follow a divide and conquer approach:

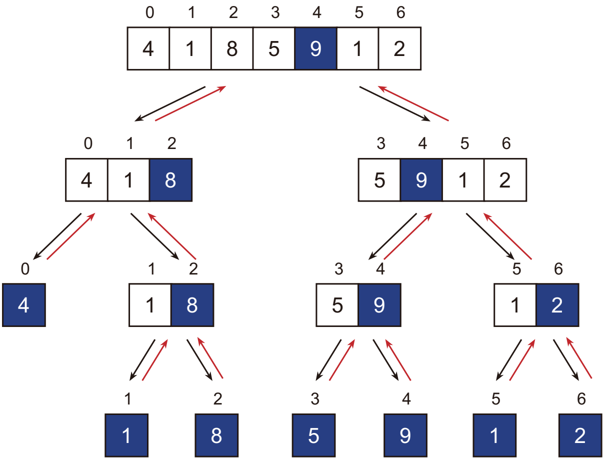
1. **[Divide]** they break the problem into several subproblems that are similar to the original problem but smaller in size,
2. **[Conquer]** solve the subproblems recursively, and then
3. **[Combine]** combine these solutions to create a solution to the original problem.

Divide and Conquer Algorithm: An Example

- Recursive divide and conquer solution for finding the maximum.

```
findMax(A, left, right)
    m = (left + right)/2
    if left == right-1
        return A[left]
    else
        u = findMax(A, left, m)
        v = findMax(A, m, right)
        if u > v
            return u
        else
            return v
```

Finding the Maximum



Analysis

```
findMax(A, left, right)
    m = (left + right)/2                      const
    if left == right-1                        const
        return A[left]                        const
    else
        u = findMax(A, left, m)               T(n/2)
        v = findMax(A, m, right)              T(n/2)
        if u > v                              const
            return u                          const
        else
            return v                          const
```

Recurrence

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.

Analysis of Finding the Maximum

- We set up recurrence for Finding the Maximum.
  1. **[Divide]** The divide step computes the middle of subarray, which takes constant time. Thus, $O(1)$.
  2. **[Conquer]** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
  3. **[Combine]** The comparison between two maximum values in two n-element subarrays takes time $O(1)$.
- The recurrence for the worst-case running time $T(n)$ of Finding the Maximum is

$$T(n) = \begin{cases} c & (n = 1) \\ 2T(n/2) + c & (n > 1) \end{cases}$$

Solving Recurrence

- The algorithm takes linear time.

$$
\begin{aligned}
T(n) &= 2T(n/2) + c \\
&= 2(2T(n/4) + c) + c \\
&= 2(2(2T(n/8) + c) + c) + c \\
\ldots \\
&= 2(2(\ldots(2T(1) + c)\ldots) + c) + c \\
&= 2(2(\ldots(2c + c)\ldots) + c) + c \\
&= (2n - 1)c
\end{aligned}
$$

Merge Sort

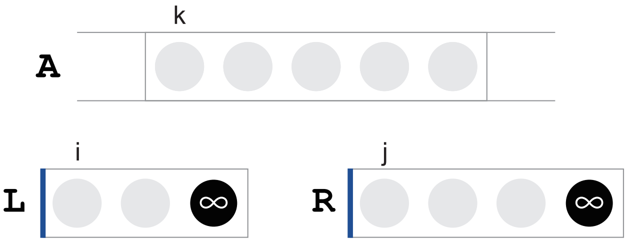
- The Merge Sort algorithm closely follows the divide and conquer paradigm. Intuitively, it operates as follows.
  1. **[Divide]** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
  2. **[Conquer]** Sort the two subsequences recursively using Merge Sort.
  3. **[Combine]** Merge the two sorted subsequences to produce the sorted answer.
- The key operation of the Merge Sort algorithm is the merging of two sorted sequences in the **[Combine]** step.

