CSE 431/531: Algorithm Analysis and Design (Fall 2024)
Divide-and-Conquer

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Solving Recurrences
6. Computing n-th Fibonacci Number

Greedy Algorithm

- mainly for combinatorial optimization problems
- trivial algorithm runs in exponential time
- greedy algorithm gives an efficient algorithm
- main focus of analysis: correctness of algorithm

Greedy Algorithm

- mainly for combinatorial optimization problems
- trivial algorithm runs in exponential time
- greedy algorithm gives an efficient algorithm
- main focus of analysis: correctness of algorithm

Divide-and-Conquer

- not necessarily for combinatorial optimization problems
- trivial algorithm already runs in polynomial time
- divide-and-conquer gives a more efficient algorithm
- main focus of analysis: running time

Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

Running time analysis

- recursive programs: recurrence

merge-sort(A, n)

```
if n = 1 then
    return A
else
    B ← merge-sort(A[1..⌊n/2⌋], ⌊n/2⌋)
    C ← merge-sort(A[⌊n/2⌋ + 1..n], ⌈n/2⌉)
    return merge(B, C, ⌊n/2⌋, ⌈n/2⌉)
```

merge-sort(A, n)

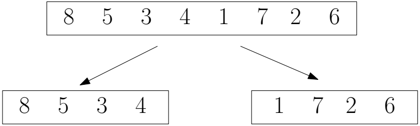
1: **if** $n = 1$ **then**
2:     **return** A
3: **else**
4:     $B \leftarrow$ merge-sort$\Big(A\big[1..\lfloor n/2 \rfloor\big], \lfloor n/2 \rfloor\Big)$
5:     $C \leftarrow$ merge-sort$\Big(A\big[\lfloor n/2 \rfloor + 1..n\big], \lceil n/2 \rceil\Big)$
6:     **return** merge$(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$

- Divide: trivial
- Conquer: 4, 5
- Combine: 6

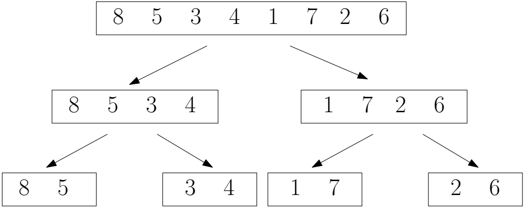
merge-sort()

8	5	3	4	1	7	2	6

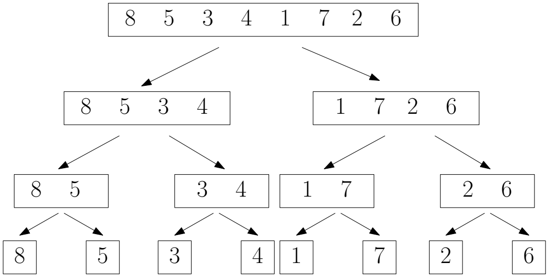
merge-sort()



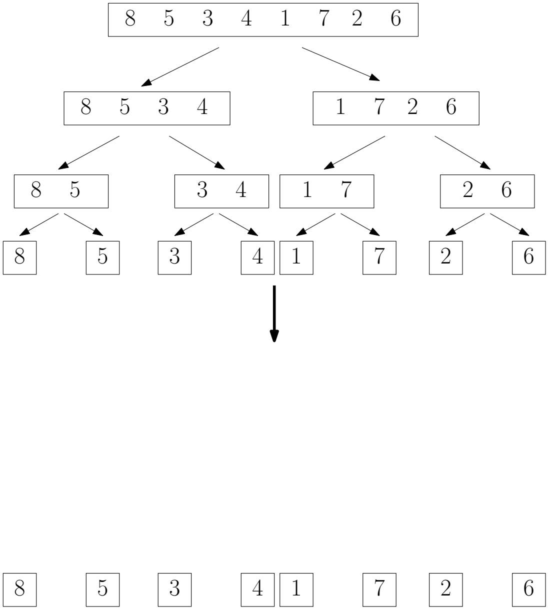
merge-sort()



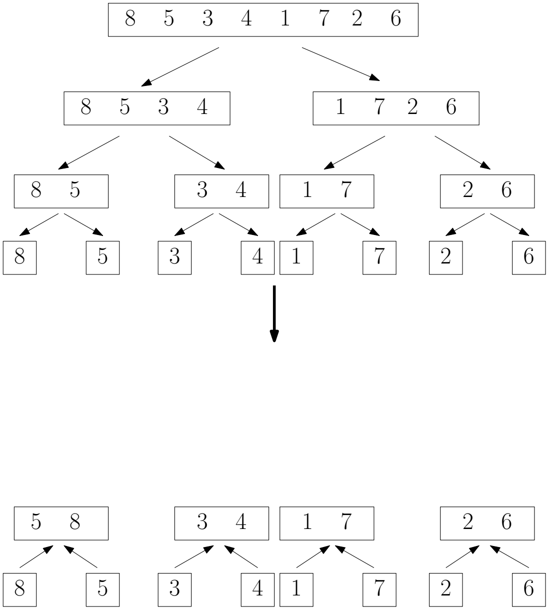
merge-sort()



