2.3

Designing Algorithms and Recursion

ITX2010, CSX3003, IT2230

Data Structures and Algorithms,

Information Structures

Learning Objectives

Recursion

Students will be able to:

- Recognize goal and principal of Object-Oriented Programming

- Explain the history of developing a program

- Comprehend the class's terminologies

- Generate a Python class

- Explain the recursive function concept

- Describe how to solve a problem with the recursion

- Illustrate recursive function workflow

- Synthesize a recursive function

Chapter Outline

Object-Oriented Programming

1. **Object-Oriented Concept**
   1) Goal and Principle
   2) Program Development
   3) Class's Terminologies
   4) Python Class

2. **Recursion**
   1) Recursive Function
   2) Recursive Function Workflow
   3) Recursive Solution
   4) Runtime Stack
   5) Multiple Recursion
   6) Recursion Application

2

Recursion

Section 1:
Object-Oriented Concept

Goal and Principle

Object-Oriented Concept: Goal

- Robustness
  - Capability of handling unexpected input or tolerating with the input which is not explicitly specified
  - Ex: Although an input is type mismatched, it can run and return error message or find the possible closet outcome without any crash.
- Adaptability (Evaluability)
  - Portable to variety of hardware's and software's specifications
  - **Ex**: It can be run on its previous or new Operating Systems or platforms.
- Reusability
  - Reusable code can be adopted to the new program for optimized development time
  - **Ex**: Importing class to support some basic operations in another class

Goal and Principle

Object-Oriented Concept: Principle

- Modularity:
  - It consists of several different module that must properly work and correctly with others to serve a functional requirement.

- Abstraction:
  - Design and Implementation can be integrated in class concept.

- Encapsulation:
  - it gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions. [1]

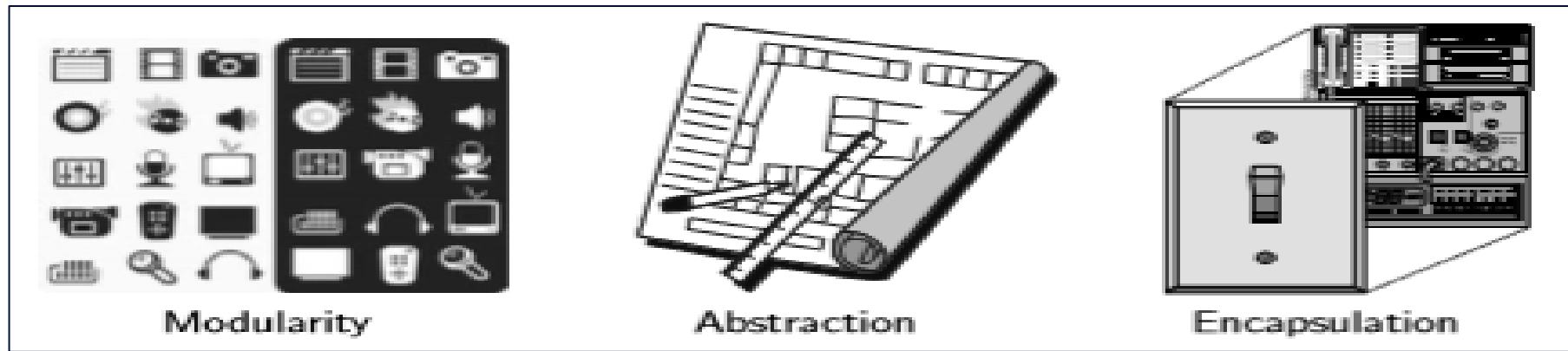# Goal and Principle

Object-Oriented Concept: Principle

Fig 2-1 General Data Structures [1]

Program Development

Object-Oriented Concept

- Non-structured Linear Programming (Spaghetti Code)
  - Program line of code run in sequence and were not prepared based on their responsibility.

- Modular Programming
  - Programs were organized in functions / method or module for serving a single purpose.

- Object-Oriented Programming
  - Functions and Data are developed within a template called class to define a behavior of class instance.

