# Recursion

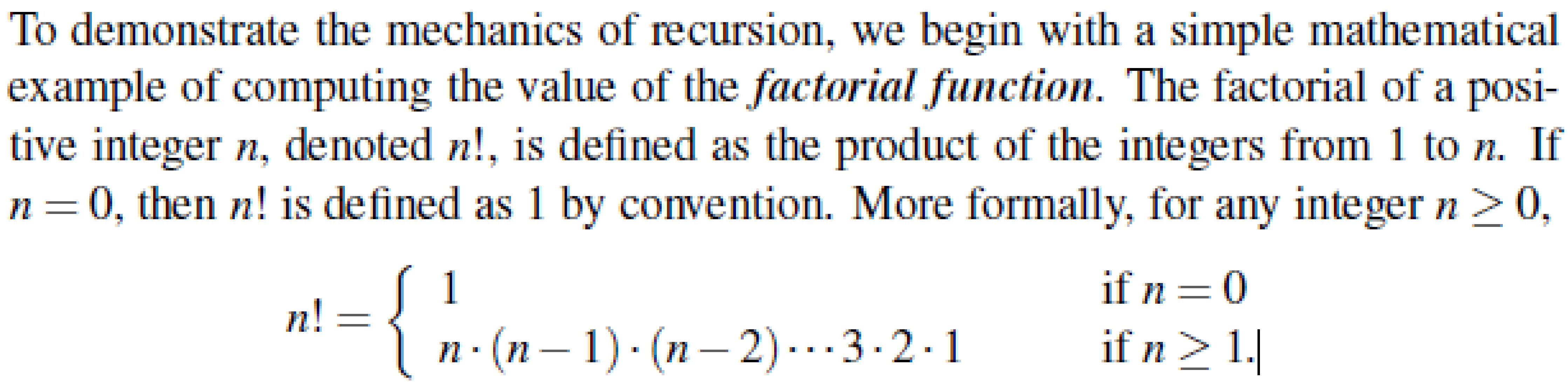
DR. SANGA CHAKI

## Recursion

## Recursion

1. One way to describe repetition within a computer program is the use of loops
2. An entirely different way to achieve repetition is through a process known as recursion.
3. Recursion is a technique by which a method makes one or more calls to itself during execution

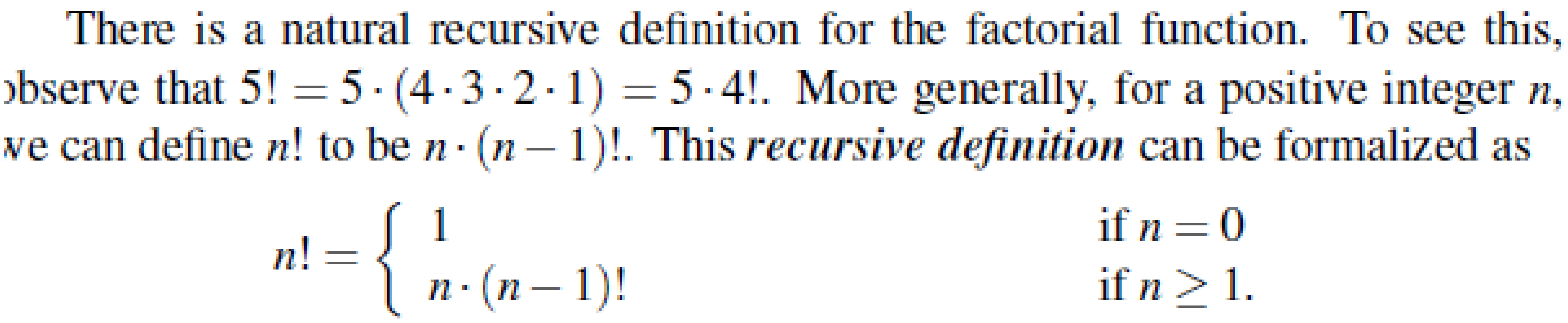
## The Factorial Function



To demonstrate the mechanics of recursion, we begin with a simple mathematical example of computing the value of the *factorial function*. The factorial of a positive integer $n$, denoted $n!$, is defined as the product of the integers from 1 to $n$. If $n = 0$, then $n!$ is defined as 1 by convention. More formally, for any integer $n \geq 0$,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

