Analysis of algorithms
Textbooks

Main textbook.

- Introduction to Algorithms. Cormen, Leiserson, Rivest and Stein.

Additional textbooks.

- Algorithm design: foundations, analysis, and Internet examples. Michael T. Goodrich, Roberto Tamassia.
- Data structures and algorithms in Java. Michael T. Goodrich, Roberto Tamassia.
- Data structures and algorithms. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman.
- Data structures & their algorithms. Harry R. Lewis, Larry Denenberg.
An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Algorithm is a tool for solving a well-specified computational problem. A computational problem is defined by an input/output relationship.

Example

In the sorting problem,

Input A sequence of \( n \) number \( a_1, \ldots, a_n \).

Output A permutation \( a'_1, \ldots, a'_n \) of the input numbers which is ordered, namely \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \).
**Insertion-Sort**($A$)

(1) **for** $j = 2$ **to** $A$.length
(2) \hspace{10pt} \text{key} \leftarrow A[j]
% Insert $A[j]$ into the sorted sequence $A[1..j - 1]$
(3) \hspace{10pt} i \leftarrow j - 1
(4) \hspace{10pt} \textbf{while} i > 0 \text{ and } A[i] > \text{key}
(5) \hspace{10pt} A[i + 1] \leftarrow A[i]
(6) \hspace{10pt} i \leftarrow i - 1
(7) \hspace{10pt} A[i + 1] \leftarrow \text{key}

\[ \begin{array}{ccccccc}
6 & 2 & 5 & 7 & 1 & 4 \\
\end{array} \]
**Insertion-Sort**(\(A\))

1. \(\text{for } j = 2 \text{ to } A\text{.length}\)
2. \(\text{key } \leftarrow A[j]\)
   
   \(\%\) Insert \(A[j]\) into the sorted sequence \(A[1..j-1]\)
3. \(i \leftarrow j - 1\)
4. \(\text{while } i > 0 \text{ and } A[i] > \text{key}\)
5. \(A[i+1] \leftarrow A[i]\)
6. \(i \leftarrow i - 1\)
7. \(A[i+1] \leftarrow \text{key}\)

\(j = 2\)
**Insertion-Sort**($A$)

1. for $j = 2$ to $A$.length
2. key ← $A[j]$
   % Insert $A[j]$ into the sorted sequence $A[1..j - 1]$
3. $i ← j - 1$
4. while $i > 0$ and $A[i] >$ key
6. $i ← i - 1$
7. $A[i + 1] ←$ key

Analysis of algorithms
Insertion sort

**Insertion-Sort**(A)

1. **for** \( j = 2 \) **to** \( A\).length
2. \( \text{key} \leftarrow A[j] \)
   
   % Insert \( A[j] \) into the sorted sequence \( A[1..j-1] \)
3. \( i \leftarrow j - 1 \)
4. **while** \( i > 0 \) **and** \( A[i] > \text{key} \)
5. \( A[i+1] \leftarrow A[i] \)
6. \( i \leftarrow i - 1 \)
7. \( A[i+1] \leftarrow \text{key} \)

4 5 7 2 6 1

\( j = 3 \)
**Insertion-Sort**

(1) **for** \( j = 2 \) to \( A.\text{length} \)

(2) \( \text{key} \leftarrow A[j] \)

\( \% \) Insert \( A[j] \) into the sorted sequence \( A[1..j - 1] \)

(3) \( i \leftarrow j - 1 \)

(4) **while** \( i > 0 \) and \( A[i] > \text{key} \)

(5) \( A[i + 1] \leftarrow A[i] \)

(6) \( i \leftarrow i - 1 \)

(7) \( A[i + 1] \leftarrow \text{key} \)

4 5 7 2 6 1

4 5 7 2 6 1

\( j = 3 \)
**Insertion-Sort**($A$)