Class Definitions

Object-Oriented Concept

- Python is an object-oriented programming language.
- Class is a template of generating an object with the following components:
  1. Class name
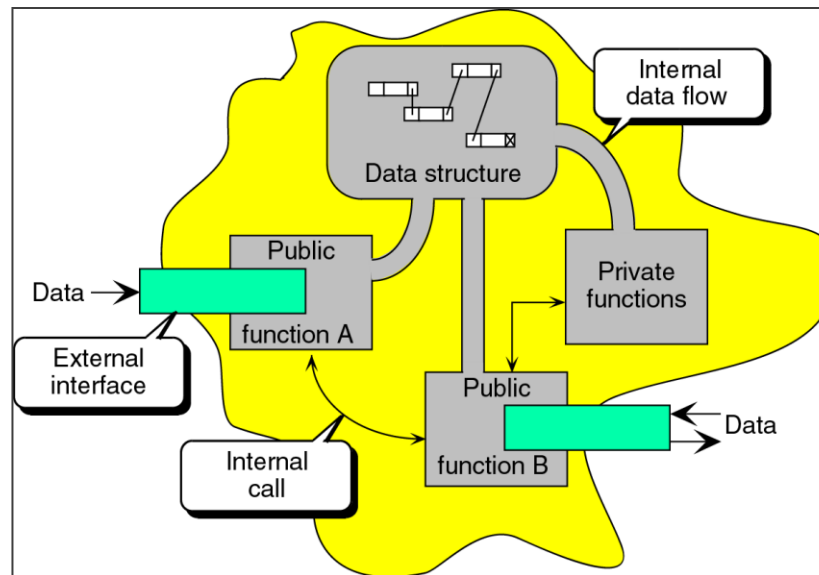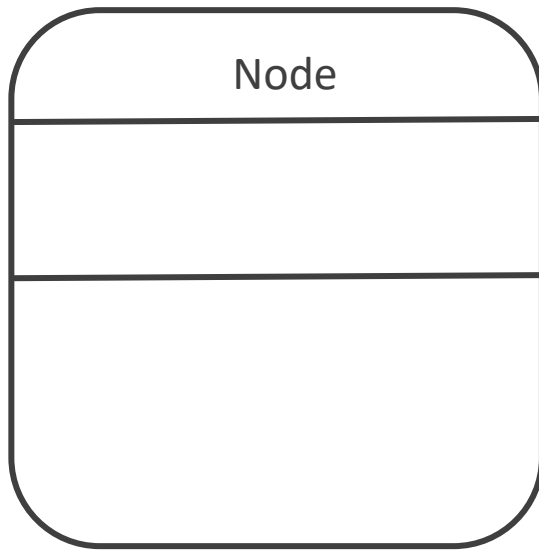 2. Class attribute
  3. Constructor
  4. Class method

Fig 2-2 General Data Structures [2]

Class Definitions

Object-Oriented Concept

1. Class name
   - Name of class or template name which must be assigned before generating its object or instance.


```
class Node:
    #Constructor

    #Class members
```
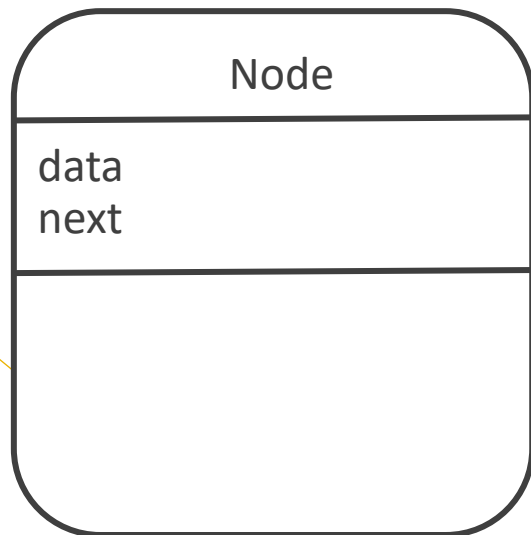
Fig 2-3 Class name

Class Definitions

Object-Oriented Concept

2. Class attribute
   - It describes the characteristics of class and can be implemented in term of variables.


```python
class Node:
    #Constructor
    def __init__(self,initData):
        #Class attibute
        self.data = initData
        self.next = None
```
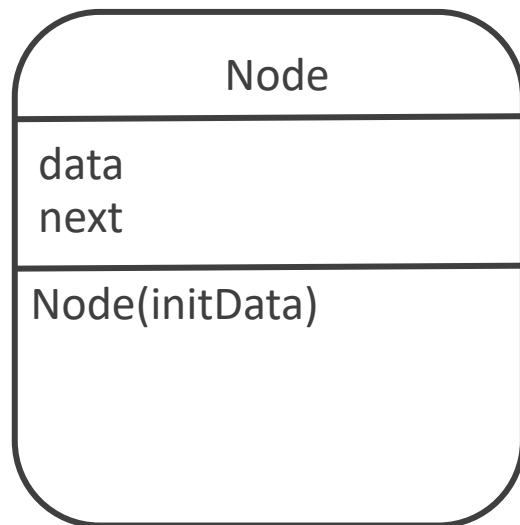
Fig 2-4 Class attribute

