CSE331 Introduction to Algorithms
Lecture 4: Merge Sort

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021

Introduction

- Reference: Section 2.3 of the textbook Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.
- In Lecture 1, we presented INSERTION SORT.
- In this lecture, we present MERGE SORT, a more efficient algorithm.
- It follows a *Divide-and-Conquer* approach:
  - Sort recursively $A[1 \dots n/2]$ and $A[n/2+1 \dots n]$
  - Combine the results.
  - See next lectures for other examples of this approach.

Merging two Sorted Sequences

Problem (Merging two Sorted Sequences)

Given two sorted sequences $(a_1, a_2, \ldots, a_{n_1})$ *and* $(b_1, b_2, \ldots, b_{n_2})$*, sort the sequence* $(a_1, a_2, \ldots, a_{n_1}, b_1, b_2, \ldots, b_{n_2})$.

Example

- **INPUT:** $(1, 6, 10, 11, 18)$ and $(3, 4, 5, 15)$
- **OUTPUT:** $(1, 3, 4, 5, 6, 10, 11, 15, 18)$

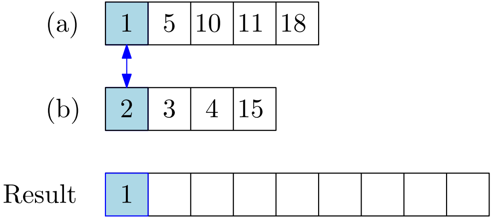
Algorithm for Merging two Sorted Sequences

(a)	1	5	10	11	18

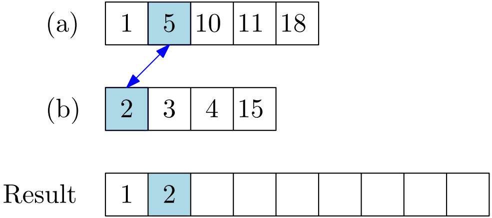
(b)	2	3	4	15

Result									

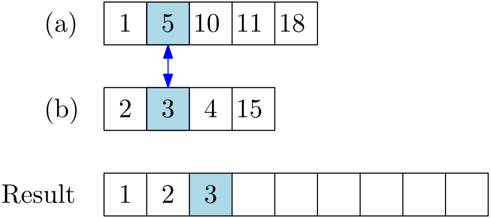
Algorithm for Merging two Sorted Sequences



Algorithm for Merging two Sorted Sequences



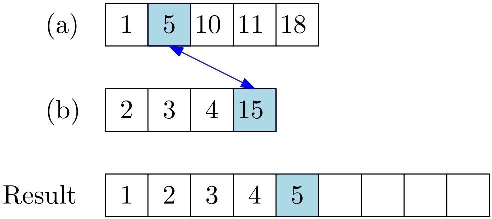
Algorithm for Merging two Sorted Sequences



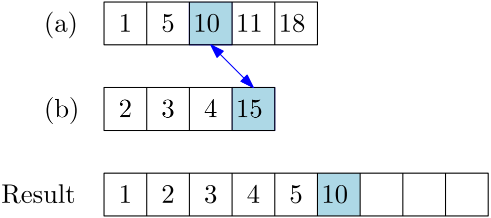
Algorithm for Merging two Sorted Sequences

(a)	1	5	10	11	18				
(b)	2	3	4	15					
Result	1	2	3	4					

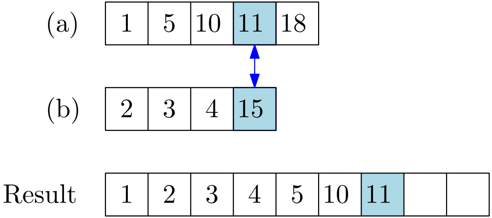
Algorithm for Merging two Sorted Sequences



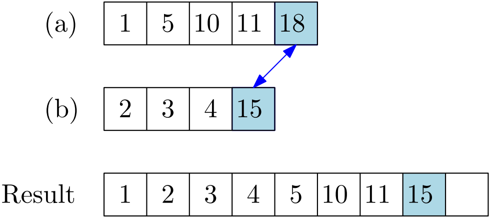
Algorithm for Merging two Sorted Sequences



Algorithm for Merging two Sorted Sequences



Algorithm for Merging two Sorted Sequences



Algorithm for Merging two Sorted Sequences

(a)	1	5	10	11	18				
(b)	2	3	4	15					
Result	1	2	3	4	5	10	11	15	18

Algorithm for Merging two Sorted Sequences

- So the algorithm works as follows:
- Keep a pointer to the current elements a_i and b_j in the input sequences, initialized to a_1 and b_1 respectively.
- At each step:
  - If $a_i \leqslant b_j$, then append a_i to the result, and move the pointer to a_i one position to the right.
  - If $a_i > b_j$, then append b_j to the result, and move the pointer to b_j one position to the right.