1. **for** $j = 2$ to $A$.length  
2. key ← $A[j]$  
   % Insert $A[j]$ into the sorted sequence $A[1..j - 1]$  
3. $i ← j - 1$  
4. **while** $i > 0$ and $A[i] >$ key  
6. $i ← i - 1$  
7. $A[i + 1] ←$ key

---

**4 5 6 7 2 1 4**  

$j = 4$
**Insertion-Sort**

1. **for** $j = 2$ to $A$.length
2. key ← $A[j]$
   
   % Insert $A[j]$ into the sorted sequence $A[1..j-1]$
3. $i ← j - 1$
4. **while** $i > 0$ and $A[i] > key$
6. $i ← i - 1$
7. $A[i+1] ← key$

*Analysis of algorithms*
**Insertion-Sort**

(1) **for** \( j = 2 \) **to** \( A.\text{length} \)

(2) \( \text{key} \leftarrow A[j] \)

\% Insert \( A[j] \) into the sorted sequence \( A[1..j - 1] \)

(3) \( i \leftarrow j - 1 \)

(4) **while** \( i > 0 \) and \( A[i] > \text{key} \)

(5) \( A[i + 1] \leftarrow A[i] \)

(6) \( i \leftarrow i - 1 \)

(7) \( A[i + 1] \leftarrow \text{key} \)
**Insertion-Sort**

(1)  \textbf{for} \ j = 2 \ \textbf{to} \ A.\text{length} \\
(2) \ \ \ \ \texttt{key} \leftarrow A[j] \\
\hspace{1cm} \% \ \text{Insert} \ A[j] \ \text{into the sorted sequence} \ A[1..j-1] \\
(3) \ i \leftarrow j - 1 \\
(4) \ \textbf{while} \ i > 0 \ \text{and} \ A[i] > \texttt{key} \\
(5) \ A[i+1] \leftarrow A[i] \\
(6) \ i \leftarrow i - 1 \\
(7) \ A[i+1] \leftarrow \texttt{key} \\

\begin{array}{ccccccc}
1 & 2 & 4 & 5 & 6 & 7 \\
\end{array}
At the start of each iteration of the for loop, $A[1..j-1]$ contains the elements that are initially in $A[1..j-1]$ in sorted order.

At the end of the algorithm ($j = n + 1$), the array is sorted.
We want to measure the resources used by an algorithm:
- Running time.
- Memory usage.

The complexity of an algorithm is a function of the input size.
- In the sorting problem — the number of values.
- In graph problems — the number of nodes and edges.

For a fixed input size, the algorithm complexity depends on the specific input. We can consider several options.
- Best case.
- Worst case.
- Average case.
Analysis of insertion sort

**Insertion-Sort**

1. for \( j = 2 \) to \( A.\text{length} \)
2. \( \text{key} \leftarrow A[j] \)
3. \( i \leftarrow j - 1 \)
4. while \( i > 0 \) and \( A[i] > \text{key} \)
5. \( A[i + 1] \leftarrow A[i] \)
6. \( i \leftarrow i - 1 \)
7. \( A[i + 1] \leftarrow \text{key} \)

\[
T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n - 1)
\]
The best case is when the array is sorted. In this case, case $t_j = 1$ for all $j$.

The running time is

$$T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_7(n - 1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$$= a'n - b'$$
The worst case is when the array is sorted in reverse order. In this case $t_j = j$ for all $j$.

$$
\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \text{and} \quad \sum_{j=2}^{n} (j - 1) = \frac{n(n-1)}{2}.
$$

The running time is

$$
T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \left(\frac{n(n + 1)}{2} - 1\right)
+ c_5 \left(\frac{n(n - 1)}{2}\right) + c_6 \left(\frac{n(n - 1)}{2}\right) + c_7(n - 1)
= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2
+ \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n
- (c_2 + c_3 + c_4 + c_7)
= an^2 + bn - c
$$
The worst-case running time of insertion sort is $an^2 + bn + c$ for some constants $a, b, c$.

