Lecture 33
• Previously discussed: the divide and conquer problem solving technique
  
  ○ Divide and conquer:
    
    - **Divide** the original problem into a number of *(smaller and easier)* sub-problems
    
    - **Solve each sub-problem** with an algorithm (i.e., write an algorithm for each sub-problem)
    
    - Illustration:

```
<table>
<thead>
<tr>
<th>input</th>
<th>Original problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Divide problem</td>
</tr>
<tr>
<td></td>
<td>into smaller</td>
</tr>
<tr>
<td></td>
<td>(easier) problems</td>
</tr>
<tr>
<td></td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>sub-problem 1</td>
</tr>
<tr>
<td></td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>sub-problem 2</td>
</tr>
<tr>
<td></td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>Algorithm 1</td>
</tr>
<tr>
<td></td>
<td>input</td>
</tr>
<tr>
<td></td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>Algorithm 2</td>
</tr>
</tbody>
</table>
```
Introduction to Recursion

- The recursive property
  - Some problems (especially in Mathematics) are defined in terms of itself
    - recursive problem = a problem that is defined with one or more smaller problems of the same kind

- Example: the factorial of the number \( n \)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( n! = n \times (n - 1)! )</td>
<td>( 0! = 1 )</td>
</tr>
</tbody>
</table>

Notice that:

- The problem of computing \( n! \) contains a smaller problem \((n-1)!\) of the same nature:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( n! = n \times (n - 1)! )</td>
<td></td>
</tr>
</tbody>
</table>

- It's like a Russian doll:
Illustration: compute \textit{factorial of 10}:

\[
10! = 10 \times 9!
\]

To compute \textit{10!}, we need to \textit{value of 9!}

- We must \textit{solve another (smaller) factorial problem}

Now, \textit{suppose} that you \textit{know} (e.g., someone told you) that:

\[
9! = 362880
\]

Then, we can solve the \textit{original problem (10!)} \textit{very easily}:

\[
10! = 10 \times 362880 = 3628800
\]

The \textit{recursive property}:

- Let problem \textit{X(n)} denote a certain problem of a given size \textit{n}

- A problem \textit{X(n)} has the \textit{recursive property} if

  - The problem \textit{X(n)} can be solved using the solution of one or more smaller problems \textit{X(n-1), X(n-2), ...}

  - A \textit{sufficient number} of \textit{base (simple) cases} of the problem can be solved \textit{readily}.

(The number of \textit{base case solutions} needed depends on \textit{how many} smaller problems \textit{X(n-1), X(n-2), ...} are used to solve the \textit{original problem X(n)})
Example: the \textit{factorial}($n$) problem has the \textit{recursive property}

- The problem \textit{factorial}($n$) can be \textit{solved using} the \textit{solution} of one smaller problem \textit{factorial}($n-1$)

- Because \textit{one smaller problem} is used to solve \textit{factorial}($n$), we need \textit{one base case}

The \textit{base (simple) case} $n = 0$ of the \textit{factorial problem} can be \textit{solved readily}:

\begin{quote}
\begin{center}
\texttt{factorial(0) = 1}
\end{center}
\end{quote}

- The \textit{number of base cases (already solved)} needed:

\begin{quote}
\begin{center}
\textit{In general.} if we use the \textit{solution} for Problem($n-k$) to formulate the solution for the \textit{original problem} Problem($n$):

\begin{quote}
\begin{center}
\texttt{Problem(n) = ........ Problem(n - k)}
\end{center}
\end{quote}

then we need \textit{solutions} for $k$ \textit{base case}.
\end{center}
\end{quote}
Recursion: a special divide and conquer problem solving technique

- Because there is a smaller problem of the same kind embedded inside a recursion problem, there is special divide and conquer solution procedure for recursive problem.

I like to describe this recursive solution procedure as:

- the "lazy man's method to solve a problem...."

An un-scientific (hopefully more understandable) way to describe the recursive solution method:

- If you want to solve a problem of size n that has the recursive property, then:
  - Hire (or delegate) someone else to solve the problem of size n - 1, size n - 2, etc...
  - (You would then wait for that person to come back and give you the solution)
  - When you receive the solution for the problem of size n - 1, size n - 2, etc..., you will then use these solutions to solve the original problem.
• The general form of the recursive problem solving algorithm

  • The recursive algorithm technique can be expressed in general as follows:

    ```java
    ReturnType solveProblem( n )
    {
      if ( n is one of the base cases )
      {
        return ( the readily available solution for that base case );
      }
      else
      {
        /* -----------------------------------------------
         * Delegate the problem:
         * hire someone to solve a (one or more) smaller problems
         * ----------------------------------------------- */
        sol1 = solveProblem ( n-1 );   // n-1 < n
        sol2 = solveProblem ( n-2 );   // n-2 < n
        ...
        /* -----------------------------------------------
         * Use the solutions of the smaller problems n-1, n-2, ...
         * to solve my problem:
         * ----------------------------------------------- */
        mySol = Solve "problem(n)" using sol1, sol1, ...

        return ( mySol );
      }
    }
    ```

  • Where can you use the recursive algorithm technique?

    - The recursive algorithm technique can be used to solve problems that have the recursive property.

