In this lecture, we will design the **merge sort** which sorts $n$ elements in $O(n \log n)$ time. The algorithm illustrates a **divide and conquer** technique based on recursion.
Recall:

The Sorting Problem

Problem Input:

A set $S$ of $n$ integers is given in an array of length $n$. The value of $n$ is inside the CPU.

Goal:

Produce an array to store the integers of $S$ in ascending order.
Recall the principle of recursion:

When dealing with a subproblem (same problem but with a smaller input), consider it solved, and use the subproblem’s output to continue the algorithm design.
Merge Sort (Divide and Conquer)

1. Sort the first half of the array $S$ (i.e., a subproblem of size $n/2$).
2. Sort the second half of the array $S$ (i.e., a subproblem of size $n/2$).
3. Consider both subproblems solved and merge the two halves of the array into the final sorted sequence (details later).
First step, sort the first half of the array by recursion.
Example

Second step, sort the second half of the array by recursion:

First, sort recursively:

```
16  ...  
```

Then, sort recursively:

```
5 9 12 17 26 28 35 38 41 47 52 68 69 72 83 88
```

Third step, merge the two halves.

First, merge:

```
16  ...  
```

Then, merge:

```
5 9 12 17 26 28 35 38 41 47 52 68 69 72 83 88
```
We are looking at the following **merging problem**.

There are two arrays — denoted as $A_1$ and $A_2$ — each containing (at most) $n/2$ integers in ascending order. The goal is to produce a sorted array $A$ containing all the integers in $A_1$ and $A_2$.

The following shows an example of the input:
Merging

At the beginning, set \( i = j = 1 \).

Repeat until \( i > n/2 \) or \( j > n/2 \):

1. If \( A_1[i] \) (i.e., the \( i \)-th integer of \( A_1 \)) is smaller than \( A_2[j] \), append \( A_1[i] \) to \( A \), and increase \( i \) by 1.

2. Otherwise, append \( A_2[j] \) to \( A \), and increase \( j \) by 1.

Think: What happens if \( i > n/2 \)? What will you do to complete the merging?
Example

At the beginning of merging:

Appending 5 to A:
Example

Appending 9 to $A$:

Appending 12 to $A$:
Example

Appending 17 to $A$:

And so on.
Running Time of Merge Sort

Let $f(n)$ denote the worst-case running time of merge sort when executed on an array of size $n$.

For $n = 1$, we have:

$$f(n) = O(1)$$

For $n \geq 1$:

$$f(n) \leq 2f(\lceil n/2 \rceil) + O(n)$$

where the term $2f(\lceil n/2 \rceil)$ is because the recursion sorts two arrays each of size at most $\lceil n/2 \rceil$, and the term $O(n)$ is the time of merging.
Running Time of Merge Sort

So it remains to solve the following recurrence:

\[
\begin{align*}
f(1) & \leq c_1 \\
f(n) & \leq 2f(n/2) + c_2 n
\end{align*}
\]

where \(c_1, c_2\) are constants (whose values we do not care). If \(n\) is a power of 2, using the expansion method, we have:

\[
\begin{align*}
f(n) & \leq 2f(n/2) + c_2 n \\
& \leq 2(2f(n/4) + c_2 n/2) + c_2 n = 4f(n/4) + 2c_2 n \\
& \leq 4(2f(n/8) + c_2 n/4) + 2c_2 n = 8f(n/8) + 3c_2 n \\
& \vdots \\
& \leq 2^i f(n/2^i) + i \cdot c_2 n \\
& \vdots \\
(h = \log_2 n) & \leq 2^h f(1) + h \cdot c_2 n \\
& \leq n \cdot c_1 + c_2 n \cdot \log_2 n = O(n \log n).
\end{align*}
\]
Running Time of Merge Sort

How to remove the assumption that $n$ is a power of 2? Hint: The rounding approach discussed in a previous lecture.