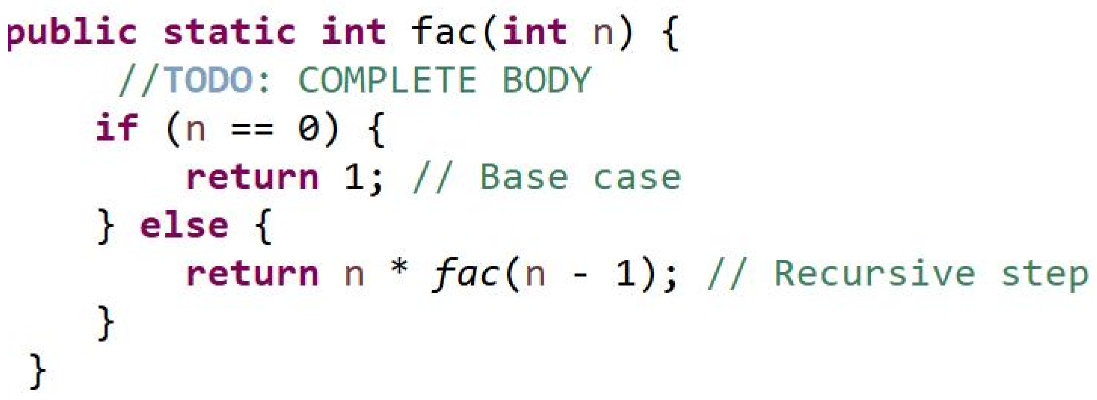
Recursion

fac Function:

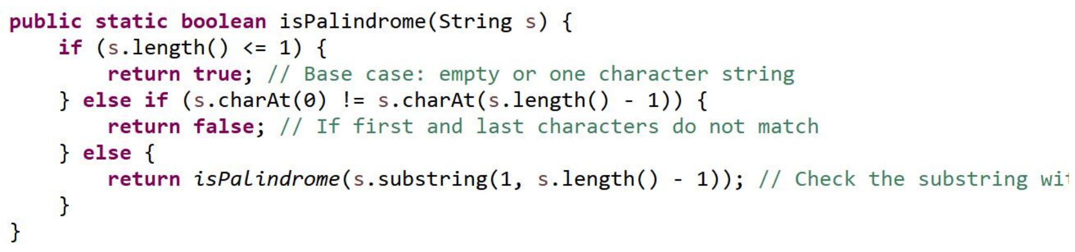
The fac function takes an integer n as input and returns its factorial. It uses recursion to calculate factorial. The function first checks whether the input n is equal to zero or not. If it is, then it returns 1 (base case). Otherwise, the function multiplies n with the result of a recursive call of fac() function with n-1 as input.



```
public static int fac(int n) {
     //TODO: COMPLETE BODY
    if (n == 0) {
        return 1; // Base case
    } else {
        return n * fac(n - 1); // Recursive step
    }
 }
```

isPalindrome Function:

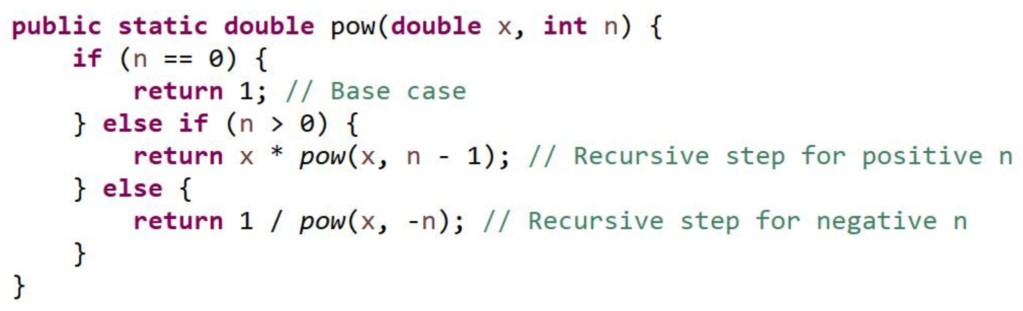
The isPalindrome function takes a string as input and returns a boolean value indicating whether the string is a palindrome or not. A string is said to be a palindrome if it reads the same backward as forward. The function uses recursion to check whether the string is a palindrome. It first checks whether the length of the string is less than or equal to one (base case). If it is, then it returns true. Otherwise, it checks whether the first and last characters of the string match. If they do, then the function recursively calls itself with the substring obtained by removing the first and last character of the string. If they don't match, then it returns false.



```
public static boolean isPalindrome(String s) {
    if (s.length() <= 1) {
        return true; // Base case: empty or one character string
    } else if (s.charAt(0) != s.charAt(s.length() - 1)) {
        return false; // If first and last characters do not match
    } else {
        return isPalindrome(s.substring(1, s.length() - 1)); // Check the substring wi
    }
}
```

pow Function:

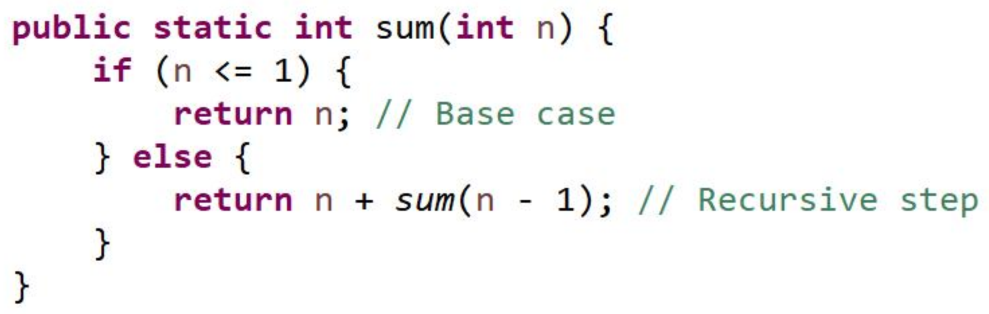
The pow function takes two inputs, a double x, and an integer n and returns x raised to the power of n. It uses recursion to calculate the power of x. If n is equal to 0, then it returns 1 (base case). If n is positive, then the function recursively calls itself with n-1 as input and multiplies the result with x. If n is negative, then it recursively calls itself with -n as input (to make it positive) and returns 1 divided by the result.



```java
public static double pow(double x, int n) {
    if (n == 0) {
        return 1; // Base case
    } else if (n > 0) {
        return x * pow(x, n - 1); // Recursive step for positive n
    } else {
        return 1 / pow(x, -n); // Recursive step for negative n
    }
}
```

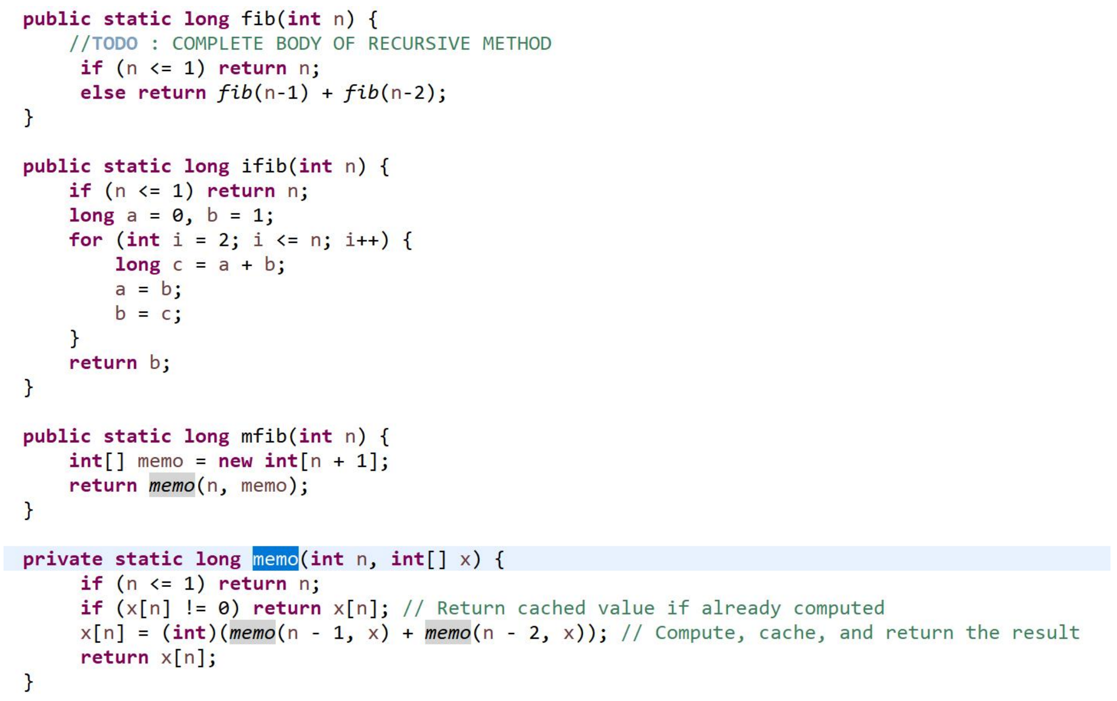
sum Function:

The sum function takes an integer n as input and returns the sum of the first n positive integers. It uses recursion to calculate the sum. If n is less than or equal to 1, then it returns n (base case). Otherwise, the function recursively calls itself with n-1 as input and adds n to the result.



```java
public static int sum(int n) {
    if (n <= 1) {
        return n; // Base case
    } else {
        return n + sum(n - 1); // Recursive step
    }
}
```

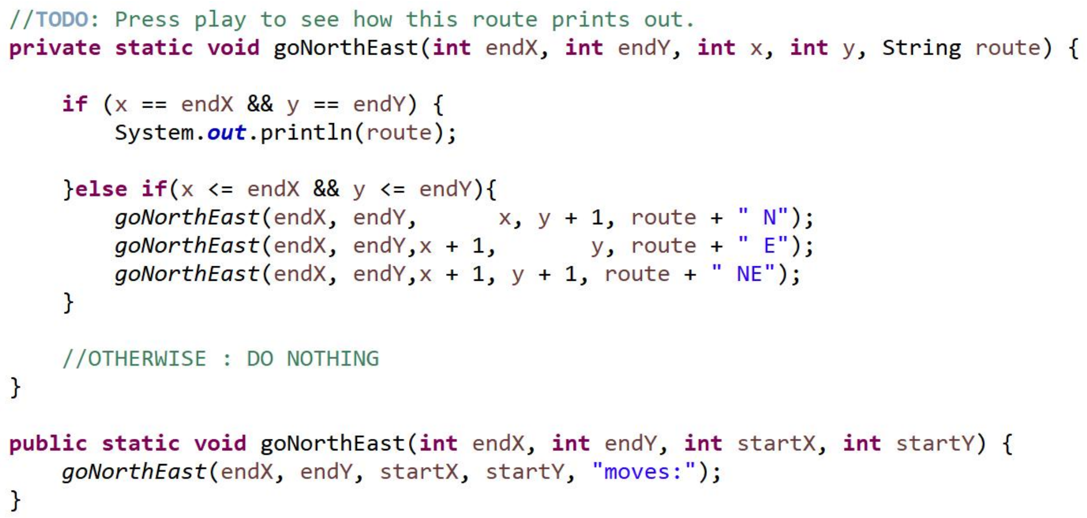
Memoization



```java
public static long fib(int n) {
    //TODO : COMPLETE BODY OF RECURSIVE METHOD
     if (n <= 1) return n;
     else return fib(n-1) + fib(n-2);
}

public static long ifib(int n) {
    if (n <= 1) return n;
    long a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        long c = a + b;
        a = b;
        b = c;
    }
    return b;
}

public static long mfib(int n) {
    int[] memo = new int[n + 1];
    return memo(n, memo);
}

private static long memo(int n, int[] x) {
     if (n <= 1) return n;
     if (x[n] != 0) return x[n]; // Return cached value if already computed
     x[n] = (int)(memo(n - 1, x) + memo(n - 2, x)); // Compute, cache, and return the result
     return x[n];
}
```