Class Definitions

Object-Oriented Concept

3. Constructor
   - Class method responses for initializing values of class attributes.


```
class Node:
    #Constructor
    def __init__(self,initData):
        #Class attibute
        self.data = initData
        self.next = None
```
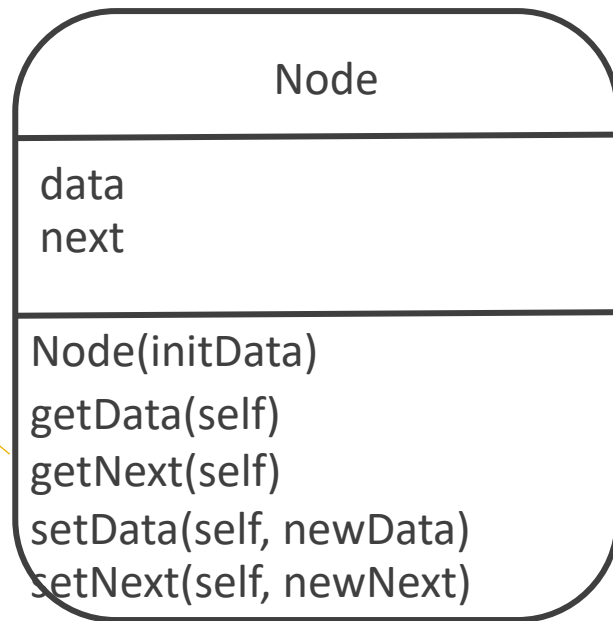
Fig 2-5 Class name

Class Definitions

Object-Oriented Concept

4. Class method
   - Class member or function define behavior or operation of class instance (object).

Node

data
next

Node(initData)

getData(self)

getNext(self)

setData(self, newData)

setNext(self, newNext)

```python
class Node:
    #Constructor
    #Class members
    def getData(self):
        return self.data
    def getNext(self):
        return self.next
    def setData(self,newdata):
        self.data = newdata
    def setNext(self,newnext):
        self.next = newnext
```

Fig 2-6 Class method

13

```python
1  #Create Node Class
2
3  class Node:
4      #Constructor
5      def __init__(self,initData):
6          #Class attibute
7          self.data = initData
8          self.next = None
9      #Class members
10     def getData(self):
11         return self.data
12     def getNext(self):
13         return self.next
14     def setData(self,newdata):
15         self.data = newdata
16     def setNext(self,newnext):
17         self.next = newnext
18  #Create an object of class Node
19  myNode = Node(10)
20  print("\n")
21  #Call the class's method getData
22  print("Print node data : ", myNode.getData())
```

Print node data : 10

2

Recursion

Section 2: Recursion

Recursion Function

What is Recursion?

- **Divide-and-conquer** method:
  - Break the problem into several subproblems that are similar to the original problem
  - Solve the subproblems recursively
  - Combine these solutions to create a solution to the original problem
- Perform three characteristic steps:
  1. **Divide** the problem into one or more subproblems
  2. **Conquer** the subproblems by solving them recursively
  3. **Combine** the problem solutions to form a final solution

Recursion Function

What is Recursion?

- **Recursion** is a process for solving problems by subdividing a larger problem into smaller cases of the problem itself and then solving the smaller, more trivial parts. **[3]**

- It recurse (call itself) one or more times to handle closely related problems. **[7]**

- This algorithm follow the **"divide-and-conquer"** method. **[7]**

Recursion Function

What is Recursion?

- **Recursive Function** is a function that calls itself, directly or indirectly. [4]

- <u>**Ex:**</u> A mathematic example of Fibonacci

- The recursive function consists of two parts:
  1. Termination condition
  2. Function body

$$F(n) = \begin{cases} 1, n = 1,2 \\ F(n-1) + F(n-2), others \end{cases}$$

Fig 2-7 Fibonacci function [3]

Recursion Function

Components

1. **Termination condition (Base case)**
   - A recursive function always contains one or more terminating condition.
   - A condition which recursive function is processing a simple case and do not call itself.