## The Factorial Function



There is a natural recursive definition for the factorial function. To see this, bserve that $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$. More generally, for a positive integer $n$, ve can define $n!$ to be $n \cdot (n-1)!$. This ***recursive definition*** can be formalized as

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

## The Factorial Function

1. This definition is typical of many recursive definitions of functions.
2. First, we have one or more base cases, which refer to fixed values of the function.
3. The above definition has one base case stating that $n! = 1$ for $n = 0$.
4. Second, we have one or more recursive cases, which define the function in terms of itself.
5. In the above definition, there is one recursive case, which indicates that $n!=n\cdot(n-1)!$ for $n\geq1$.

## Recursion – Program for Factorial

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );     // argument must be nonnegative
  else if (n == 0)
    return 1;                                  // base case
  else
    return n * factorial(n-1);                 // recursive case
}
```

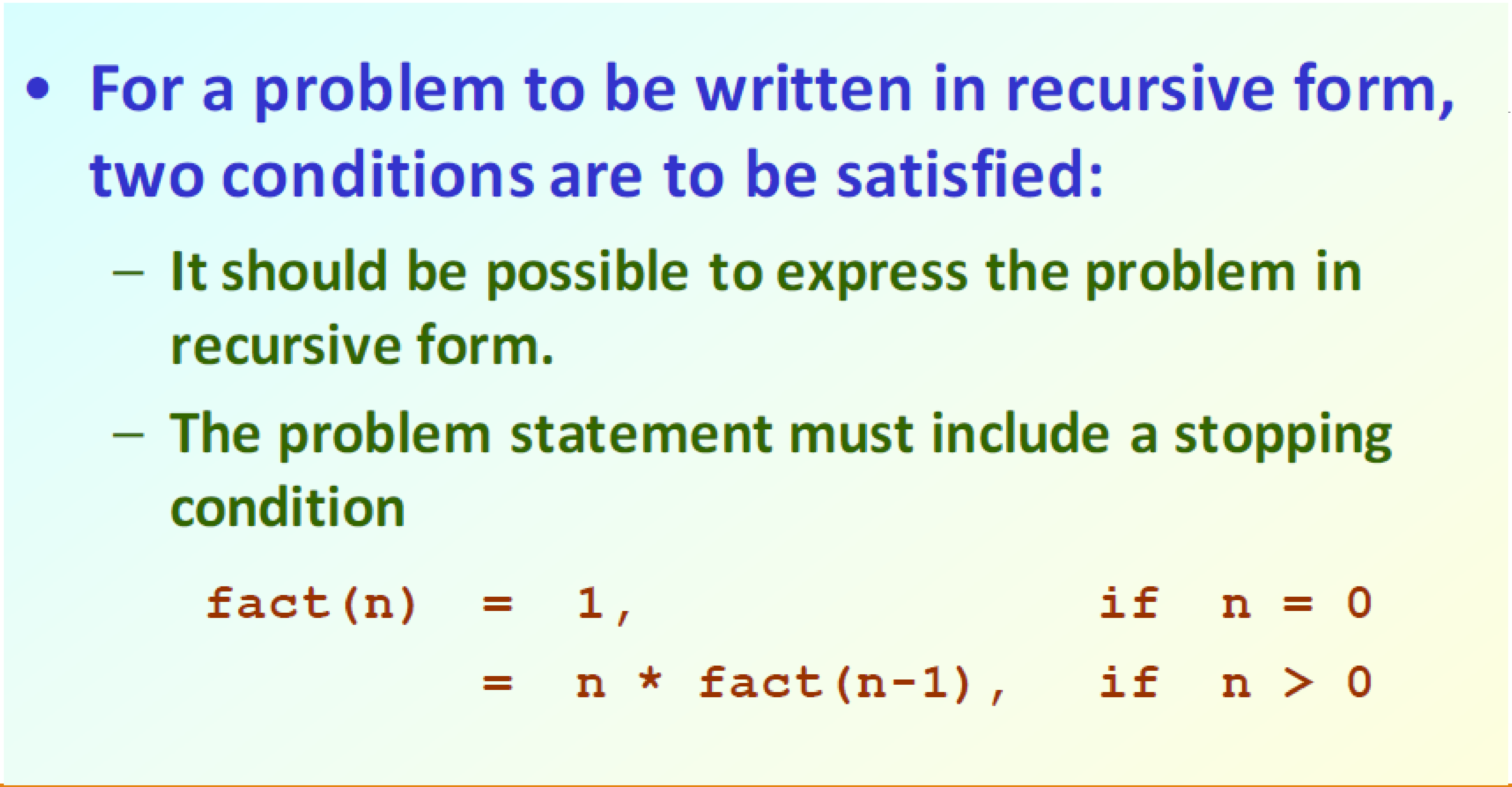
## The Factorial Function

1. This method does not use any explicit loops.
2. Repetition is achieved through repeated recursive invocations of the method.
3. The process is finite because each time the method is invoked, its argument is smaller by one, and when a base case is reached, no further recursive calls are made

## Recursion Definition

1. In computer science, recursion is a way of solving a computational problem where the whole solution depends on solutions to smaller instances of the same problem.
2. Recursion solves such recursive problems by using methods that call themselves from within their own code.
3. So, the process by which a method calls itself repeatedly is called recursion

# Recursion Definition



- For a problem to be written in recursive form, two conditions are to be satisfied:
  - It should be possible to express the problem in recursive form.
  - The problem statement must include a stopping condition