The exact values of $a, b, c$ depend on the computer on which we run the algorithm.

In theoretical computer science, we are interested in the asymptotic complexity, namely the behavior of the worst-case complexity function as $n$ approaches infinity.

In the expression $an^2 + bn + c$, the terms $bn$ and $c$ become negligible compared to $an^2$ when $n$ approaches infinity.

Thus, the time complexity of insertion sort is $\approx an^2$.

We are not interested in the constant $a$. Thus, we say that the running time of insertion sort grows like $n^2$ when $n$ approaches infinity.
## Order of growth

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>33.2</td>
<td>100</td>
<td>1,000</td>
<td>1,024</td>
</tr>
<tr>
<td>20</td>
<td>86.4</td>
<td>400</td>
<td>8,000</td>
<td>1,048,576</td>
</tr>
<tr>
<td>30</td>
<td>147.2</td>
<td>900</td>
<td>27,000</td>
<td>1,073,741,824</td>
</tr>
<tr>
<td>100</td>
<td>664.4</td>
<td>10,000</td>
<td>1,000,000</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>1528.8</td>
<td>40,000</td>
<td>8,000,000</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>2468.6</td>
<td>90,000</td>
<td>27,000,000</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>9965.8</td>
<td>1,000,000</td>
<td>1,000,000,000</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>21931.6</td>
<td>4,000,000</td>
<td>8,000,000,000</td>
<td></td>
</tr>
</tbody>
</table>

**Analysis of algorithms**
- $\Theta(g)$ is the set of all functions that grow like $g$ when $n$ approaches infinity.

- Formally,

$\Theta(g) = \{ f : \text{there are positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$
Example

Claim
\[ \frac{1}{2} n^2 - 3n \in \Theta(n^2). \]

Proof.
We need to show that there are \( c_1, c_2, n_0 \) such that for \( n \geq n_0 \),

\[ c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2. \]

Clearly, \( \frac{1}{2} n^2 - 3n \leq \frac{1}{2} n^2 \), so we can take \( c_2 = 1/2 \).
Furthermore,

\[ c_1 n^2 \leq \frac{1}{2} n^2 - 3n \iff 3n \leq \left( \frac{1}{2} - c_1 \right) n^2 \iff \frac{3}{\frac{1}{2} - c_1} \leq n \]

We can take \( c_1 = 1/4 \) and \( n_0 = 12 \).
Claim

Let \( f(n) = b_k n^k + b_{k-1} n^{k-1} + \cdots + b_1 n + b_0 \), where \( b_k > 0 \), be a polynomial of degree \( k \). Then \( f(n) \in \Theta(n^k) \).

Proof.

Let \( a_i = |b_i| \). Then,

\[
\begin{align*}
f(n) &\leq a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\
&\leq a_k n^k + a_{k-1} n^k + \cdots + a_1 n^k + a_0 n^k \\
&= (a_k + a_{k-1} + \cdots + a_0) n^k
\end{align*}
\]

\[
\begin{align*}
f(n) &\geq a_k n^k - a_{k-1} n^{k-1} - \cdots - a_1 n - a_0 \\
&\geq a_k n^k - a_{k-1} n^{k-1} - \cdots - a_1 n^{k-1} - a_0 n^{k-1} \\
&= a_k n^k - (a_{k-1} + \cdots + a_1 + a_0) n^{k-1}
\end{align*}
\]

so \( f(n) \geq \frac{1}{2} a_k n^k \) for large enough \( n \).
Some functions in $\Theta(n^2)$ are

- $n^2 + 10n$
- $3n^2 + 100n + 5$
- $3n^2 + \sqrt{n} + \log n$
\( O(g) \) is the set of all function that grow like \( g \) or slower when \( n \) approaches infinity.