Algorithm for Merging two Sorted Sequences

- Next slide:
  - A more detailed pseudocode for the special case where the input sequences are two contiguous subarrays $A[p \dots q]$ and $A[q+1 \dots r]$.
  - We will place *sentinels*, represented by the value ∞, that we assume to be larger than all the keys.

	8	9	10	11	12	13	14	15	16	17
A	...	2	4	5	7	1	2	3	6	...
		k								

$p = 9$
$q = 12$
$r = 16$

	1	2	3	4	5
L	2	4	5	7	∞
	i				

	1	2	3	4	5
R	1	2	3	6	∞
	j				

Pseudocode

```
procedure MERGE(A, p, q, r)
    n1 ← q − p + 1, n2 ← r − q
    create new arrays L[1 . . . n1 + 1], R[1 . . . n2 + 1]
    for i ← 1, n1 do
        L[i] ← A[p + i − 1]
    for j ← 1, n2 do
        R[j] ← A[q + j]
    L[n1 + 1] ← ∞, R[n2 + 1] ← ∞
    i ← 1, j ← 1
    for k ← p, r do
        if L[i] ⩽ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← j + 1
```

Proof of Correctness

Loop Invariant

At the start of each iteration of the for loop of lines 10–16, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$ in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- Proof of Initialization, Maintenance and Termination done in class. See pages 32–33 of the textbook.

Analysis

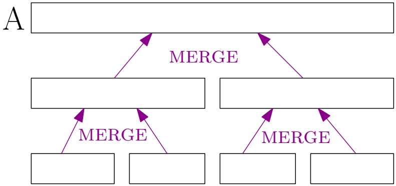
- There are respectively $n_1 + 1$, $n_2 + 1$, and $n_1 + n_2 + 1$ iterations in the three loops.
- So the running time is $c_1(n_1 + n_2) + c_2$ for some constants c_1, c_2.
- In terms of the input size $n = r - p + 1 = n_1 + n_2$, this is $c_1 n + c_2$. So we just proved:

Theorem

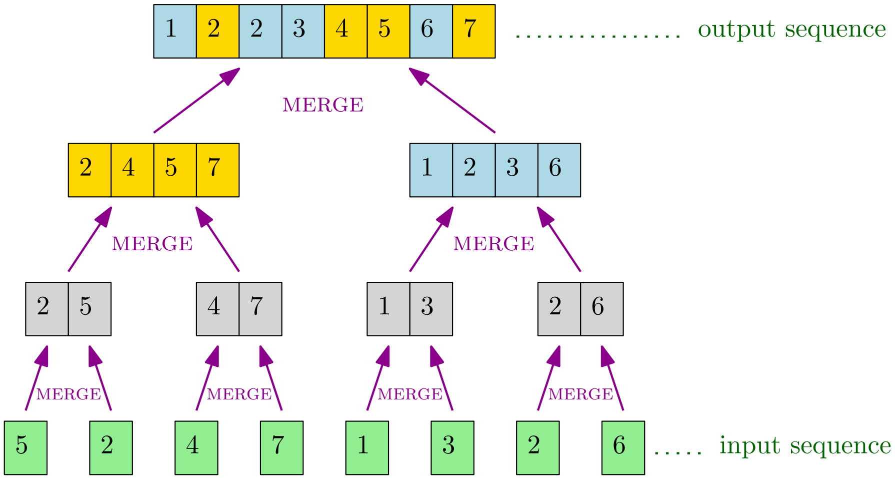
Two sorted sequences can be merged in $c_1 n + c_2$ time, where n is the sum of the lengths of the two sequences, and c_1, c_2 are two constants. In other words, two sorted sequences can be merged in linear time.

