Recursion

Data structures and Algorithms
What is recursion?

- A way of thinking about problems.
- A method for solving problems.
- Related to mathematical induction.

In programming:
- A function calls itself
  - direct recursion

- A function calls its invoker
  - indirect recursion

```cpp
int f () {
    ... f(); ...
}

int g () {
    ... g(); ...
    ... f(); ...
}
```
Outline of a Recursive Function

if (answer is known) provide the answer
else make a recursive call to solve a smaller version of the same problem

base case  recursive case
Recursive Factorial Method

- $n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$
- $n! = n \times (n-1)!$
- $0! = 1$

Algorithm `recursiveFactorial(n)`

if $n == 0$ then
  return 1
else
  return $n \times \text{recursiveFactorial}(n-1)$
Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21, ....

\[
\begin{align*}
\text{fib}\,(n) &= \begin{cases} 
1 & \text{for } n = 1 \\
1 & \text{for } n = 2 \\
\text{fib}\,(n-2) + \text{fib}\,(n-1) & \text{for } n > 2
\end{cases}
\end{align*}
\]

Algorithm \(\text{fib}(n)\)
- if \(n\leq2\) then
  - return 1
- else
  - return \(\text{fib}(n-2) + \text{fib}(n-1)\)
Tracing \text{fib}(6)

Algorithm \text{fib}(n)
if \( n \leq 2 \) then
    return 1
else
    return \text{fib}(n-2) + \text{fib}(n-1)

computation repeats!
Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of \( n \), that can be solved directly.
- A problem of a given size \( n \) can be split into one or more \textit{smaller} versions of the \textit{same} problem (recursive case).
- Recognize the base case and provide a solution to it.
- Devise a strategy to split the problem into smaller versions of itself while making progress toward the base case.
- Combine the solutions of the smaller problems in such a way as to solve the larger problem.
Euclid's Algorithm

- Finds the greatest common divisor of two nonnegative integers that are not both 0
- Recursive definition of gcd algorithm
  - gcd (a, b) = a        (if b is 0)
  - gcd (a, b) = gcd (b, a % b)     (if b != 0)
- Implementation:

```c
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```
Iterative vs. recursive gcd

```cpp
int gcd (int a, int b) {
    int temp;
    while (b != 0) {
        temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

```cpp
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```
Multiple recursion

- Tail recursion: a linearly recursive method makes its recursive call as its last step.
  - e.g. recursive gcd
  - Can be easily converted to non-recursive methods

- Binary recursion: there are **two** recursive calls for each non-base case
  - e.g. fibonacci sequence

- Multiple recursion: makes potentially **many** recursive calls (not just one or two).
Multiple recursion – Example

List all 'abc' strings of length \( l \)

```c
void listAllStrings(int length, char* start) {
    if (length < 1) {
        // base case: empty string
        *start = '\0';
        output();
    } else {
        // recursive case: reduce length by 1
        for (char c = 'a'; c <= 'c'; c++) {
            *start = c;
            listAllStrings(length-1, start+1);
        }
    }
}
```

Recursion 11
Why using recursion?

- Recursion makes your code faster? No!
  - overhead for function call and return
  - values recomputed

- Recursion uses less memory? No!
  - overhead for a function call and return (stack memory)

- Recursion makes your code simple? Sometimes.
  - readable code that is easy to debug