2.3 Divide & Conquer Algorithms

We often consider Recursive Algorithms, (also Divide & Conquer). These are Algorithms which call themselves in the course of execution.

MergeSort is a Recursive Sorting Algorithm. It uses a Sub-Algorithm called Merge. The call Merge(A, p, q, r), where p ≤ q ≤ r, combines Sorted Sub-Arrays A[p...q] and A[q+1...r] into a Single Sorted Sub-Array A[p...r].

\[
\begin{array}{c}
\text{SORTED} \\
A[p...q] & A[q+1...r] \\
\end{array}
\]

Exercise:
Write Pseudo-Code for Merge(A, p, q, r), (or read it on p. 29). Show that Merge Does r-p Comparisons To Sort A[p...r].
**MergeSort** $(A, p, r)$

1. If $p < r$
2. $q = \left\lfloor \frac{p + r}{2} \right\rfloor$
3. **MergeSort** $(A, p, q)$
4. **MergeSort** $(A, q+1, r)$
5. **Merge** $(A, p, q, r)$

If $p = r$, then $A[p...r]$ contains at most one element, and is therefore already sorted. **MergeSort** does nothing in this case. If $p < r$, **MergeSort** is called recursively on the subarrays $A[p...q]$ and $A[q+1...r]$, where $q = \left\lfloor \frac{p + r}{2} \right\rfloor$ is the 'middle' index. The two sorted subarrays are then combined using **Merge**.

The top level call **MergeSort** $(A, 1, n)$ sorts the full array $A[1...n]$.

**MergeSort** divides the array of size $n$ into two subarrays of size $\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil$ respectively.

```
               n
             /\    \\
            /  \   \\
         \    /    \\
          \   /     \\
          n /      \\
```

Divide

```
[\left\lfloor \frac{n}{2} \right\rfloor]  \left\lceil \frac{n}{2} \right\rceil
```

Conquer

```
          n
```

Combine
EX \( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

\( \begin{array}{ccccccc}
8 & 3 & 1 & 6 & 7 & 5 & 4 & 2 \\
\end{array} \)

We want to determine the runtime of MergeSort. As before, we take the comparison operator as parameter.

Let \( T(n) \) denote the number of array comparisons performed by MergeSort on arrays of length \( n \). Then

\[
T(n) = \begin{cases} 
0 & n = 1 \\
T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + (n-1) & n \geq 2
\end{cases}
\]

Note: Best, worst, average are same.
IF \( n \) happens to be a power of 2, this reduces to

\[
T(n) = \begin{cases} 
0 & n = 1 \\
2T\left(\frac{n}{2}\right) + (n-1) & n \geq 2 
\end{cases}
\]

The solution to this recurrence is

\[
T(n) = n \log_2 n - n + 1 \quad (n \text{ a power of 2}).
\]

Here \( \log_2 n \) denotes \( \log_2 n \).

Check:

\[
\text{RHS} = 2T\left(\frac{n}{2}\right) + (n-1) \\
= 2\left(\frac{n}{2} \log_2 \left(\frac{n}{2}\right) - \frac{n}{2} + 1\right) + (n-1) \\
= n \log_2 \left(\frac{n}{2}\right) - n + 2 + n - 1 \\
= n (\log_2 n - 1) + 1 \\
= n \log_2 n - n + 1 \\
= T(n) \\
= \text{LHS}.
\]

The highest order term in \( T(n) \) is \( n \log_2 n \), thus

\[
T(n) = \Theta(n \log_2 n).
\]

We'll see in Chapter 4 that this is
True even when \( n \) is not a power of 2.

All this is somewhat vague until we define precisely what is meant by \( \Theta(\cdot) \), but we can see that for sufficiently large \( n \)

\[
\frac{\log n}{n} < n^2,
\]

so that for large \( n \), Mergesort is considered more efficient than InsertionSort.

This is typical of the way we analyse recursive algorithms. Let \( T(n) \) be the running time (or number of parameter operations) performed on inputs of size \( n \). If \( n \) is small, say \( n \leq n_0 \) for some constant \( n_0 \), no subdivisions are necessary and the solution takes constant time:

\[
T(n) = c = \text{const}.
\]

This is the point at which the recursion 'bottoms out'.
If \( n > n_0 \) we divide the problem into \( \alpha \) subproblems, each of size \( \frac{1}{b} \) times that of the original (i.e., of size \( \frac{n}{b} \)).

Suppose our algorithm takes time \( D(n) \) to divide the problem into subproblems, and time \( C(n) \) to combine solutions to the subproblems into a solution to the original problem. We obtain the recurrence

\[
T(n) = \begin{cases} 
    C & n \leq n_0 \\
    \alpha T\left(\frac{n}{b}\right) + D(n) + C(n) & n > n_0
\end{cases}
\]

We will learn in Chapter 4 how to solve recurrences of this form, both explicitly, and in the asymptotic sense.

Read: Handout on Asymptotic Growth Rates