Formally,

\[
O(g) = \{ f : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}\]
Example

Some functions in $\Theta(n^2)$ are
- $n^2 + 10n$
- $3n^2 + 100n + 5$
- $3n^2 + \sqrt{n} + \log n$

Some functions in $O(n^2)$ are
- $n^2 + 10n$
- $3n^2 + 100n + 5$
- $3n^2 + \sqrt{n} + \log n$
- $n$
- $n^{1.999}$
- $\log n$
- $\log \log n$
- $\Omega(g)$ is the set of all functions that grow at least as fast as $g$ when $n$ approaches infinity.
- Formally,

$$\Omega(g) = \{ f : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

- $\Theta(g) = O(g) \cap \Omega(g)$. 

![Graph showing $f(n)$ and $cg(n)$]
Some functions in $\Omega(n^2)$ are

- $n^2 + 10n$
- $3n^2 + 100n + 5$
- $3n^2 + \sqrt{n} + \log n$
- $n^2 \log \log n$
- $4n^3$
- $2^n$
Asymptotic notation in equations and inequalities

- Stands alone on right-hand side means set membership. **Example:** \( n = O(n^2) \) means \( n \in O(n^2) \).

- In general, asymptotic notation on the right side stands for some anonymous function. **Example:** \( 2n^2 + 3n + 1 = 2n^2 + \Theta(n) \) means that \( 2n^2 + 3n + 1 = 2n^2 + f(n) \) for some function \( f(n) \) in \( \Theta(n) \).

- On the left-hand side stands for any anonymous function. **Example:** \( 2n^2 + \Theta(n) = \Theta(n^2) \) means that for any function \( g(n) \) in \( \Theta(n) \) there is a function \( f(n) \) in \( \Theta(n^2) \) such that \( 2n^2 + g(n) = f(n) \).
Intuitively,
- $O$ is like $\leq$.
- $\Omega$ is like $\geq$.
- $\Theta$ is like $=$.

Unlike real numbers, some functions cannot be compared.

For example, $f(n) = n$ and $g(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$.
Properties

Transitivity

- $f \in \Theta(g)$ and $g \in \Theta(h)$ implies $f \in \Theta(h)$.
- Same for $O, \Omega$.

Reflexivity

- $f \in \Theta(f)$.
- Same for $O, \Omega$.

Symmetry

- $f \in \Theta(g)$ if and only if $g \in \Theta(f)$.

Transpose symmetry

- $f \in O(g)$ if and only if $g \in \Omega(f)$.
The Divide and Conquer Paradigm

Divide

- Divide the problem into several (smaller) sub-problems.
- Solve the sub-problems recursively.
- Stop the recursion when the size of a sub-problem is small enough. Then, either the sub-problem does not require solving, or apply some naive algorithm on the sub-problem.

Conquer

- Combine the solutions of the sub-problems into the solution for the original problem.
Merge sort

Devide
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.

\[
\begin{array}{cccccccc}
5 & 2 & 4 & 8 & 1 & 3 & 2 & 6 \\
\end{array}
\]
Merge sort

Divide

- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer

- Merge the two sorted sub-arrays to produce the sorted array.

```
5 2 4 8 1 3 2 6
```
Merge sort

Devide

- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer

- Merge the two sorted sub-arrays to produce the sorted array.

```
5 2 4 8 1 3 2 6
```
Merge sort

Divide
- Divide the array into two sub-arrays of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.

\[
\begin{array}{cccccccc}
5 & 2 & 4 & 8 & 1 & 3 & 2 & 6
\end{array}
\]
Merge sort

Devide
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.

```
3 4 8 2
5 1
2 6
```

merge
Merge sort

**Devide**
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

**Conquer**
- Merge the two sorted sub-arrays to produce the sorted array.

```
2 5 4 8 1 3 2 6
```
Merge sort

**Divide**
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

**Conquer**
- Merge the two sorted sub-arrays to produce the sorted array.

```
2 5 4 8 1 3 2 6
```
Merge sort

Devide
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.