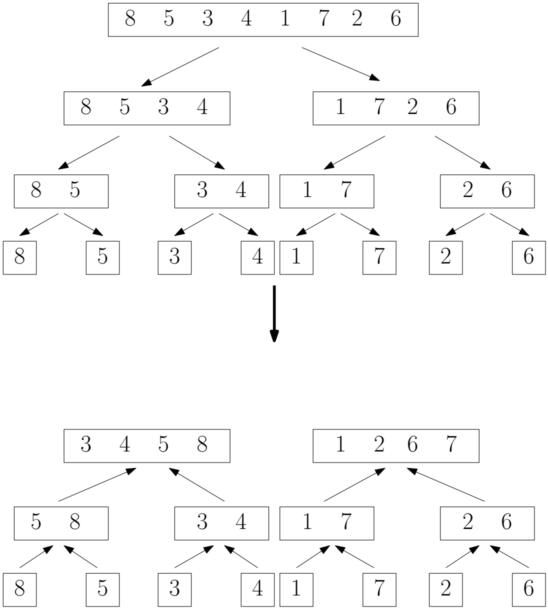
merge-sort()



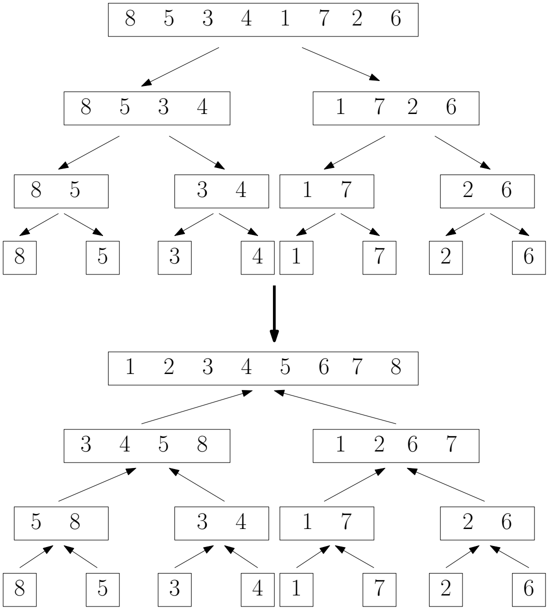
merge-sort()



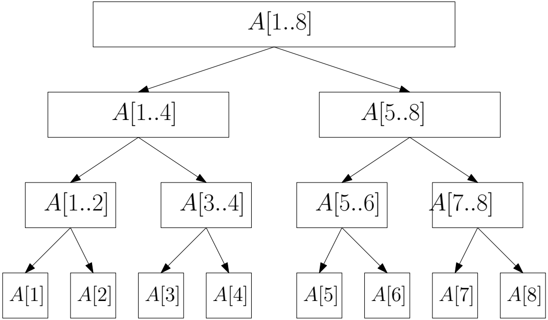
merge-sort()


8 5 3 4 1 7 2 6
8 5 3 4
1 7 2 6
8 5
3 4
1 7
2 6
8
5
3
4
1
7
2
6
3 4 5 8
1 2 6 7
5 8
3 4
1 7
2 6
8
5
3
4
1
7
2
6

merge-sort()


8 5 3 4 1 7 2 6
8 5 3 4
1 7 2 6
8 5
3 4
1 7
2 6
8
5
3
4
1
7
2
6
1 2 3 4 5 6 7 8
3 4 5 8
1 2 6 7
5 8
3 4
1 7
2 6
8
5
3
4
1
7
2
6

Running Time for Merge-Sort



- Each level takes running time $O(n)$
- There are $O(\lg n)$ levels
- Running time $= O(n \lg n)$
- Better than insertion sort

Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a+b)/2 \rfloor$: $A[a, q]$ and $A[q+1, b]$; or $A[a, q-1]$ and $A[q, b]$?

Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward

Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a + b)/2 \rfloor$: $A[a, q]$ and $A[q + 1, b]$; or $A[a, q - 1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant

Running Time for Merge-Sort

Implementation

- Divide $A[a,b]$ by $q = \lfloor (a+b)/2 \rfloor$: $A[a,q]$ and $A[q+1,b]$; or $A[a,q-1]$ and $A[q,b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant

Running Time for Merge-Sort

Implementation

- Divide $A[a, b]$ by $q = \lfloor (a+b)/2 \rfloor$: $A[a, q]$ and $A[q+1, b]$; or $A[a, q-1]$ and $A[q, b]$?
- Speed-up: avoid the constant copying from one layer to another and backward
- Speed-up: stop the dividing process when the sequence sizes fall below constant

Stable sorting algorithm

- Stable sorting algorithm has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input.

Running Time for Merge-Sort Using Recurrence

- $T(n) =$ running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

Running Time for Merge-Sort Using Recurrence

- $T(n)$ = running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

Running Time for Merge-Sort Using Recurrence

- $T(n) =$ running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Even simpler: $T(n) = 2T(n/2) + O(n)$. (Implicit assumption: $T(n) = O(1)$ if n is at most some constant.)

Running Time for Merge-Sort Using Recurrence

- $T(n) =$ running time for sorting n numbers,then

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \end{cases}$$

- With some tolerance of informality:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Even simpler: $T(n) = 2T(n/2) + O(n)$. (Implicit assumption: $T(n) = O(1)$ if n is at most some constant.)
- Solving this recurrence, we have $T(n) = O(n \lg n)$ (we shall show how later)