Recursion

- Once we learn Inheritance we will understand:

- What is recursion and recursive algorithm
- Learn recursion elements
- Decide when to use recursion
- Write recursive methods

Recursion

- *Recursion* is the process in which a method calls itself directly or indirectly.
- Apply the result of a method to the method.
- It perform the same steps multiple times with different inputs.
- In every step, the input breaks down to smaller inputs to make the problem smaller.
- Commonly used in mathematical problems and sorting algorithms.

Recursion

- Can be used as a substitute to iteration.
- Instead of looping, it divides the problem to a sub-problems of the same type as the original.
- There is a necessary step called *Base case or condition.*
- *Base case:* needed to stop the recursion, otherwise infinite loop will be the outcome.
- *Recursive case:* Result is achieved by combining the result of smaller problem of the same type.

Recursion Elements

- There are three basic elements to recursion:
- 1) A test to stop or continue the recursion.
- 2) An end case that eliminates the recursion.
- 3)A recursive call that continue the recursion.

Finding Recursion Solution

- Steps to achieve recursive solution:
- 1) Write the solution for a few input cases to understand the recursive structure.
- 2) Figure out how the problem can be broken into smaller problem with similar steps to identify the recursive case.
  - Think of how does solving the smaller problem leads to the answer of larger problem.
- 3) Identify the base case.
- 4) Formulate the solution with the base case and the recursive case.

Iteration vs. Recursion

	Recursion	Iteration
Implementation	Self calling	Loops
State	Parameter value	Loop index
Progression	Looking for base index	Evaluate the condition
Termination	Base case reached	Index reach condition
Code size	Less	More
Speed	Slower	Faster

Recursion Pros & Cons

- **Advantages:**
- Easier to read/ Write.
- Easier to debug.
- **Disadvantages**:
- Can be slower.
- Use more memory.

Stack over flow

- What is a stack?
- What is the stack?
- When any Method is called from main(), the memory is allocated to it on the stack.
- Stack Overflow error occurs If the base case is not reached or not defined.

Recursion Example

- The factorial of N is the product of the first N positive integers.
- N! = N * (N -1) * (N -2) * (N -3) * ……. * 2 * 1
- Think how to solve it without for-loop?

$$\text{factorial}(N) = \begin{cases} 1 & \text{if } N = 1 \\ N * \text{factorial}(N-1) & \text{otherwise} \end{cases}$$

Recursion Example

$$\text{factorial}(N) = \begin{cases} 1 & \text{if } N = 1 \\ N * \text{factorial}(N-1) & \text{otherwise} \end{cases}$$

```
public int factorial(int n){

if (n<1) return 1;
else return n * factorial(n-1);
}
```

Check if the answer

is correct for

N = 13, N = 14? Why?

I count Backwards Example

- I count backwards display the numbers in descending order.
- I.e. if N=10 the output is: 10 9 8 ……. 2 1
- Think how to solve it without for-loop?

I count Backwards Example

- I count backwards display the numbers in descending order.
- I.e. if N=10 the output is: 10 9 8 ……. 2 1
- Think how to solve it without for-loop?

```
public static int ICountBackwards(int n){
System.out.print(n + " " );
if (n==1) return 1;
else return ICountBackwards(n-1);
}
```

Exponent Example

- Given a base and an exponent, find the base exponent recursively..
- I.e. if base= 2 & exponent = 4
  - Output is: 16
  - That is 2 * 2 * 2 * 2
- Think how to solve it without for-loop?

Exponent Example

- Given a base and an exponent, find the base exponent recursively..

```
public int Exponent(int base, int exp){
if(exp < 1) return 1;
else return base * Exponent(base, exp-1);
}
```

Array search Example

- Given an array arr[] of n elements, write a method to recursively search a given element x in arr[].
- Think of looking from the left and the right.
-

```
public int RecursiveSearch(int [] arr, int left ,
int right , int number){

//Fill it up
}
```

When Not To Use Recursion

- The **Fibonacci numbers**, commonly denoted F_n, form a sequence, the **Fibonacci sequence**, in which each number is the sum of the two preceding ones.
- $F_n = F_{n-1} + F_{n-2}$, …… $F_0 = 0$, $F_1 = 1$
- i.e : 0. 1. 1. 2. 3. 5. 8. 13. 21. 34. 55. 89. 144

$$\text{fibonacci}(N) = \begin{cases} 1 & \text{if } N = 0 \text{ or } N = 1 \\ \text{fibonacci}(N-1) + \text{fibonacci}(N-2) & \text{otherwise} \end{cases}$$