Merge Sort: Merge

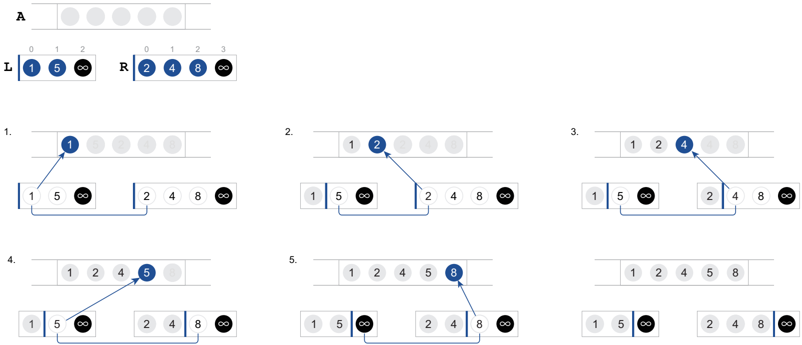
```
merge(A, l, m, r)
    n1 = m - l
    n2 = r - m
    create an array L[0..n1]
    create an array R[0..n2]
    for i = 0 to n1-1
        L[i] = A[l+i]
    for j = 0 to n2-1
        R[j] = A[m+j]
    L[n1] = SENTINEL
    R[n2] = SENTINEL
```

```
i = 0
j = 0
for k = l to r-1
    if L[i] < R[j]
        A[k] = L[i]
        i = i+1
    else
        A[k] = R[j]
        j = j+1
```

Variables for Merge

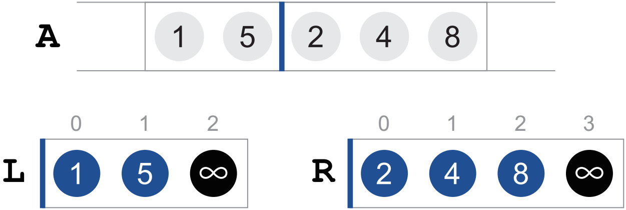


Merge



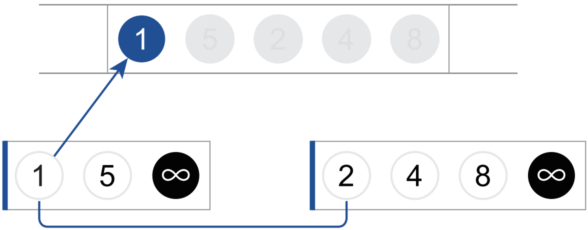
Y. Watanobe, J. Huang, Y. Pei, W. Chen, S. Semba, Y. Zhou, D. Saxena, A. Siriweera University of Aizu

Merge (0)



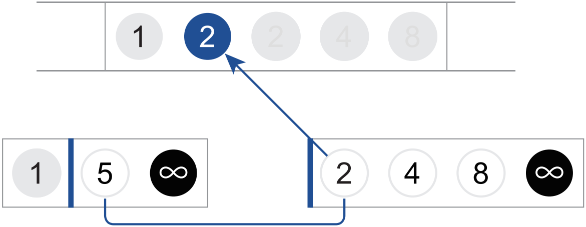
Merge (1)

1.



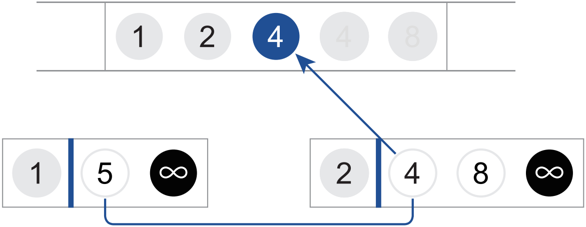
Merge (2)

2.



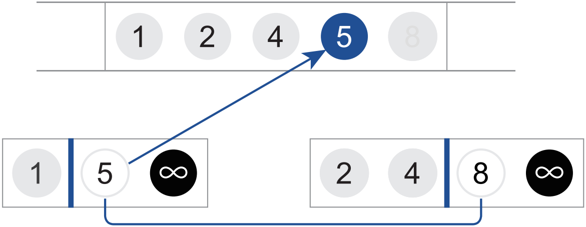
Merge (3)

3.



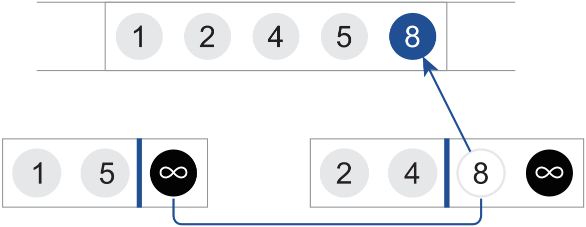
Merge (4)

4.

