Recursion

- A powerful problem-solving technique
- Some difficult problems often have simple recursive solutions
- Main character: Divide a problem into problems that identical in nature but smaller in size
- An alternative problem-solving approach to *iteration*

Example

Problem: Search a dictionary for a word
Solution: using binary search algorithm

```
bsearch(theDictionary, aWord) {
  if (dictionary has 1 page) {
     scan the page for aWord
  }
  else {
     divide dictionary into 2 halves:A and B
     if (aWord in A) {
        bsearch(A, aWord)
     }
     else bsearch(B, aWord)
     } // end if
  } // end if
} // end search
```

Observations

1. Method bsearch calls itself at a step
2. Dictionary size reduced to half every time of calling bsearch in the method
3. *Base case* when the dictionary has only 1 page, and
4. This special case will be eventually reached.

Deriving recursive solutions

We need to

- Define a problem in terms of smaller but identical problem
- Define a base case

How?

Example

Problem: Computing the factorial of an integer n recursively.

Solution: ?

Example

```
factorial(n)=n*(n-1)*(n-2)*..*1
            =n*factorial(n-1)
factorial(0)=1
```


Design guideline

See Chapter 1 Carrano and Prichard)

1. Analyse and understand the problem as fully as possible
2. Develop ADT and algorithms using OOD[a] and TDD[b] techniques
3. Use OOD for problems that primarily involve data
4. Use TDD to design algorithms for an object's operations
5. Consider TDD for problems that emphasize algorithms over data
6. Focus on what instead of how when designing ADTs and algorithms
7. Consider apply previously written software components into your design.

[a] Object Oriented Design
[b] Top Down Design

Java method

```
public static int factorial(int n) {
// ------------------------------------------------
// Computers the factorial of a nonnegative integer.
// Precondition: n>=0.
// Postcondition: Returns the factorial of n.
// ------------------------------------------------
  if (n==0) {
     return 1;
  }
  else {
     return factorial(n-1);
  }  // end if
} end factorial
```

Example

Problem: Derive a recursive solution for writing a string backward.

Solution: ?

Example

Problem: Derive a recursive solution for writing a string backward.

Solution: ?

Let s be a string. The algorithm in pseudocode:

```
writeBackward(s) {
  if (s is empty) {
     do nothing
  }
  else {
     write the last character of s
     writeBackward(s minus its last character)
  } // end if
} // end writeBackward
```

Java method

```
public static void writeBackward(String s,int size) {
// ------------------------------------------------
// Writes a character string backward.
// Precondition: s contains size characters and
// size>=0.
// Postcondition: s is written backward, but remains
// unchanged.
// ------------------------------------------------
  if (size>0) {
    System.out.println(s.substring(size-1, size));
    writeBackward(s, size-1);
  } // end if
  // base case: size==0, do nothing
} // end writeBackward
```


Example

Counting Problem: Printing the Fibonacci sequence.

Solution: ?

Analysis of the problem

```
fibonacciTerm(n)=
       fibonacciTerm(n-1)+
       fibonacciTerm(n-2) when n>2
fibonacciTerm(1)=
       fibonacciTerm(2)=1
```


Java method

```
public static int fibonacciTerm(int n) {
// ----------------------------------------
// Computes a term in the Fibonacci sequence.
// Precondition: n is a positive integer.
// Postcondition: Returns value of the nth
//                Fibonacci term.
// ----------------------------------------
  if (n<=2) {
     return 1;
  }
  else {
    return fibonacciTerm(n-1)+fibonacciTerm(n-2);
  } // end if
} // end fibonacciTerm
```


Algorithm

```
fibonacciTerm(n) {
  if (n=1) or (n=2) {
    return 1;
  }
  else {
    return fibonacciTerm(n-1)+
              fibonacciTerm(n-2);
  } // end if
} // end fibonacciTerm
```

Recursion and Efficiency

- Recursive solutions can be inefficient.
- Two factors contribute to potential inefficiency:
  1. The overhead associated with method calls
  2. the inherent inefficiency of some recursive algorithms.

- By efficient algorithm, informally, we primarily mean that the number of execution steps of the algorithm is relatively small.
- By efficient program, informally, we primarily mean that the time required to complete a algorithm step is relatively short.
- An iterative solution can be more efficient than a recursive solution.

Recursion and Efficiency

```
public static int iterativeFibonacciTerm(int n) {
// Iterative solution to the Fibonacci problem.
  // initialise base cases:
  int previous=1;
  int current=1;
  int next=1;

  // compute next Fibonacci term when n>=3
  for (int i=3; i<=n; i++) {
    next=current+previous;
    previous=current;
    current=next;
  } // end for
  return next;
} // end iterativeFibonacciTerm
```