```
fact(n)  =  1,                 if  n = 0
         =  n * fact(n-1),     if  n > 0
```

# The Factorial Function

1. We illustrate the execution of a recursive method using a **recursion trace**.
2. Each entry of the trace corresponds to a recursive call.
3. Each new recursive method call is indicated by a downward arrow to a new invocation.
4. When the method returns, an arrow showing this return is drawn and the return value may be indicated alongside this arrow.

# Recursion – Program for Factorial

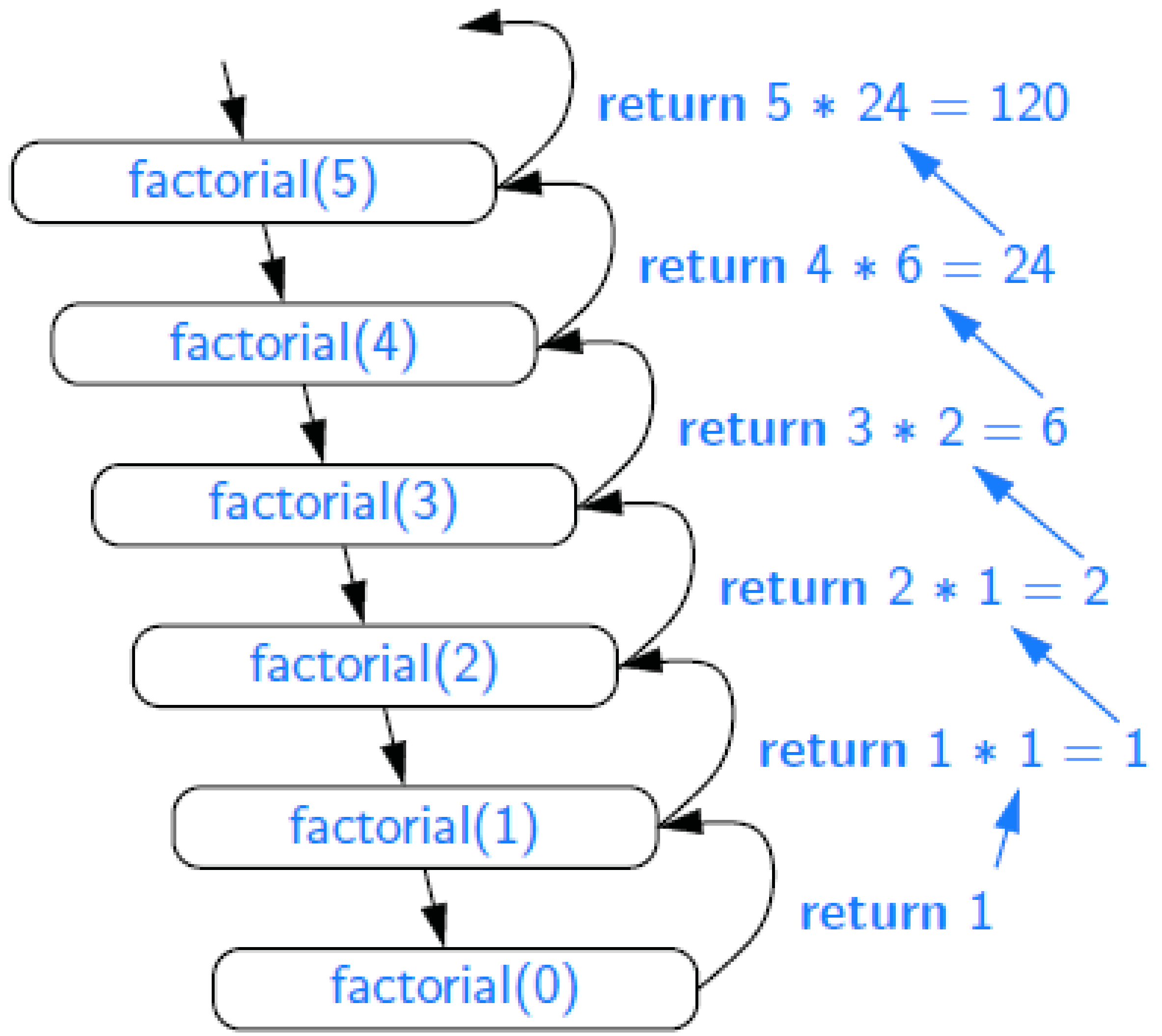


**Figure 5.1:** A recursion trace for the call factorial(5).

## The Factorial Function

1. A recursion trace closely mirrors a programming language’s execution of the recursion.
2. In Java, each time a method (recursive or otherwise) is called, a structure known as an **activation record or activation frame** is created to store information about the progress of that invocation of the method.
3. This frame stores the parameters and local variables specific to a given call of the method, and information about which command in the body of the method is currently executing.
4. When the execution of a method leads to a nested method call, the execution of the former call is suspended and its frame stores the place in the source code at which the flow of control should continue upon return of the nested call.

