Data Structures and Algorithms

Recursion
The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - \( n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n \)
- Recursive definition:
  
  \[
  f(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  n \cdot f(n - 1) & \text{else}
  \end{cases}
  \]
The Recursion Pattern

• As a Java method:

```java
// recursive factorial function
public static int recursiveFactorial(int n) {
    if (n == 0) return 1; // base case
    else return n * recursiveFactorial(n - 1); // recursive case
}
```
Content of a Recursive Method

• **Base case(s).**
  – Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  – Every possible chain of recursive calls **must** eventually reach a base case.

• **Recursive calls.**
  – Calls to the current method.
  – Each recursive call should be defined so that it makes progress towards a base case.
Visualizing Recursion

- Recursion trace
- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

Example recursion trace:

```
recursiveFactorial(4)
  call
recursiveFactorial(3)
    call
recursiveFactorial(2)
      call
recursiveFactorial(1)
        call
recursiveFactorial(0)
```

```
return 4*6 = 24  →  final answer
return 3*2 = 6
return 2*1 = 2
return 1*1 = 1
return 1
```

```python
recursiveFactorial(4)  # 4! = 4*3*2*1 = 24
recursiveFactorial(3)  # 3! = 3*2*1 = 6
recursiveFactorial(2)  # 2! = 2*1 = 2
recursiveFactorial(1)  # 1! = 1
recursiveFactorial(0)  # 0! = 1
```
Example – English Rulers

• Define a recursive way to print the ticks and numbers like an English ruler:

```
----- 0
----
----- 1
----- 2
------ 0
-----
------ 1
------ 2
------ 3
```
A Recursive Method for Drawing Ticks on an English Ruler

// draw a tick with no label
public static void drawOneTick(int tickLength) {
    drawOneTick(tickLength, -1); // draw one tick
}

// draw one tick
public static void drawOneTick(int tickLength, int tickLabel) {
    for (int i = 0; i < tickLength; i++)
        System.out.print("-");
    if (tickLabel >= 0)
        System.out.print(" "+tickLabel);
    System.out.print("n");
}

public static void drawTicks(int tickLength) {
    if (tickLength > 0) {
        drawTicks(tickLength-1); // recursively draw left ticks
        drawOneTick(tickLength);  // draw center tick
        drawTicks(tickLength-1);  // recursively draw right ticks
    }
}

public static void drawRuler(int nInches, int majorLength) {
    drawOneTick(majorLength, 0); // draw tick 0 and its label
    for (int i = 1; i <= nInches; i++)
        { // draw ticks for this inch
            drawTicks(majorLength-1);
            drawOneTick(majorLength, i); // draw tick i and its label
        }
}
Visualizing the DrawTicks Method

- An interval with a central tick length $L \geq 1$ is composed of the following:
  - an interval with a central tick length $L-1$,
  - a single tick of length $L$,
  - an interval with a central tick length $L-1$.

Phạm Bảo Sơn - DSA
Recall the Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - \( n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n \)
- Recursive definition:

\[
f(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot f(n-1) & \text{else}
\end{cases}
\]

- As a Java method:

```java
// recursive factorial function
public static int recursiveFactorial(int n) {
    if (n == 0) return 1; // base case
    else return n * recursiveFactorial(n - 1); // recursive case
}
```
Linear Recursion

- **Test for base cases.**
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.

- **Recur once.**
  - Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
  - Define each possible recursive call so that it makes progress towards a base case.
Algorithm LinearSum($A$, $n$):

Input:
An integer array $A$ and an integer $n \geq 1$, such that $A$ has at least $n$ elements

Output:
The sum of the first $n$ integers in $A$

if $n = 1$ then
  return $A[0]$
else
  return LinearSum($A$, $n - 1$) + $A[n - 1]$

Example recursion trace:
Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if i < j then
    Swap A[i] and A[j]
    ReverseArray(A, i + 1, j - 1)
return

Phạm Bảo Sơn - DSA
Defining Arguments for Recursion

• In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
• This sometimes requires we define additional parameters that are passed to the method.
• For example, we defined the array reversal method as \text{ReverseArray}(A, i, j)$, not \text{ReverseArray}(A)$. 
Computing Powers

• The power function, \( p(x,n) = x^n \), can be defined recursively:

\[
p(x,n) = \begin{cases} 
1 & \text{if } n = 0 \\
x \cdot p(x, n-1) & \text{else}
\end{cases}
\]

• This leads to a power function that runs in \( O(n) \) time (for we make \( n \) recursive calls).
• We can do better than this, however.
Recursive Squaring

• We can derive a more efficient linearly recursive algorithm by using repeated squaring:

\[
p(x, n) = \begin{cases} 
1 & \text{if } x = 0 \\
x \cdot p(x, (n - 1) / 2)^2 & \text{if } x > 0 \text{ is odd} \\
p(x, n / 2)^2 & \text{if } x > 0 \text{ is even}
\end{cases}
\]

• For example,

\[
\begin{align*}
2^4 &= 2^{(4/2)}^2 = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\
2^5 &= 2^{1+(4/2)^2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\
2^6 &= 2^{(6/2)^2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\
2^7 &= 2^{1+(6/2)^2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.
\end{align*}
\]
A Recursive Squaring Method

Algorithm Power(x, n):

Input: A number x and integer n = 0
Output: The value $x^n$

if $n = 0$ then
  return 1
if $n$ is odd then
  $y = \text{Power}(x, (n - 1)/2)$
  return $x \cdot y \cdot y$
else
  $y = \text{Power}(x, n/2)$
  return $y \cdot y$

Phạm Bảo Sơn - DSA
Analyzing the Recursive Squaring Method

Algorithm Power(x, n):

*Input:* A number x and integer n = 0

*Output:* The value $x^n$

if $n = 0$ then
    return 1
if $n$ is odd then
    $y = \text{Power}(x, (n - 1)/2)$
    return $x \cdot y \cdot y$
else
    $y = \text{Power}(x, n/2)$
    return $y \cdot y$

Each time we make a recursive call we halve the value of $n$; hence, we make $\log n$ recursive calls. That is, this method runs in $O(\log n)$ time.