When Not To Use Recursion

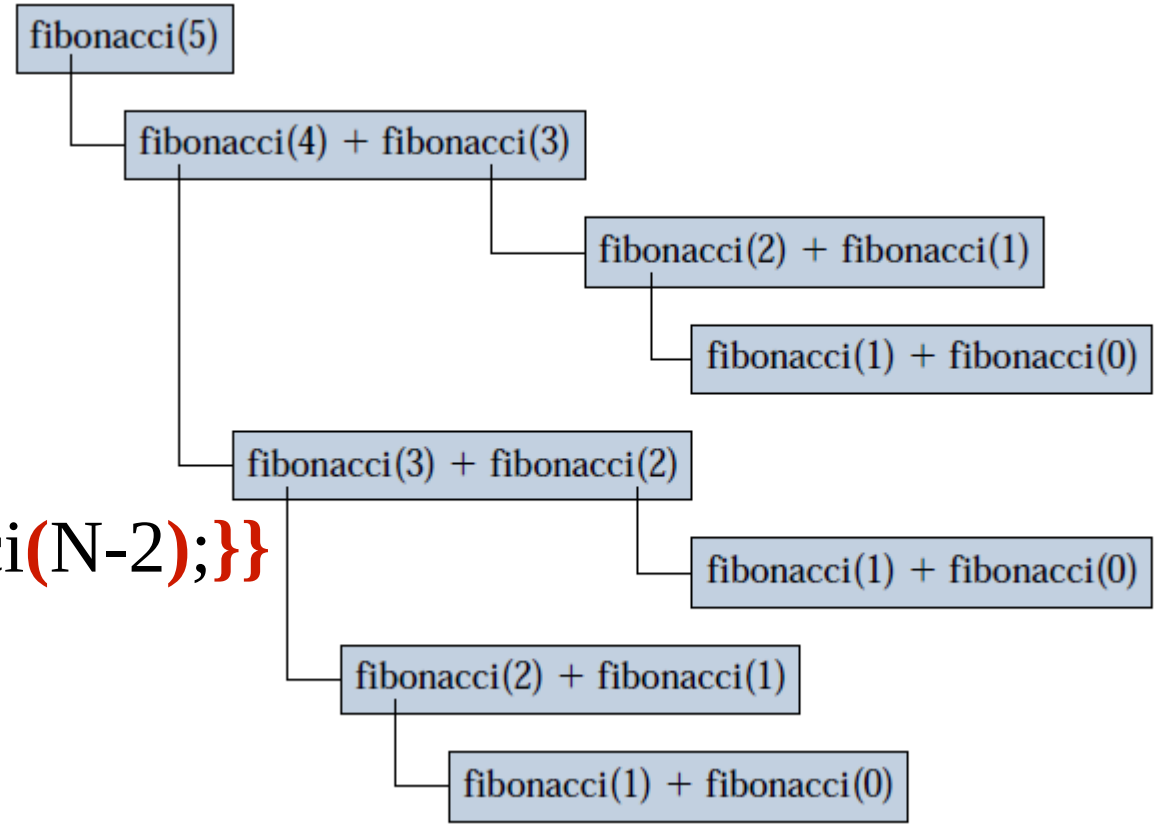
$$\text{fibonacci}(N) = \begin{cases} 1 & \text{if } N = 0 \text{ or } N = 1 \\ \text{fibonacci}(N-1) + \text{fibonacci}(N-2) & \text{otherwise} \end{cases}$$

```
public int fibonacci(int N) {
if (N == 0 || N == 1) {
return 1; //end case
} else {//recursive case
return fibonacci(N-1) + fibonacci(N-2);}}
```

When Not To Use Recursion

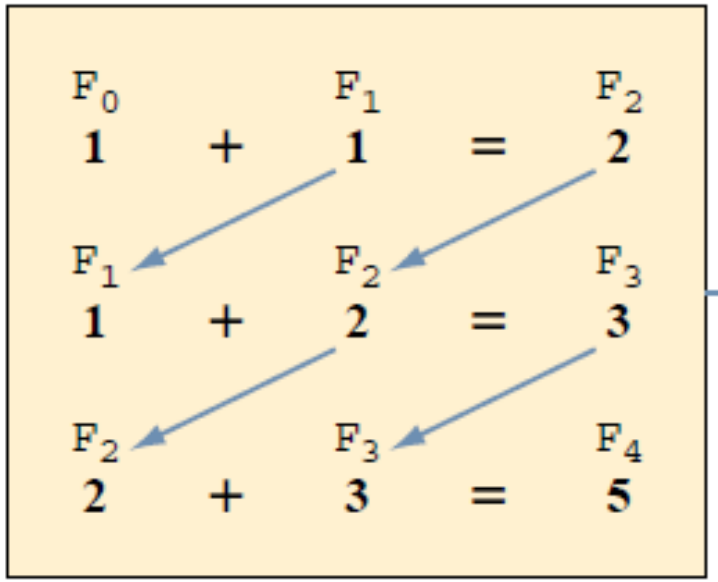
- Take an example Fibonacci(5);

```
public int fibonacci(int N) {
if (N == 0 || N == 1) {
return 1; //end case
} else {//recursive case
return fibonacci(N-1) + fibonacci(N-2);}}
```



When Not To Use Recursion

- The none recursive is more efficient.
- It is not that difficult to understand.




```
public int fibonacci(int N) {
int fibN, fibN1, fibN2, cnt;
if (N == 0 || N == 1) {
return 1;
} else {
fibN1 = fibN2 = 1; cnt = 2;
while (cnt <= N) {
fibN = fibN1 + fibN2; //get the next fib no.

fibN1 = fibN2;
fibN2 = fibN;

cnt ++;
}
return fibN;
}
```

Possible Recursion Issues

- Missing base case.
- No guarantee of coverage, by not covering all possible subproblems.
- Excessive space requirement, stack overflow.
- Excessive recomputation, i.e Fibonacci case.