First Approach

- INSERTION SORT runs in $c_4 n^2 + c_5 n + c_6$ for some constants c_4, c_5, c_6.
- First approach:
  - (We assume $n \in 2\mathbb{N}$.)
  - Sort $A[1 \dots n/2]$ and $A[n/2+1 \dots n]$ using INSERTION SORT.
  - Merge the two results.
- Running time: $\frac{c_4}{2} n^2 + (c_1 + c_5)n + c_2 + 2c_6$
- The leading coefficient has improved by a factor 2.
- For large values of n, it is about twice faster than insertion sort.
- We can push this idea further and split A into 4 parts:
- It improves the leading coefficient further.



Merge Sort



1 2 2 3 4 5 6 7
output sequence
MERGE
2 4 5 7
1 2 3 6
MERGE
MERGE
2 5
4 7
1 3
2 6
MERGE
MERGE
MERGE
MERGE
5 2 4 7 1 3 2 6
input sequence

Merge Sort

- Merge sort splits recursively until the arrays have size 1.
  - No need to call Insertion Sort.

Pseudocode

```
procedure MergeSort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A, p, q, r)
```

- In order to sort $A[1 \ldots n]$, call $\textsc{MergeSort}(A, 1, n)$

Analysis

- Not counting the call to MERGE and the two recursive calls, MERGE SORT takes constant time c_3.
- So the running time $T(n)$ of MERGE SORT is given by the recurrence relation:

$$T(n) = \begin{cases} c_3 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_1 n + c_2 + c_3 & \text{if } n \geqslant 2 \end{cases}$$

- An upper bound $U(n) \geqslant T(n)$ is given by the relation

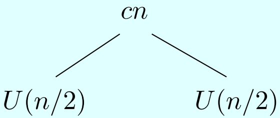
$$U(n) = \begin{cases} c & \text{if } n = 1 \\ U(\lfloor n/2 \rfloor) + U(\lceil n/2 \rceil) + cn & \text{if } n \geqslant 2 \end{cases}$$

  where $c = c_1 + c_2 + c_3$.
- We now show how to solve this recurrence relation using the *recursion tree* method.
- We first assume that n is a power of 2, i.e. $n = 2^h$ for some $h \in \mathbb{N}$.