  • Notes:

    - I can only give you the general form of the recursive algorithm technique... because
    - Each problem with the recursive property is solved differently...
• Recursive methods

  ◦ Consider the above algorithm:

```java
ReturnType solveProblem( n )
{
    if ( n is one of the base cases )
    {
        return ( the readily available solution for that base case );
    }
    else
    {
        /* --------------------------------------------------
         * Delegate the problem:
         * hire someone to solve a (one or more) smaller problems
         * -------------------------------------------------- */
        sol1 = solveProblem ( n-1 ); // n-1 < n
        sol2 = solveProblem ( n-2 ); // n-2 < n
        ...
        /* --------------------------------------------------
         * Use the solutions of the smaller problems n-1, n-2, ...
         * to solve my problem:
         * -------------------------------------------------- */
        mySol = Solve “problem(n)” using sol1, sol2, ...
        return ( mySol );
    }
}
```

Notable fact:

- When you implement the algorithm as a Java method, the resulting method will invoke itself!!!
- We call these kind of methods: recursive methods

• Definition:

  • Recursive method = a method that invokes itself

We will see some recursive methods in the next few webpages.
• Procedure to develop a recursive problem solving algorithm
  
  o General procedure to develop a recursive problem solving algorithm:

  Make sure that the problem has the recursive property (if the problem does not have the recursive property, then you cannot use recursion!!!)

  1. Find out which smaller problems you need to use to solve the original problem.

     Example:

     \[
     \text{factorial}(n) \text{: smaller problem used } = \text{factorial}(n-1)
     \]

  2. Find out how to use the solutions of the smaller problems to solve the original problem.

     Depending on the problem, this step can be quite challenging....

  3. Find the solutions for a sufficient number of the base cases.

     Rule:

     • If you used a smaller problem \( \text{Problem}(n-k) \) to solve the original problem \( \text{Problem}(n) \), then you will need to find solutions for \( k \) base cases.

     Example:

     • \( \text{factorial}(n) \) uses \( \text{factorial}(n-1) \), so you will need the solution for 1 base case.

     • We usually use the base case: \( \text{factorial}(0) = 1 \).
Computing $n!$ (factorial) using recursion

- **Problem description**
  - Problem description:
    - Write a `method` to compute the factorial of the input number $n$
    - In other words, the `method` should have the following `header`:
      ```java
      public static int fac( int n )
      {
        ....
      }
      ```
      The input parameter $n$ is an integer
      The output value ($= n!$) is also an integer value
• Divide and conquer the factorial problem
  ◦ From Mathematics:

    \[ n! = n \times (n-1)! \]
    
    with:
    
    \[ 0! = 1 \]

  ◦ The factorial(n) problem has the recursive property

    We saw in the previous webpage, that the factorial n (n!) problem has the recursive property:

    ▪ The problem \textit{factorial}(n) can be solved using the solution of one smaller problem \textit{factorial}(n−1) (i.e.: \( m_1 = n−1 \))
    
    ▪ The base (simple) case \( n = 0 \) of the factorial problem can be solved readily:

      \[ \text{factorial}(0) = 1 \]

  ◦ So we can apply the recursive algorithm technique to compute n!:

    ▪ Which smaller problem do we use to solve \textit{factorial}(n):

      \[ \text{sol}_1 = \text{factorial}(n-1) \]

    ▪ \textit{How} do we use the solution \text{sol}_1 to solve \textit{factorial}(n):

      \[ \text{factorial}(n) = n \times \text{sol}_1 \]

    ▪ Because we used \textit{factorial}(n−1), we will need the solution for 1 base case:

      \[ \text{factorial}(0) = 1 \]
The recursive algorithm to solve `factorial(n)`:

```java
int factorial( int n )
{
    if ( n == 0 /* the base cases */) 
    {
        return 1; // The readily available solution for the base case
    }
    else
    {
        /* ----------------------------------------
           Delegation: hire someone to solve (n-1)!
           ---------------------------------------- */
        sol1 = factorial( n-1 );

        /* ----------------------------------------
           Use the solutions of the smaller (n-1)!
           to solve my problem:                    */
        mySol = n * sol1;
        return mySol;
    }
}
```

The algorithm written in Java:

```java
public class Recurse
{
    public static int factorial( int n )
    {
        int sol1, mySol; // Define variable...

        if ( n == 0 /* the base cases */) 
        {
            return 1; // The readily available solution for the base case
        }
        else
        {
            sol1 = factorial( n-1 ); // Let "someone else" (-factorial(n-1)
                                     // solve the smaller problem

            mySol = n * sol1; // Use the solution of the smaller problem
                               // to solve my own problem

            return mySol; // Return my solution
        }
    }
}
```
Terminology: \textit{recursive method}

\begin{itemize}
\item \textbf{Definition:}
\begin{itemize}
\item \textbf{Recursive method} = a method that \textit{invokes itself} during the execution of that method.
\end{itemize}
\begin{itemize}
\item \textbf{Types of recursive methods:}
\begin{itemize}
\item \textbf{Direct recursion}: the method \textit{invokes itself} inside the method.
\item \textbf{Indirect recursion}: the method \textit{invokes another method} which will invoke the \textit{original method}.
\end{itemize}
\end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{Example - direct recursion:} \texttt{factorial()}
\end{itemize}

\begin{verbatim}
public class Recurse1 {
    public static int factorial( int n ) {
        int sol1, solution; // Define variable...
        if ( n == 0 /* the base cases */ ) {
            return 1; // The readily available solution for the base case
        } else {
            // factorial(n) is solved using solutions of
            // the smaller problem factorial(n-1)...
            sol1 = factorial( n-1 ); // Solve a smaller problem
            solution = n * sol1; // Use the solution of the smaller problem
                                 // to solve the original problem
            return solution;
        }
    }
}
\end{verbatim}