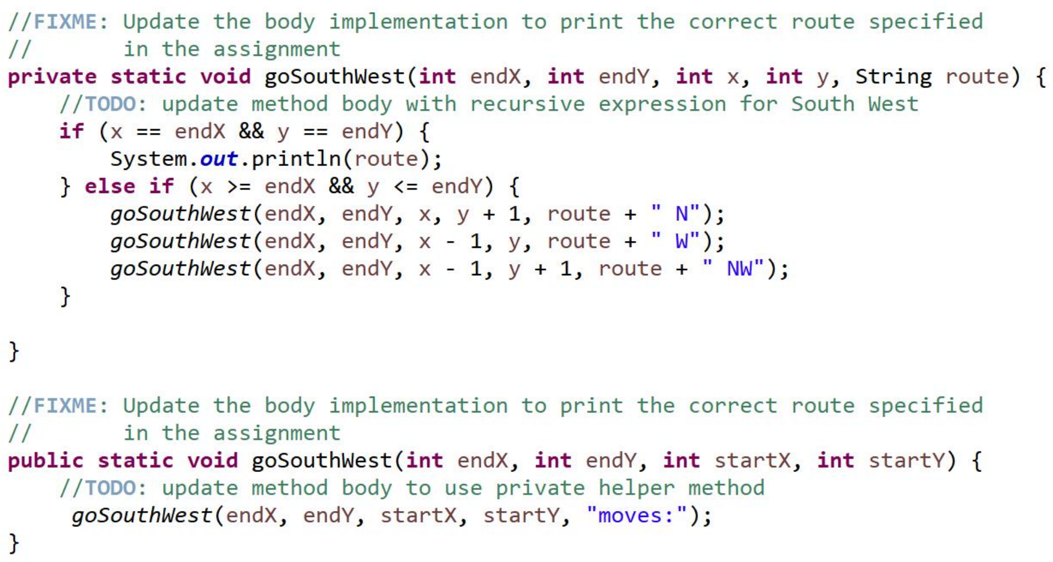
The Test02Fibonacci class provides three different implementations of the Fibonacci sequence: a recursive approach, an iterative approach, and a memoized recursive approach. The recursive fib method calculates the Fibonacci sequence by recursively calling itself for the previous two numbers in the sequence until it reaches the base case. The iterative ifib method uses a loop to iteratively calculate the Fibonacci sequence by maintaining two variables and updating them in each iteration. The mfib method implements memoization by caching previously computed Fibonacci numbers in an array, avoiding redundant calculations. Additionally, the class includes various testing methods to print and analyze the results of each implementation.

Recursive Backtracking



```java
//TODO: Press play to see how this route prints out.
private static void goNorthEast(int endX, int endY, int x, int y, String route) {

    if (x == endX && y == endY) {
        System.out.println(route);

    }else if(x <= endX && y <= endY){
        goNorthEast(endX, endY,      x, y + 1, route + " N");
        goNorthEast(endX, endY,x + 1,       y, route + " E");
        goNorthEast(endX, endY,x + 1, y + 1, route + " NE");
    }

    //OTHERWISE : DO NOTHING
}

public static void goNorthEast(int endX, int endY, int startX, int startY) {
    goNorthEast(endX, endY, startX, startY, "moves:");
}
```

The goNorthEast method in the Test05TravelDirection class is responsible for printing all possible routes from a starting point (x, y) to an ending point (endX, endY) while only allowing movements to the north, east, or northeast directions. The method uses recursion to explore all possible paths by incrementing the y-coordinate for moving north, the x-coordinate for moving east, and both coordinates for moving northeast. If the current position (x, y) matches the ending point (endX, endY), the method prints the route. Otherwise, it recursively calls itself to explore all valid movements until reaching the destination.



```java
//FIXME: Update the body implementation to print the correct route specified
//        in the assignment
private static void goSouthWest(int endX, int endY, int x, int y, String route) {
    //TODO: update method body with recursive expression for South West
    if (x == endX && y == endY) {
        System.out.println(route);
    } else if (x >= endX && y <= endY) {
        goSouthWest(endX, endY, x, y + 1, route + " N");
        goSouthWest(endX, endY, x - 1, y, route + " W");
        goSouthWest(endX, endY, x - 1, y + 1, route + " NW");
    }

}

//FIXME: Update the body implementation to print the correct route specified
//        in the assignment
public static void goSouthWest(int endX, int endY, int startX, int startY) {
    //TODO: update method body to use private helper method
     goSouthWest(endX, endY, startX, startY, "moves:");
}
```

On the other hand, the goSouthWest method is intended to print all possible routes from a starting point (startX, startY) to an ending point (endX, endY) while only allowing movements to the south, west, or northwest directions. Similar to goNorthEast, this method also utilizes recursion to examine different paths by updating the x and y coordinates accordingly for each direction. If the current position aligns with the ending point, the route is printed. Otherwise, the method continues exploring valid movements through recursive calls until reaching the target destination.