Recursion

1 Definition

Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem.
Key components:

- **Base case**: the condition under which the recursion stops.
- **Recursive case**: the part where the function calls itself with a simpler or smaller argument.

2 How It Works

Each recursive call is pushed onto the call stack. When a base case is reached, the stack unwinds, returning results back up the chain.

3 Example 1: Factorial

Compute $n!$ defined by

$$n! = \begin{cases} 1, & n \leq 1, \\ n \times (n-1)!, & n > 1. \end{cases}$$

```cpp
// factorial.cpp
#include <iostream>

unsigned long factorial(unsigned int n) {
    if (n <= 1)                 // base case
        return 1;
    return n * factorial(n-1); // recursive case
}

int main() {
    std::cout << "5! = " << factorial(5) << "\n";
    return 0;
}
```

4 Example 2: Fibonacci (Naïve)

Compute the nth Fibonacci number:

$$F(n) = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ F(n-1) + F(n-2), & n > 1. \end{cases}$$

```cpp
// fibonacci.cpp
#include <iostream>

unsigned int fib(unsigned int n) {
    if (n < 2)                  // base case
        return n;
    return fib(n-1) + fib(n-2); // recursive case
}

int main() {
    std::cout << "F(6) = " << fib(6) << "\n";
    return 0;
}
```

5 Notes and Pitfalls

- **Infinite recursion**: missing or incorrect base case leads to stack overflow.
- **Performance**: naïve recursion (e.g., Fibonacci) may have exponential time complexity.

6 More Examples of Recursion in C++

Examples of common problems that can be expressed both iteratively and recursively

6.1 Computing Integer Powers

Iterative Version:

```cpp
int power_iter(int base, int exponent) {
int result = 1;
for (int i = 0; i < exponent; ++i) {
result *= base;
}
return result;
}
```

Recursive Version:

```cpp
int power_rec(int base, int exponent) {
if (exponent == 0) return 1;
return base * power_rec(base, exponent - 1);
}
```

6.2 Sum of First n Natural Numbers

Iterative Version:

```cpp
int sum_iter(int n) {
int sum = 0;
for (int i = 1; i <= n; ++i) {
sum += i;
}
return sum;
}
```

Recursive Version:

```cpp
int sum_rec(int n) {
if (n == 0) return 0;
return n + sum_rec(n - 1);
}
```

6.3 Reversing a String

Iterative Version:

```
string reverse_iter(const string& s) {
string result = "";
for (int i = s.size - 1; i >= 0; --i) {
result += s[i];
}
return result;
}
```

Recursive Version:

```
string reverse_rec(const string& s) {
if (s.empty()) return "";
return reverse_rec(s.substr(1)) + s[0];
}
```

These examples show how recursive thinking can naturally model problems that involve self-similarity or can be broken into subproblems of the same type. Recursion often leads to simpler and more elegant code, though care must be taken to avoid excessive stack usage or inefficient repeated computations.