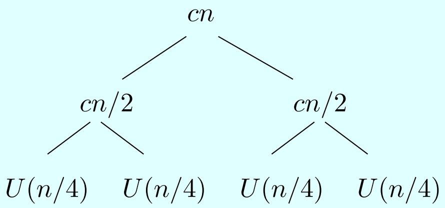
Recursion Tree Method

$$U(n)$$

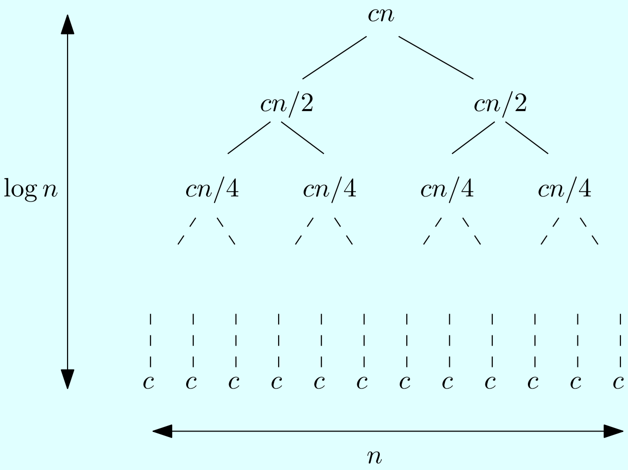
Recursion Tree Method



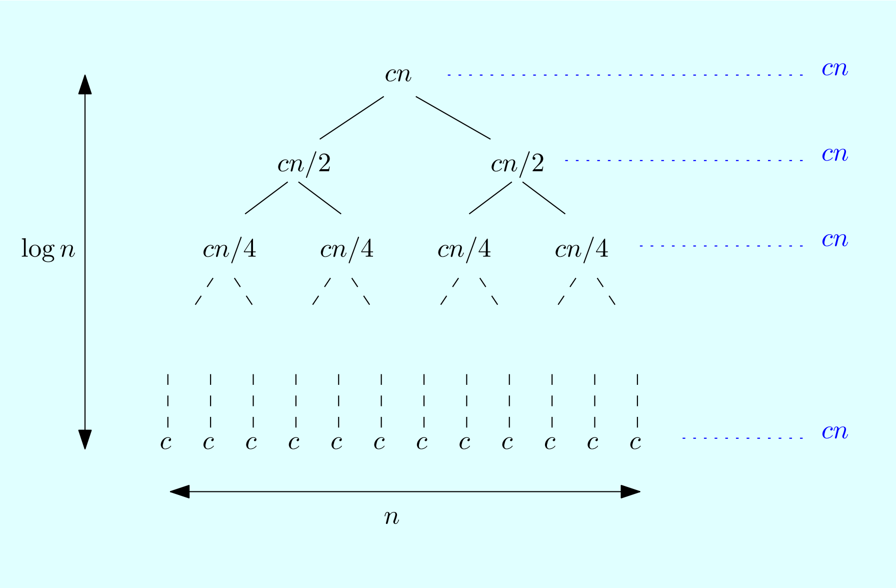
Recursion Tree Method



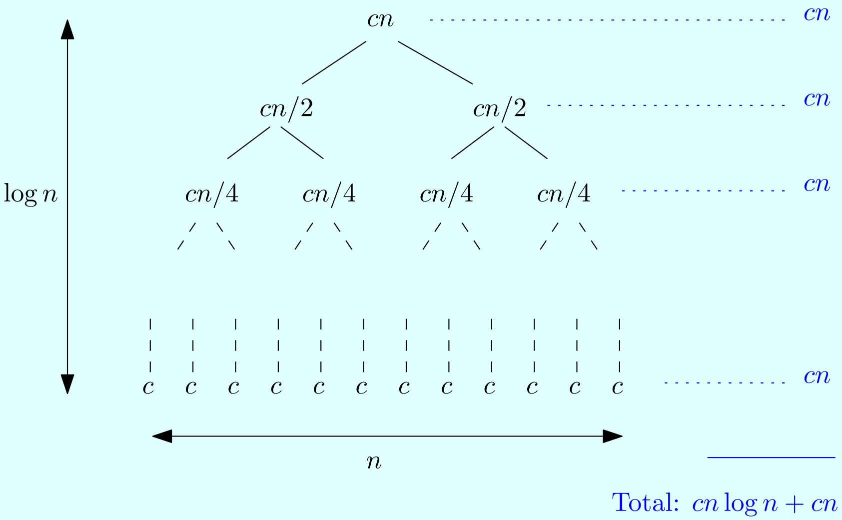
Recursion Tree Method



Recursion Tree Method



Recursion Tree Method



Recursion Tree Method

- Recursion tree method:
  - Expand the recurrence relation into a tree.
  - Add the cost across each level of the tree.
  - Add the costs of all levels.
- Here, the *height* of the tree is $\log n$.
  - In this course, log means $\log_2$
- So there are $1 + \log n$ levels in the tree.
- The cost of each level is cn.
- So $U(n) = cn \log n + cn$.

Technicality

- We only proved $U(n) = cn \log n + cn$ when n is a power of 2.
- Suppose that $2^k < n < 2^{k+1}$.
  - Then the height of the recursion tree is $k+1$. (Proof?)
  - So $U(n) < cn \log n + 2cn$.

Result

- We have just proved that $T(n) < cn \log n + 2cn$ for some constant c.
- The same approach can be used to show that $T(n) > c' n \log n$ for some constant $c' > 0$.
- It can be summarized as follows.

Theorem

MERGE SORT *runs in* $\Theta(n \log n)$ *time.*

- We will argue later this semester that $\Theta(n \log n)$ is the best possible for sorting.

Comparison with INSERTION SORT

- INSERTION SORT runs in $\Theta(n^2)$ time.
  - So for large values of n, MERGE SORT is much faster.
- In the best case, INSERTION SORT runs in $\Theta(n)$ time, while MERGE SORT still runs in $\Theta(n \log n)$ time.
  - So INSERTION SORT is better in the best case.
  - However, when we analyze algorithms, we usually pay more attention to the worst case running time.
- INSERTION SORT is better in terms of memory requirement:
  - The MERGE procedure needs to create two auxiliary arrays of linear size.
  - INSERTION SORT only stores one key outside the input array.
  - We say that INSERTION SORT is *in place*:

Definition

A sorting algorithm is *in place* if it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time.