It is important that we used a variable twice here rather than calling the method twice.
Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

  **Algorithm** IterativeReverseArray(A, i, j):
  
  **Input:** An array A and nonnegative integer indices i and j
  
  **Output:** The reversal of the elements in A starting at index i and ending at j
  
  **while** i < j **do**
  
  - Swap A[i] and A[ j ]
  - i = i + 1
  - j = j - 1

  **return**
Binary Recursion

• Binary recursion occurs whenever there are two recursive calls for each non-base case.
• Example: the DrawTicks method for drawing ticks on an English ruler.
A Binary Recursive Method for Drawing Ticks

// draw a tick with no label
public static void drawOneTick(int tickLength) { drawOneTick(tickLength, -1); }
   // draw one tick
public static void drawOneTick(int tickLength, int tickLabel) {
   for (int i = 0; i < tickLength; i++)
      System.out.print("-");
   if (tickLabel >= 0) System.out.print(" "+tickLabel);
   System.out.print("n");
}
public static void drawTicks(int tickLength) { // draw ticks of given length
   if (tickLength > 0) {
      // stop when length drops to 0
      drawTicks(tickLength-1);
      // recursively draw left ticks
      drawOneTick(tickLength);
      // draw center tick
      drawTicks(tickLength-1);
      // recursively draw right ticks
   }
}
public static void drawRuler(int nInches, int majorLength) { // draw ruler
   drawOneTick(majorLength, 0); // draw tick 0 and its label
   for (int i = 1; i <= nInches; i++)
      { // draw ticks for this inch
      drawTicks(majorLength-1);
      drawOneTick(majorLength, i); // draw tick i and its label
   }
}

Note the two recursive calls
Another Binary Recursive Method

• Problem: add all the numbers in an integer array $A$:

  Algorithm $\text{BinarySum}(A, i, n)$:
  
  Input: An array $A$ and integers $i$ and $n$
  
  Output: The sum of the $n$ integers in $A$ starting at index $i$

  if $n = 1$ then
    return $A[i]$
  return $\text{BinarySum}(A, i, n/2) + \text{BinarySum}(A, i + n/2, n/2)$

• Example trace:
Computing Fibonacci Numbers

• Fibonacci numbers are defined recursively:
  \[ F_0 = 0 \]
  \[ F_1 = 1 \]
  \[ F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1. \]

• As a recursive algorithm (first attempt):
  
  Algorithm BinaryFib\(k\):
  
  \textbf{Input:} Nonnegative integer \(k\)
  \textbf{Output:} The \(k\)th Fibonacci number \(F_k\)
  
  if \(k \leq 1\) then
  return \(k\)
  else
  return BinaryFib\((k - 1)\) + BinaryFib\((k - 2)\)
Analyzing the Binary Recursion Fibonacci Algorithm

- Let $n_k$ denote number of recursive calls made by $\text{BinaryFib}(k)$. Then
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.

- Note that the value at least doubles for every other value of $n_k$. It is exponential!
A Better Fibonacci Algorithm

• Use linear recursion instead:

   Algorithm LinearFibonacci($k$):
   
   **Input:** A nonnegative integer $k$
   **Output:** Pair of Fibonacci numbers $(F_k, F_{k-1})$

   if $k = 1$ then
   
   return $(k, 0)$

   else

   $(i, j) = \text{LinearFibonacci}(k - 1)$

   return $(i + j, i)$

• Runs in $O(k)$ time.

Phạm Bảo Sơn - DSA
Multiple Recursion

- Motivating example: summation puzzles
  - \( pot + pan = bib \)
  - \( dog + cat = pig \)
  - \( boy + girl = baby \)

- Multiple recursion: makes potentially many recursive calls (not just one or two).
- Find all subset of a certain length.
Algorithm for Multiple Recursion

**Algorithm** PuzzleSolve(k, S, U):

**Input:** An integer k, sequence S, and set U (the universe of elements to test)

**Output:** An enumeration of all k-length extensions to S using elements in U without repetitions

1. **if** k = 0 **then**
   - Test whether S is a configuration that solves the puzzle
   - **if** S solves the puzzle **then**
     - return “Solution found: ” S
   - else
     - **for all** e in U **do**
       - Remove e from U  \{e is now being used\}
       - Add e to the end of S
       - PuzzleSolve(k - 1, S, U)
       - Add e back to U  \{e is now unused\}
       - Remove e from the end of S

Phạm Bảo Sơn - DSA
Visualizing PuzzleSolve

PuzzleSolve(3,(),{a,b,c})

- PuzzleSolve(2,a,{b,c})
  - PuzzleSolve(1,ab,{c})
  - PuzzleSolve(1,ac,{b})
    - initial call
- PuzzleSolve(2,b,{a,c})
  - PuzzleSolve(1,ba,{c})
- PuzzleSolve(2,c,{a,b})
  - PuzzleSolve(1,ca,{b})
  - PuzzleSolve(1,cb,{a})