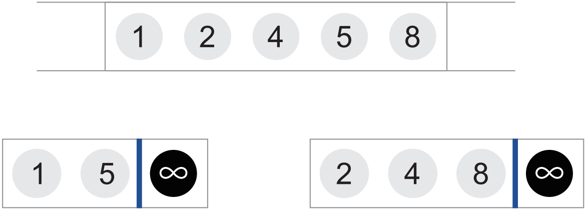


Merge (5)

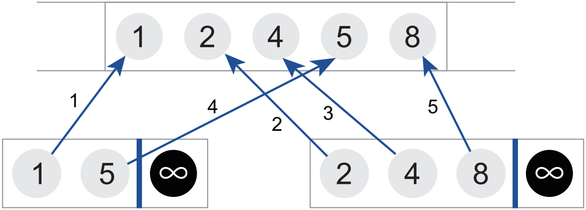
5.



Merge (6)



Merge (7)

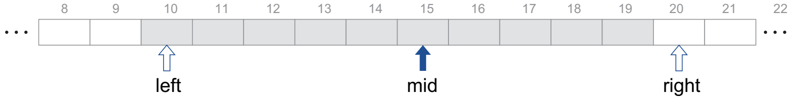


Merge Sort

```
mergeSort(A, left, right)
    if left + 1 < right
        mid = (left + right)/2
        mergeSort(A, left, mid)
        mergeSort(A, mid, right)
        merge(A, left, mid, right)
```

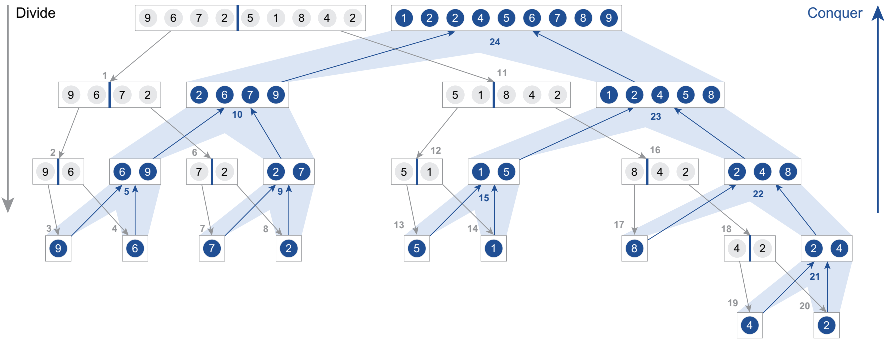
- To sort the entire sequence $A = (A[0], A[1], ..., A[n-1])$, we make the initial call `mergeSort(A, 0, n)`.

Variables for the Merge Sort Algorithm



The target of sorting is managed by a half-open interval [*left*, *right*).

Merge Sort



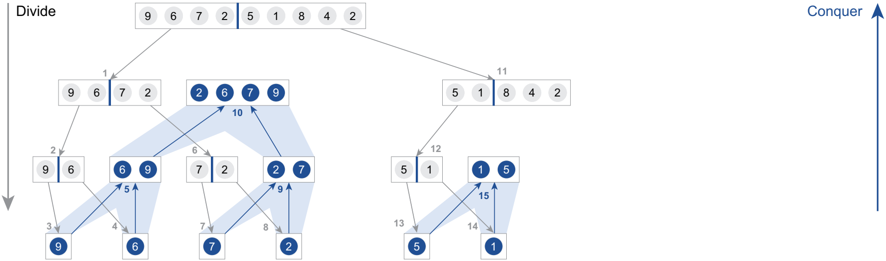
- The operation of Merge Sort on the array $A = \{9, 6, 7, 2, 5, 1, 8, 4, 2\}$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Merge Sort (1-5)

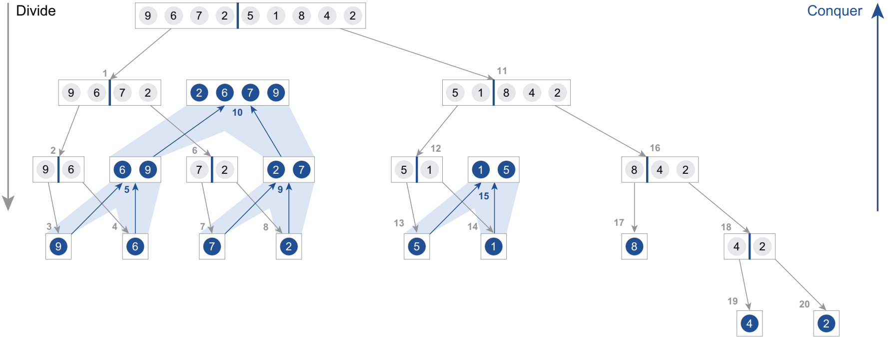
Merge Sort (6-8)