Analysis of algorithms
Merge sort

Devide

- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer

- Merge the two sorted sub-arrays to produce the sorted array.

| 2 | 5 | 4 | 8 | 1 | 3 | 2 | 6 |
Divide

- Divide the array into two sub-arrays of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer

- Merge the two sorted sub-arrays to produce the sorted array.
Merge sort

Devide
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.

2 4 5 8 1 3 2 6
Devide
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.
Divide
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer
- Merge the two sorted sub-arrays to produce the sorted array.
Merge sort

Devide

- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer

- Merge the two sorted sub-arrays to produce the sorted array.

\[2 \quad 4 \quad 5 \quad 8 \quad 1 \quad 2 \quad 3 \quad 6\]
Merge sort

**Divide**
- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

**Conquer**
- Merge the two sorted sub-arrays to produce the sorted array.

```
2 4 5 8 1 2 3 6
```

merge
Merge sort

Devide

- Divide the array into two sub-array of equal sizes.
- Sort the two sub-arrays recursively.
- Stop the recursion when the size of the sub-array is 1. An array of size 1 is always sorted, so no work is needed.

Conquer

- Merge the two sorted sub-arrays to produce the sorted array.

```
1 2 2 3 4 5 6 8
```
Merge sort

**MERGE-SORT**\(A, \text{left}, \text{right}\)

1. if \(\text{left} < \text{right}\)
2. \(\text{mid} \leftarrow \lfloor (\text{left} + \text{right})/2 \rfloor\)
3. \text{MERGE-SORT}(A, \text{left}, \text{mid})
4. \text{MERGE-SORT}(A, \text{mid} + 1, \text{right})
5. \text{MERGE}(A, \text{left}, \text{mid}, \text{right})

The array \(A\) is sorted by calling \text{MERGE-SORT}(A, 1, A.\text{length}).
Merge

**Merge**\((A, \text{left}, \text{mid}, \text{right})\)

(1) \( n_1 \leftarrow \text{mid} - \text{left} + 1 \)
(2) \( n_2 \leftarrow \text{right} - \text{mid} \)
(3) Create array \(L[1..n_1 + 1]\)
(4) Create array \(R[1..n_2 + 1]\)
(5) for \(i = 1\) to \(n_1\)
(6) \(L[i] \leftarrow A[\text{left} + i - 1]\)
(7) for \(j = 1\) to \(n_2\)
(8) \(R[j] \leftarrow A[\text{mid} + j]\)
(9) \(L[n_1 + 1] \leftarrow \infty\)
(10) \(R[n_2 + 1] \leftarrow \infty\)
(11) \(i \leftarrow 1\)
(12) \(j \leftarrow 1\)
(13) for \(k = \text{left}\) to \(\text{right}\)
(14) if \(L[i] \leq R[j]\)
(15) \(A[k] \leftarrow L[i]\)
(16) \(i \leftarrow i + 1\)
(17) else
(18) \(A[k] \leftarrow R[j]\)
(19) \(j \leftarrow j + 1\)

**Merge**\((A, 9, 12, 16)\)
MERGE($A$, left, mid, right)

1. $n_1 \leftarrow \text{mid} - \text{left} + 1$
2. $n_2 \leftarrow \text{right} - \text{mid}$
3. Create array $L[1..n_1 + 1]$
4. Create array $R[1..n_2 + 1]$
5. for $i = 1$ to $n_1$
   6. $L[i] \leftarrow A[\text{left} + i - 1]$
7. for $j = 1$ to $n_2$
   8. $R[j] \leftarrow A[\text{mid} + j]$
9. $L[n_1 + 1] \leftarrow \infty$
10. $R[n_2 + 1] \leftarrow \infty$
11. $i \leftarrow 1$
12. $j \leftarrow 1$
13. for $k = \text{left}$ to $\text{right}$
   14. if $L[i] \leq R[j]$
   15. $A[k] \leftarrow L[i]$
   16. $i \leftarrow i + 1$
   17. else
   18. $A[k] \leftarrow R[j]$
   19. $j \leftarrow j + 1$

