5: Recursive Functions

Recursive Definition:
A function is defined by itself.

1 1

Factorials

Formal definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

2

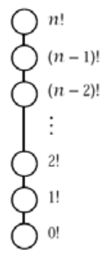
Factorial (2)

```
int  factorial(int n)
/* Pre: n is a nonnegative integer.
Post: Return the value of the factorial of n. */
{
   if (n == 0)
       return 1;
   else
       return ( n * factorial(n − 1) );
}
```

3

Factorial (3): Example

factorial(5) = 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
= 5 * (4 * (3 * (2 * (1 * factorial(0)))))
= 5 * (4 * (3 * (2 * (1 * 1))))
= 5 * (4 * (3 * (2 * 1)))
= 5 * (4 * (3 * 6))
= 5 * (4 * 6)
= 5 * 24
= 120.



4

Factorial (4): Iterative

The *factorial function of a positive integer* is

n! = n x (n − 1) x ... x1

Nonrecursive (Iterative) approach:

```
int factorial(int n)
{
   int count, product = 1;
   for (count = 1; count <= n; count++)
      product *= count;
   return product;
}
```

5

Fibonacci Sequence

Fibonacci numbers are defined by the recurrence relation

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

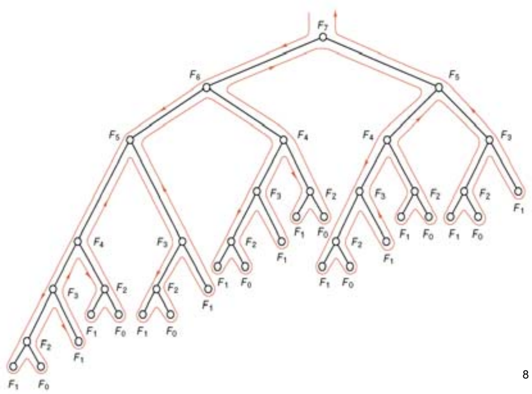
6

Fibonacci Sequence (2)

```
int fibonacci(int n)
/* Fibonacci: recursive version
Pre: The parameter n is a nonnegative integer.
Post: The function returns the nth Fibonacci number. */
{
   if (n <= 0) return 0;
   else if (n == 1) return 1;
   else return ( fibonacci(n − 1) + fibonacci(n − 2) );
}
```

7

Fibonacci Sequence (3)



8

Fibonacci Sequence (4): Nonrecursive function

```
int fibonacci(int n)
/* Fibonacci: iterative version
Pre: The parameter n is a nonnegative integer.
Post: The function returns the nth Fibonacci number. */
{
    int last_but_one;  // second previous Fibonacci number, Fi−2
    int last_value;    // previous Fibonacci number, Fi−1
    int current;       // current Fibonacci number Fi
```

9

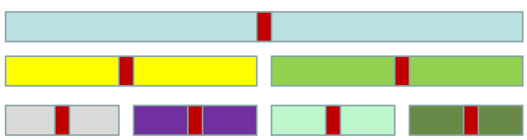
Fibonacci Sequence (4): Nonrecursive function/2

```
    if (n <= 0) return 0;
    else if (n == 1) return 1;
    else {
        last_but_one = 0;
        last_value = 1;
        for (int i = 2; i <= n; i++) {
            current = last_but_one + last_value;
            last _but_one = last_value;
            last_value = current;
        }
        return current;
    }
}
```

10

Recursive: Binary Search Example

Search for an item in an array of items, assuming the array of N items are sorted (in ascending or descending order)

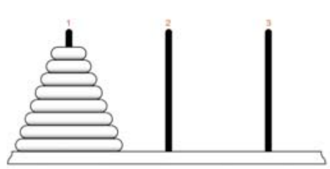


Each time, the array is divided into two subarrays, and the search would either left or right, but not both.

Therefore, the search becomes $\log_2(N)$

11

Tower of Hanoi



Rules:

- Move only one disk at a time.
- No larger disk can be on top of a smaller disk.

12

Tower of Hanoi (2)

void move(int count, int start, int finish, int temp);

Pre: There are at least count disks on the tower start. The top disk (if any) on each of towers temp and finish is larger than any of the top count disks on tower start.

Post: The top count disks on start have been moved to finish; temp (used for temporary storage) has been returned to its starting position.

13

Tower of Hanoi (3): Recursive function

```
void move(int count, int start, int finish, int temp)
{
   if (count > 0) {
     move(count − 1, start, temp, finish);
     cout << "Move disk " << count << " from " <<
       start << " to " << finish << "." << endl;
     move(count − 1, temp, finish, start);
   }
}
```

14

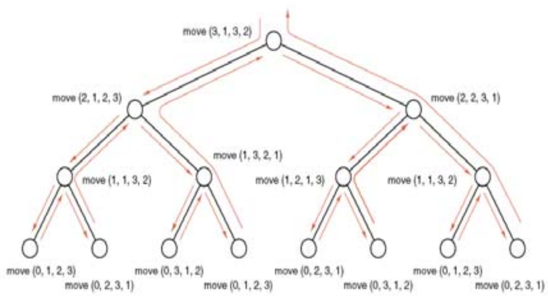
Tower of Hanoi (4): main( )

```
const int disks = 3;
void move(int count, int start, int nish, int temp);
/* Pre: None.
Post: The simulation of the Towers of Hanoi has terminated.*/

void main( )
{
   move(disks, 1, 3, 2);
}
```

15

Tower of Hanoi (5)



16