Merge Sort (9-10)

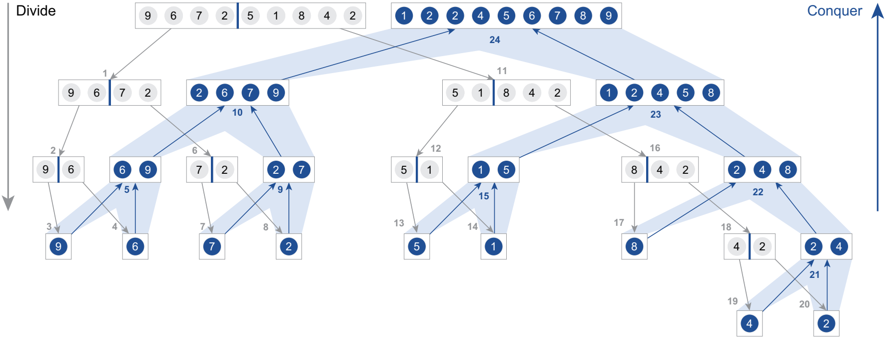
Merge Sort (11-15)



Merge Sort (16-20)



Merge Sort (21-24)

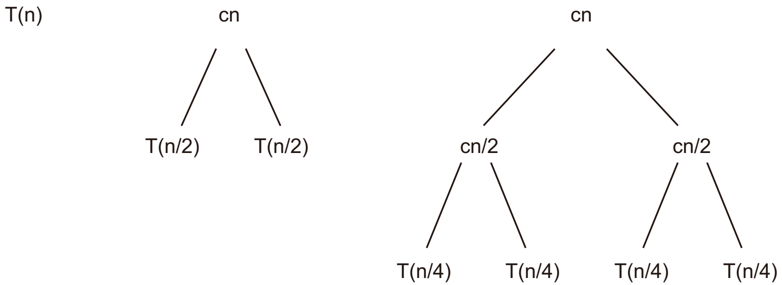


Analysis of Merge Sort (1)

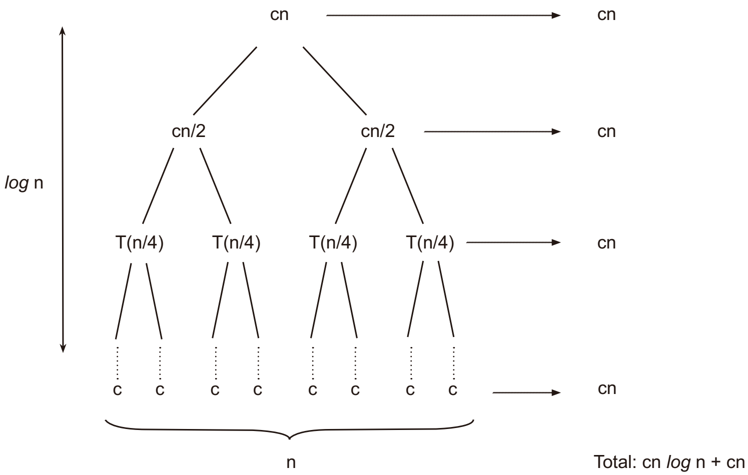
- We set up recurrence for Merge Sort.
  1. **[Divide]** The divide step computes the middle of subarray, which takes constant time. Thus, $O(1)$.
  2. **[Conquer]** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
  3. **[Combine]** The Merge procedure on an n element subarray takes time $O(n)$
- The recurrence for the worst-case running time $T(n)$ of Merge Sort is

$$T(n) = \begin{cases} c & (n = 1) \\ 2T(n/2) + cn & (n > 1) \end{cases}$$

Analysis of Merge Sort (2)



Analysis of Merge Sort (3)



Analysis of Merge Sort (4)

- Merge sort is a fast algorithm with $O(n \log n)$ in the worst case.
- Merge sort is stable because it does not swap elements which are located separately.
- Merge sort is an external sorting algorithm that requires another array in addition to the array that manages the input data.

Reference

1. Introduction to Algorithms (third edition), Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2012.