Analysis of algorithms
Merge

\[ \text{Merge}(A, \text{left}, \text{mid}, \text{right}) \]

1. \( n_1 \leftarrow \text{mid} - \text{left} + 1 \)
2. \( n_2 \leftarrow \text{right} - \text{mid} \)
3. Create array \( L[1..n_1 + 1] \)
4. Create array \( R[1..n_2 + 1] \)
5. \( \text{for } i = 1 \text{ to } n_1 \)
6. \( L[i] \leftarrow A[\text{left} + i - 1] \)
7. \( \text{for } j = 1 \text{ to } n_2 \)
8. \( R[j] \leftarrow A[\text{mid} + j] \)
9. \( L[n_1 + 1] \leftarrow \infty \)
10. \( R[n_2 + 1] \leftarrow \infty \)
11. \( i \leftarrow 1 \)
12. \( j \leftarrow 1 \)
13. \( \text{for } k = \text{left} \text{ to } \text{right} \)
14. \( \text{if } L[i] \leq R[j] \)
15. \( A[k] \leftarrow L[i] \)
16. \( i \leftarrow i + 1 \)
17. \( \text{else} \)
18. \( A[k] \leftarrow R[j] \)
19. \( j \leftarrow j + 1 \)
Merge

\textbf{MERGE}(A, \text{left}, \text{mid}, \text{right})

1. \( n_1 \leftarrow \text{mid} - \text{left} + 1 \)
2. \( n_2 \leftarrow \text{right} - \text{mid} \)
3. Create array \( L[1..n_1 + 1] \)
4. Create array \( R[1..n_2 + 1] \)
5. for \( i = 1 \) to \( n_1 \)
6. \( L[i] \leftarrow A[\text{left} + i - 1] \)
7. for \( j = 1 \) to \( n_2 \)
8. \( R[j] \leftarrow A[\text{mid} + j] \)
9. \( L[n_1 + 1] \leftarrow \infty \)
10. \( R[n_2 + 1] \leftarrow \infty \)
11. \( i \leftarrow 1 \)
12. \( j \leftarrow 1 \)
13. for \( k = \text{left} \) to \( \text{right} \)
14. if \( L[i] \leq R[j] \)
15. \( A[k] \leftarrow L[i] \)
16. \( i \leftarrow i + 1 \)
17. else
18. \( A[k] \leftarrow R[j] \)
19. \( j \leftarrow j + 1 \)

Analysis of algorithms
**Merge**

**Merge**($A$, left, mid, right)

1. $n_1 \leftarrow \text{mid} - \text{left} + 1$
2. $n_2 \leftarrow \text{right} - \text{mid}$
3. Create array $L[1..n_1 + 1]$
4. Create array $R[1..n_2 + 1]$
5. for $i = 1$ to $n_1$
6. \quad $L[i] \leftarrow A[\text{left} + i - 1]$
7. for $j = 1$ to $n_2$
8. \quad $R[j] \leftarrow A[\text{mid} + j]$
9. $L[n_1 + 1] \leftarrow \infty$
10. $R[n_2 + 1] \leftarrow \infty$
11. $i \leftarrow 1$
12. $j \leftarrow 1$
13. for $k = \text{left}$ to $\text{right}$
14. \quad if $L[i] \leq R[j]$
15. \quad \quad $A[k] \leftarrow L[i]$
16. \quad \quad $i \leftarrow i + 1$
17. \quad else
18. \quad \quad $A[k] \leftarrow R[j]$
19. \quad \quad $j \leftarrow j + 1$

**Example**

$\text{Merge}(A, 9, 12, 16)$

$A = [1, 2, 5, 8, 1, 2, 3, 6]$

$L = [2, 4, 5, 8, \infty]$

$R = [1, 2, 3, 6, \infty]$

$k = 9, 10, 11, 12, 13, 14, 15, 16$
Merge

\text{MERGE}(A, \text{left}, \text{mid}, \text{right})

