TYPES OF RECURSIONS

There are many types of recursions, but we can categorize them into two main types: direct recursion and indirect recursion.

1. Direct Recursion: the function calling itself within its body.

a. Linear recursion: function calls itself once every time it runs. This is the most popular type of recursion because it is the simplest form of recursion. Example:

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

In this function, we calculate the factorial of a given integer. The function will call itself recursively with n – 1 as the argument for the recursive call, until it reaches the base case which is n = 1, it terminates. The process of recursively calling itself is called "Winding", and when it returns, we call it "Un-Winding". If we pass 4 into this function, it will return 24 which is 4 * 3 * 2 * 1.

b. Tail recursion: function calls itself at the end of its body. Everything is done before the self-reference occurs. Example:

```
int evenCount(int *arr, int i, int count, int size)
{
    if (i >= size)
        return count;
    if (arr[i] % 2 == 0)
        ++count;
    evenCount(arr, i + 1, count, size);
}
```

In this function, we check for every integer in the array and output the count of even integers. Each time the self-reference occurs, the function check whether the next integer in the array is even, if true then increment "count" and move on, until it reaches the end of the array. Tail recursion sometimes is just a recursive style of a for or a while loop.

c. Binary recursion: function calls itself twice in its body, unlike linear recursion which only call itself once. It is often used in tree traversal, but a simpler example of binary recursion is the generate the n-th Fibonacci number function. Example:

```
int fibo(int n)
{
    if (n == 1 || n == 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
```

In this function, we call the function twice in its body since each Fibonacci number is defined by adding the two Fibonacci numbers before it. The function will self-reference until it reaches the base case: the first and second Fibonacci number which is 1. This process occurs recursively until it calculates all the numbers needed to find the required Fibonacci number.

d. Nested recursion: function recursive calls will take itself as an argument. We could also say "recursion in recursion". Example:

```
int nested(int n)
{
    if (n > 0)
        return n - 1;
    return nested(nested(n + 2));
}
```

In this function, whenever the argument n is larger than zero, it will minus n by 1 and terminate immediately. Otherwise, it will call itself again, but this time, the argument is itself with n + 2 as parameter. This function will output n – 1 if n > 0 or 0 if n <= 0.

2. Indirect recursion (Mutual recursion): function call each other recursively, which means if function A call function B, then function B will call function A again, this recursive process occurs until it reaches the terminated conditions. Example:

```
int x(int n)
{
    if (n == 0)
        return 1;
    return x(n - 1) + y(n - 1);
}

int y(int n)
{
    if (n == 0)
        return 0;
    return 3 * x(n-1) + 2 * y(n-1);
}
```

If we run one of these function with $n > 0$, it must call the other mutual function to calculate the result. In this example, if we call $x(2)$ then the result is 4, if we call $y(2)$ then the result is 9.

Conclusion:

Recursion is a costly function because it relies heavily on the stack memory. If we cannot maintain the base case to terminate the recursive calls, recursion is not recommended. We only use recursion when a problem can be broken down to smaller problems which use the same method to solve. Recursion might be clear for problems like factorial or Fibonacci numbers but in other cases, if we can use iteration, we should not use recursion, since iteration's readability and time complexity is clearer and more memory efficient.