## The Factorial Function

1. A new frame is then created for the nested method call.
2. This process is used both in the standard case of one method calling a different method, or in the recursive case where a method invokes itself.
3. The key point is to have a separate frame for each active call.

## The Binary Search

1. In this section, we describe a classic recursive algorithm, ***binary search***, used to efficiently locate a target value within a sorted sequence of $n$ elements stored in an array. This is among the most important of computer algorithms, and it is the reason that we so often store data in sorted order (as in Figure 5.4).

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 5 | 7 | 8 | 9 | 12 | 14 | 17 | 19 | 22 | 25 | 27 | 28 | 33 | 37 |

## The Binary Search

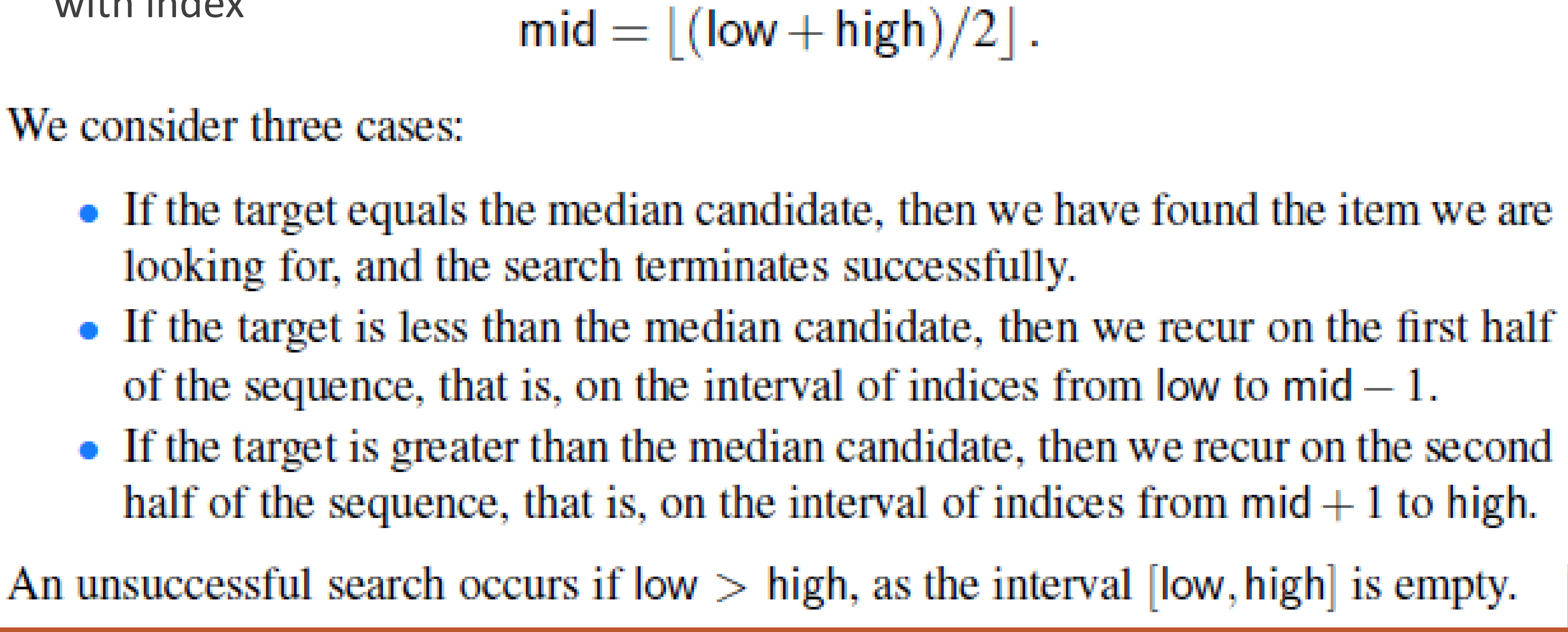
1. When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or
2. exhausting the data set.
3. This algorithm is known as linear search, or sequential search, and it runs in O(n) time (i.e., linear time) since every element is inspected in the worst case.

## The Binary Search

1. When the sequence is sorted and indexable, there is a more efficient algorithm.
2. If we consider an arbitrary element of the sequence with value v, we can be sure that all elements prior to that in the sequence have values less than or equal to v, and that all elements after that element in the sequence have values greater than or equal to v.
3. The algorithm maintains two parameters, low and high, such that all the candidate elements have index at least low and at most high. Initially, low = 0 and high = n−1.

## The Binary Search

1. ~~We then compare the target value to the median candidate, that is, the element~~ with index



$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor .$$

We consider three cases:

- If the target equals the median candidate, then we have found the item we are looking for, and the search terminates successfully.
- If the target is less than the median candidate, then we recur on the first half of the sequence, that is, on the interval of indices from low to $\text{mid} - 1$.
- If the target is greater than the median candidate, then we recur on the second half of the sequence, that is, on the interval of indices from $\text{mid} + 1$ to high.

An unsuccessful search occurs if $\text{low} > \text{high}$, as the interval $[\text{low}, \text{high}]$ is empty.

## The Binary Search

1. This algorithm is known as binary search.
2. Whereas sequential search runs in O(n) time, the more efficient binary search runs in O(log n) time.
3. This is a significant improvement, given that if n is 1 billion, log n is only 30.

## Recursion – Program for Binary Search