(1) \quad n_1 \leftarrow \text{mid} - \text{left} + 1
(2) \quad n_2 \leftarrow \text{right} - \text{mid}
(3) \quad \text{Create array } L[1..n_1 + 1]
(4) \quad \text{Create array } R[1..n_2 + 1]
(5) \quad \text{for } i = 1 \text{ to } n_1
(6) \quad \quad L[i] \leftarrow A[\text{left} + i - 1]
(7) \quad \text{for } j = 1 \text{ to } n_2
(8) \quad \quad R[j] \leftarrow A[\text{mid} + j]
(9) \quad L[n_1 + 1] \leftarrow \infty
(10) \quad R[n_2 + 1] \leftarrow \infty
(11) \quad i \leftarrow 1
(12) \quad j \leftarrow 1
(13) \quad \text{for } k = \text{left} \text{ to } \text{right}
(14) \quad \quad \text{if } L[i] \leq R[j]
(15) \quad \quad A[k] \leftarrow L[i]
(16) \quad \quad i \leftarrow i + 1
(17) \quad \quad \text{else}
(18) \quad \quad A[k] \leftarrow R[j]
(19) \quad \quad j \leftarrow j + 1

\text{Analysis of algorithms}
**Merge**

```plaintext
MERGE(A, left, mid, right)
(1)  \( n_1 \leftarrow \text{mid} - \text{left} + 1 \)  \hspace{1cm} (11)  \( i \leftarrow 1 \)
(2)  \( n_2 \leftarrow \text{right} - \text{mid} \)
(3)  Create array \( L[1..n_1 + 1] \)
(4)  Create array \( R[1..n_2 + 1] \)
(5)  for \( i = 1 \) to \( n_1 \)
(6)  \( L[i] \leftarrow A[\text{left} + i - 1] \)
(7)  for \( j = 1 \) to \( n_2 \)
(8)  \( R[j] \leftarrow A[\text{mid} + j] \)
(9)  \( L[n_1 + 1] \leftarrow \infty \)
(10) \( R[n_2 + 1] \leftarrow \infty \)
(12) \( j \leftarrow 1 \)
(13) for \( k = \text{left} \) to \( \text{right} \)
(14) if \( L[i] \leq R[j] \)
(15) \( A[k] \leftarrow L[i] \)
(16) \( i \leftarrow i + 1 \)
(17) else
(18) \( A[k] \leftarrow R[j] \)
(19) \( j \leftarrow j + 1 \)
```

**Analysis of algorithms**
**Merge**

**MERGE**(A, left, mid, right)

1. \( n_1 \leftarrow \text{mid} - \text{left} + 1 \)
2. \( n_2 \leftarrow \text{right} - \text{mid} \)
3. Create array \( L[1..n_1 + 1] \)
4. Create array \( R[1..n_2 + 1] \)
5. for \( i = 1 \) to \( n_1 \)
6. \( L[i] \leftarrow A[\text{left} + i - 1] \)
7. for \( j = 1 \) to \( n_2 \)
8. \( R[j] \leftarrow A[\text{mid} + j] \)
9. \( L[n_1 + 1] \leftarrow \infty \)
10. \( R[n_2 + 1] \leftarrow \infty \)
11. \( i \leftarrow 1 \)
12. \( j \leftarrow 1 \)
13. for \( k = \text{left} \) to \( \text{right} \)
14. if \( L[i] \leq R[j] \)
15. \( A[k] \leftarrow L[i] \)
16. \( i \leftarrow i + 1 \)
17. else
18. \( A[k] \leftarrow R[j] \)
19. \( j \leftarrow j + 1 \)

Analysis of algorithms
Merge($A$, left, mid, right)

