Merge Sort
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

It has $O(n \log n)$ running time
Merge-Sort

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 mergetSort**$(A, first, last)$

**Input** array $A[first, …, last]$

**Output** Sorted array $A$

```
if first < last then {
    mid ← (first + last) / 2
    mergeSort(A, first, mid)
    mergeSort(A, mid+1, last)
    merge(A, first, mid, last)
}
```
Algorithm \( \text{merge}(A, \text{first}, \text{mid}, \text{last}) \)

Input array \( A \) and indices \( \text{first}, \text{mid}, \text{last} \). The first half of the array \( A[\text{first}, \ldots, \text{mid}] \) is sorted, and the second half \( A[\text{mid}+1, \ldots, \text{last}] \) is also sorted.

Output sorted array \( A \)

\begin{align*}
\text{B} & \leftarrow \text{empty array of size n} \\
\text{i} & \leftarrow \text{first} \\
\text{j} & \leftarrow \text{mid} + 1 \\
\text{i}_B & \leftarrow \text{first} \\
\text{while (i <= mid) and (j <= last) do} \{ \\
& \text{if } A[\text{i}] < A[\text{j}] \text{ then} \{ \\
& & \text{B}[\text{i}_B] \leftarrow A[\text{i}] \\
& & \text{i} \leftarrow \text{i} + 1 \\
& & \} \\
& \text{else} \{ \\
& & \text{B}[\text{i}_B] \leftarrow A[\text{j}] \\
& & \text{j} \leftarrow \text{j} + 1 \\
& & \} \\
& \text{i}_B \leftarrow \text{i}_B + 1 \\
& \}\}
\end{align*}
if \( i \leq \text{mid} \) then  // There are values remaining in the first half of the array
    while (\( i \leq \text{mid} \)) do {
        \( B[i_B] \leftarrow A[i] \)
        \( i \leftarrow i + 1 \)
        \( i_B \leftarrow i_B + 1 \)
    }
else  // There are values remaining in the second half of the array
    while (\( j \leq \text{last} \)) do {
        \( B[i_B] \leftarrow A[j] \)
        \( j \leftarrow j + 1 \)
        \( i_B \leftarrow i_B + 1 \)
    }

for \( i \leftarrow \text{first} \) to \( \text{last} \) do  // Copy back all values to \( A \)
    \( A[i] \leftarrow B[i] \)

return \( A \)
Algorithm **merge** has several loops. Each iteration of each loop performs a constant number of operations. To determine the total number of iterations performed by all the loops, note that the **while** loops copy each value from A to B, so the total number of iterations that the 3 loops perform is \( n \).

The **for** loop copies all values back from B to A and so it also performs \( n \) iterations. Therefore the total number of operations performed by merge is \( c_2 \cdot n \) for some constant \( c_2 \), so the time complexity is \( O(n) \).

Let \( f(n) \) be the time complexity of **mergesort** when the input has size \( n \). The following recurrence equation characterizes the time complexity of the algorithm:

\[
\begin{align*}
    f(1) & = c, \text{ where } c \text{ is a constant} \\
    f(n) & = c_1 + c_2 \cdot n + 2f(n/2), \text{ if } n > 1, \text{ where } c_1, c_2 \text{ are constants}
\end{align*}
\]
Time Complexity

We solve the above recurrence equation using the method of repeated substitution. For simplicity, so we do not have to round numbers up or down, we assume that n is a power of 2, i.e. \( n = 2^k \), for some integer k. Hence the recurrence equation can be written as

\[
\begin{align*}
    f(2^0) &= c \\
    f(2^k) &= c_1 + c_2 2^k + 2^1 f(2^{k-1}), \text{ if } n > 1. \text{ We need to compute } 2^1 f(2^{k-1}): \\
    2^1 f(2^{k-1}) &= 2^1 c_1 + 2^1 c_2 2^{k-1} + 2^2 f(2^{k-2}), \text{ now we need to compute } 2^2 f(2^{k-2}) \\
    2^2 f(2^{k-2}) &= 2^2 c_1 + c_2 2^2 2^{k-2} + 2^3 f(2^{k-3}), \text{ and so on.} \\
    \\
    2^{k-1} f(2^1) &= 2^{k-1} c_1 + c_2 2^{k-1} 2^1 + 2^{k} f(2^0).
\end{align*}
\]

Substituting the value of \( 2^1 f(2^{k-1}) \) in the formula for \( f(2^k) \), then substituting the value of \( 2^2 f(2^{k-2}) \) in this formula and so on, we get

\[
f(2^k) = c_1 + c_2 2^k + 2^1 c_1 + 2^1 c_2 2^{k-1} + 2^2 c_1 + c_2 2^2 2^{k-2} + \ldots + 2^{k-1} c_1 + c_2 2^{k-1} 2^1 + 2^{k} f(2^0)
\]
Time Complexity

Then,

\[ f(n) = f(2^k) = c_1 + c_2 2^k + 2^1 c_1 + c_2 2^k + 2^2 c_1 + c_2 2^k + \ldots + 2^{k-1} c_1 + c_2 2^k + 2^k f(0) \]

\[ = c_1 (2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}) + c_2 2^k k + 2^k c \]

\[ = c_1 (2^k - 1) + c_2 2^k k + 2^k c = 2^k (c_1 + c) + c_2 2^k k - c_1 \]

\[ = (c_1 + c) n + c_2 n \log n - c_1 \text{ is } O(n \log n) \]
Execution Example. Execution tree

Partition

7 2 9 4 | 3 8 6 1
Recursive call, partition
Execution Example (cont.)

Recursive call, partition

7 2 9 4 | 3 8 6 1

7 2 | 9 4

7 | 2
Execution Example (cont.)

Recursive call, base case

7 2 9 4 | 3 8 6 1

7 2 | 9 4

7 → 7
Execution Example (cont.)

Recursive call, base case

7 2 9 4 3 8 6 1

7 2 | 9 4

7 2 | 7 2

7 → 7 2 → 2
Execution Example (cont.)

Merge

7 2 9 4 | 3 8 6 1

7 2 | 9 4

7 2 → 2 7

7 → 7  2 → 2
Execution Example (cont.)

Recursive call, ..., base case, merge

7 2 9 4 | 3 8 6 1
Execution Example (cont.)

Merge
Execution Example (cont.)

Recursive call, ..., merge, merge
Execution Example (cont.)

Merge

7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

7 2 | 9 4 → 2 4 7 9

3 8 6 1 → 1 3 6 8

7 | 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

3 → 3

8 → 8

6 → 6

1 → 1
## Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for small data sets</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for small data sets</td>
</tr>
<tr>
<td>Ord-Dict sort</td>
<td>$O(n \log n)$</td>
<td>fast</td>
</tr>
<tr>
<td>with AVL trees</td>
<td></td>
<td>for large data sets</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for large data sets</td>
</tr>
</tbody>
</table>