```
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                              // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                             // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);         // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);        // recur right of the middle
  }
}
```

## Recursion – Program for Binary Search

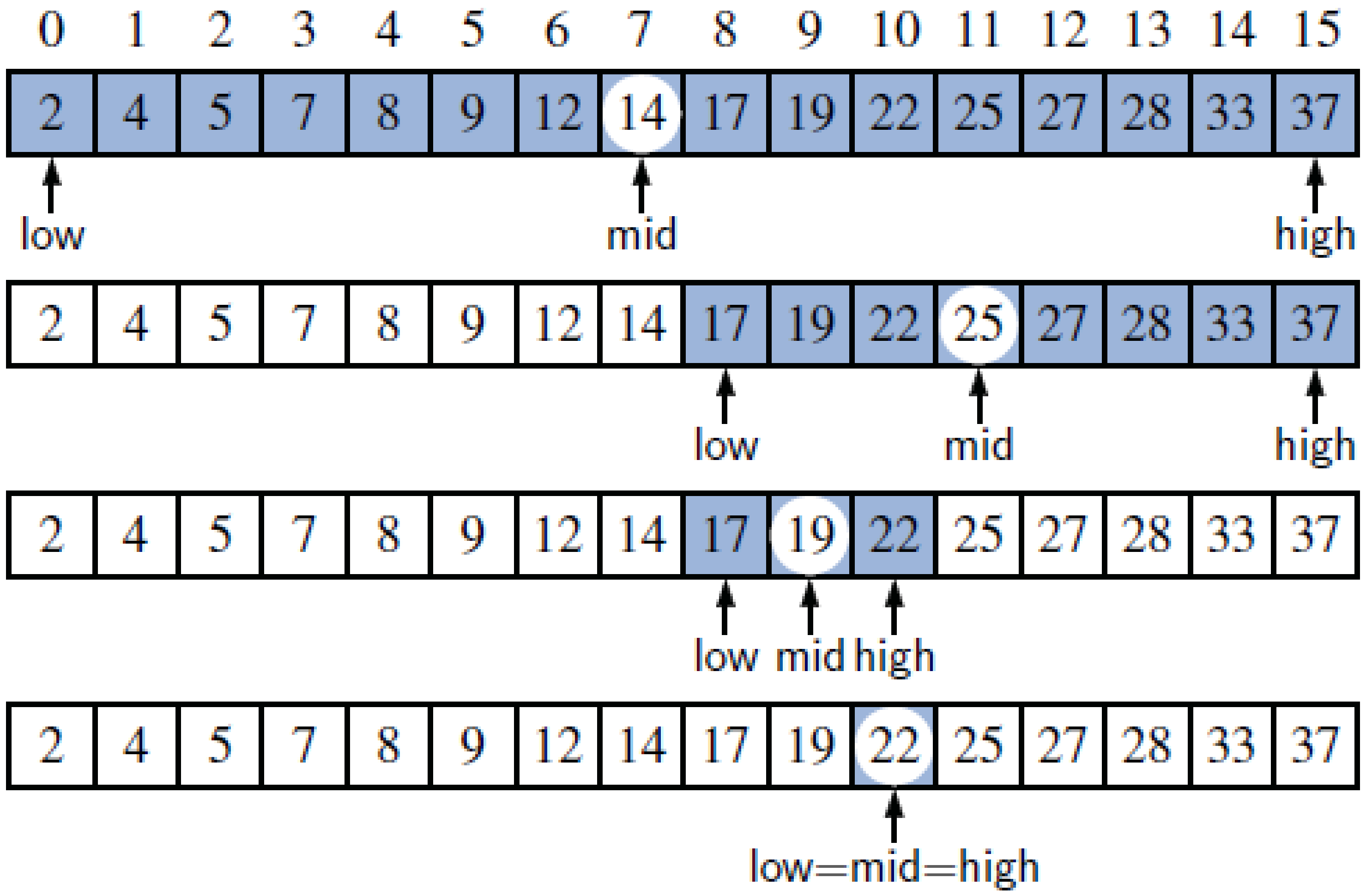
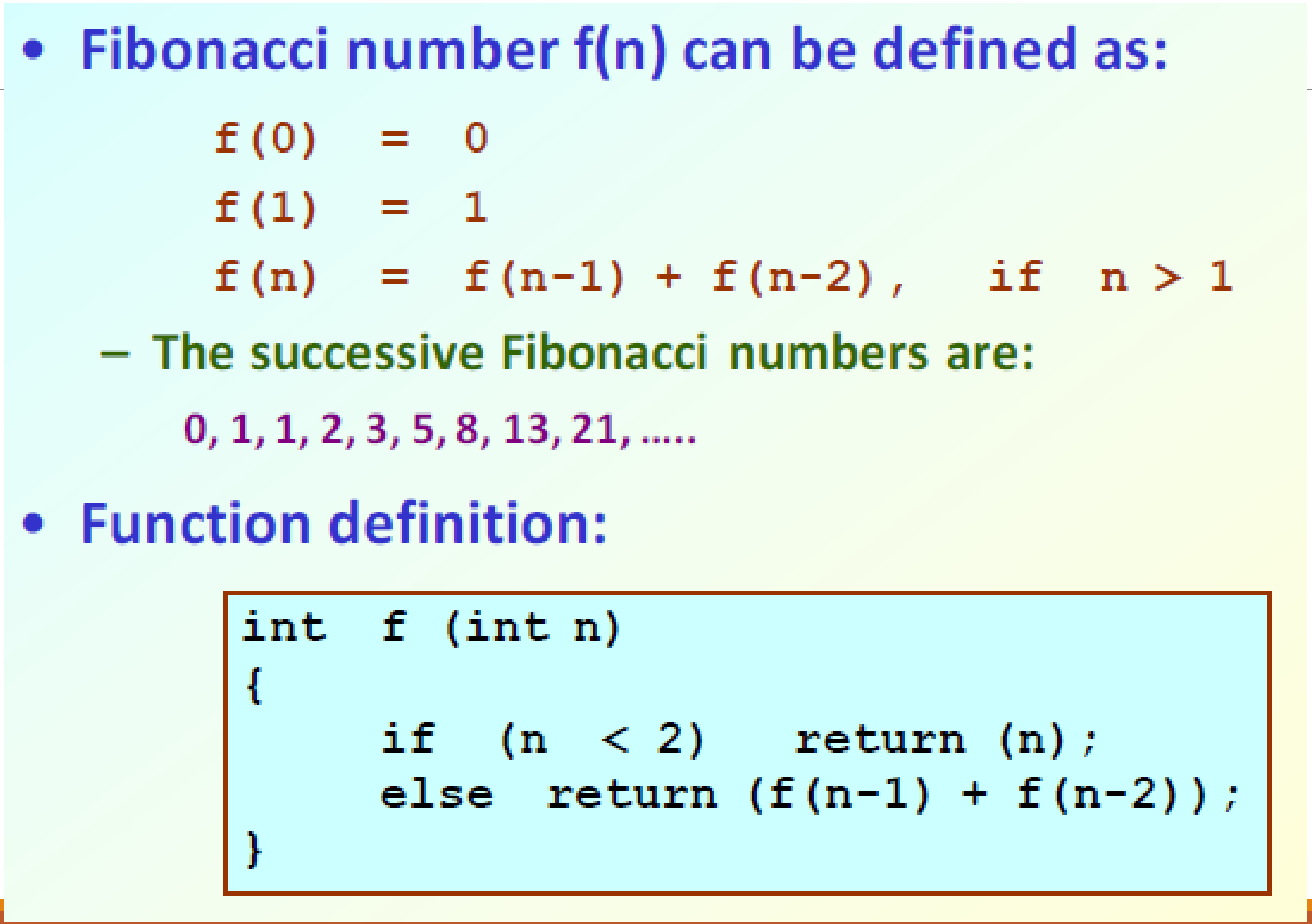


Figure 5.5: Example of a binary search for target value 22 on a sorted array with 16 elements.

# Recursion – Fibonacci Numbers



- Fibonacci number f(n) can be defined as:

```
f(0)  =  0
f(1)  =  1
f(n)  =  f(n-1) + f(n-2),   if  n > 1
```

  - The successive Fibonacci numbers are:

    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

- Function definition:

```
int  f (int n)
{
     if  (n  < 2)   return (n);
     else  return (f(n-1) + f(n-2));
}
```

# Thank You