(1) $n_1 \leftarrow \text{mid} - \text{left} + 1$
(11) $i \leftarrow 1$
(2) $n_2 \leftarrow \text{right} - \text{mid}$
(12) $j \leftarrow 1$
(3) Create array $L[1..n_1 + 1]$
(13) for $k = \text{left}$ to $\text{right}$
(4) Create array $R[1..n_2 + 1]$
(14) if $L[i] \leq R[j]$
(5) for $i = 1$ to $n_1$
(15) $A[k] \leftarrow L[i]$
(6) $L[i] \leftarrow A[\text{left} + i - 1]$
(16) $i \leftarrow i + 1$
(7) for $j = 1$ to $n_2$
(17) else
(8) $R[j] \leftarrow A[\text{mid} + j]$
(18) $A[k] \leftarrow R[j]$
(9) $L[n_1 + 1] \leftarrow \infty$
(19) $j \leftarrow j + 1$
(10) $R[n_2 + 1] \leftarrow \infty$

Analysis of algorithms
**Merge**

\[ \text{Merge}(A, \text{left}, \text{mid}, \text{right}) \]

(1) \[ n_1 \leftarrow \text{mid} - \text{left} + 1 \]
(2) \[ n_2 \leftarrow \text{right} - \text{mid} \]
(3) \[ \text{Create array } L[1..n_1 + 1] \]
(4) \[ \text{Create array } R[1..n_2 + 1] \]
(5) \[ \text{for } i = 1 \text{ to } n_1 \]
(6) \[ L[i] \leftarrow A[\text{left} + i - 1] \]
(7) \[ \text{for } j = 1 \text{ to } n_2 \]
(8) \[ R[j] \leftarrow A[\text{mid} + j] \]
(9) \[ L[n_1 + 1] \leftarrow \infty \]
(10) \[ R[n_2 + 1] \leftarrow \infty \]
(11) \[ i \leftarrow 1 \]
(12) \[ j \leftarrow 1 \]
(13) \[ \text{for } k = \text{left} \text{ to } \text{right} \]
(14) \[ \text{if } L[i] \leq R[j] \]
(15) \[ A[k] \leftarrow L[i] \]
(16) \[ i \leftarrow i + 1 \]
(17) \[ \text{else} \]
(18) \[ A[k] \leftarrow R[j] \]
(19) \[ j \leftarrow j + 1 \]

**Analysis of algorithms**
MERGE($A$, $left$, $mid$, $right$)

(1) $n_1 \leftarrow mid - left + 1$
(2) $n_2 \leftarrow right - mid$
(3) Create array $L[1..n_1 + 1]$
(4) Create array $R[1..n_2 + 1]$
(5) for $i = 1$ to $n_1$
(6) $L[i] \leftarrow A[left + i - 1]$
(7) for $j = 1$ to $n_2$
(8) $R[j] \leftarrow A[mid + j]$
(9) $L[n_1 + 1] \leftarrow \infty$
(10) $R[n_2 + 1] \leftarrow \infty$
(11) $i \leftarrow 1$
(12) $j \leftarrow 1$
(13) for $k = left$ to $right$
(14) if $L[i] \leq R[j]$
(15) $A[k] \leftarrow L[i]$
(16) $i \leftarrow i + 1$
(17) else
(18) $A[k] \leftarrow R[j]$
(19) $j \leftarrow j + 1$

Analysis of algorithms
**Analysis of merge sort**

```markdown
**MERGE-SORT**\((A, left, right)\)

1. **if** \(left < right\)
2. \(mid \leftarrow \lfloor (left + right)/2 \rfloor\)
3. **MERGE-SORT**\((A, left, mid)\)
4. **MERGE-SORT**\((A, mid + 1, left)\)
5. **MERGE**\((A, left, mid, right)\)

\[T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases}\]
```

**cost**

\(\Theta(1)\)

\(T(n/2)\)

\(T(n/2)\)

\(\Theta(n)\)