2. **Function body (Recursive case)** – including recursive expansion
   - The main logic of the function contains in the body of the function.
   - It contains the recursion expansion statement that in turn calls the function itself.
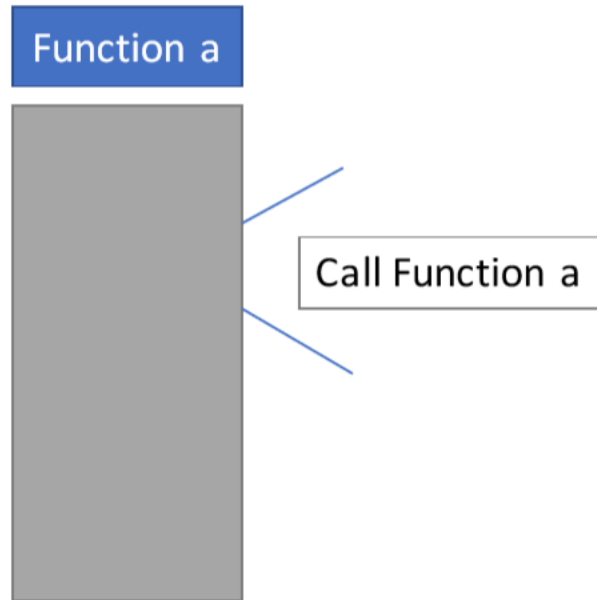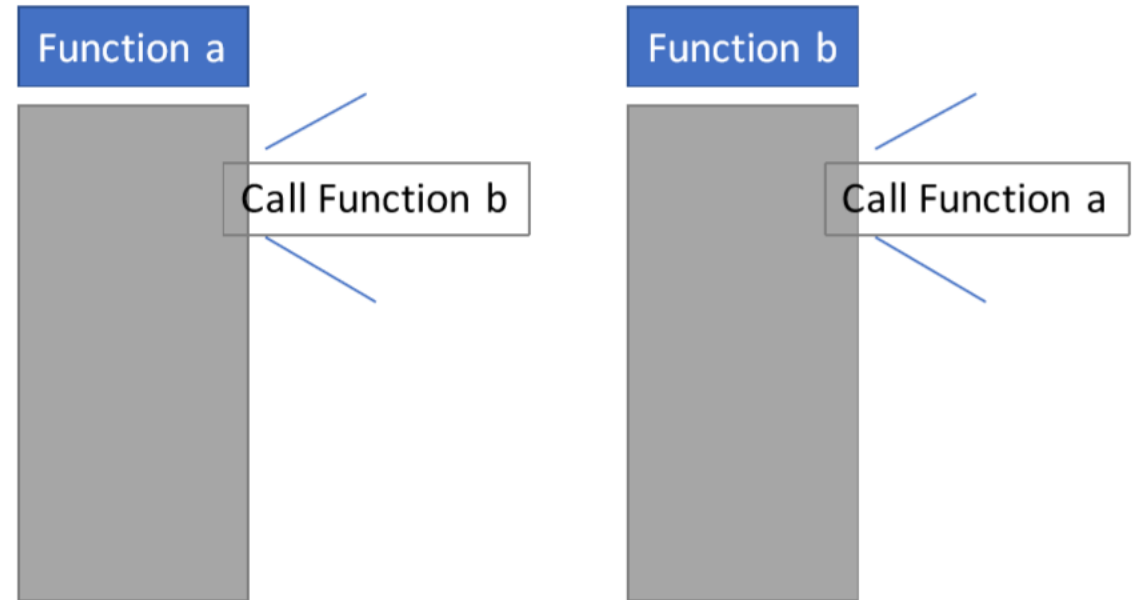
Recursion Function

Properties

1. It must have a terminate condition (base case).
   - Without the terminate condition, the recursive function will forever run and consume all stack memory.
2. It must call itself which contain a recursive case.
3. It must change its state until toward to the terminate condition (base case).

- **Note that**:
  - Property 1 and 3 guarantee that the recursive function can stop.
  - Property 2 divides the problem into smaller pieces (Divide and Conquer).

Recursion Function

Properties

Fig 2-8 Recursive functions [3]
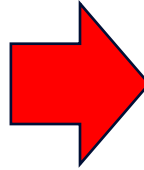
Recursion Function Workflow

- Given a print function:

Fig 2-9 Example of print function [3]

Recursion Function Work Flow
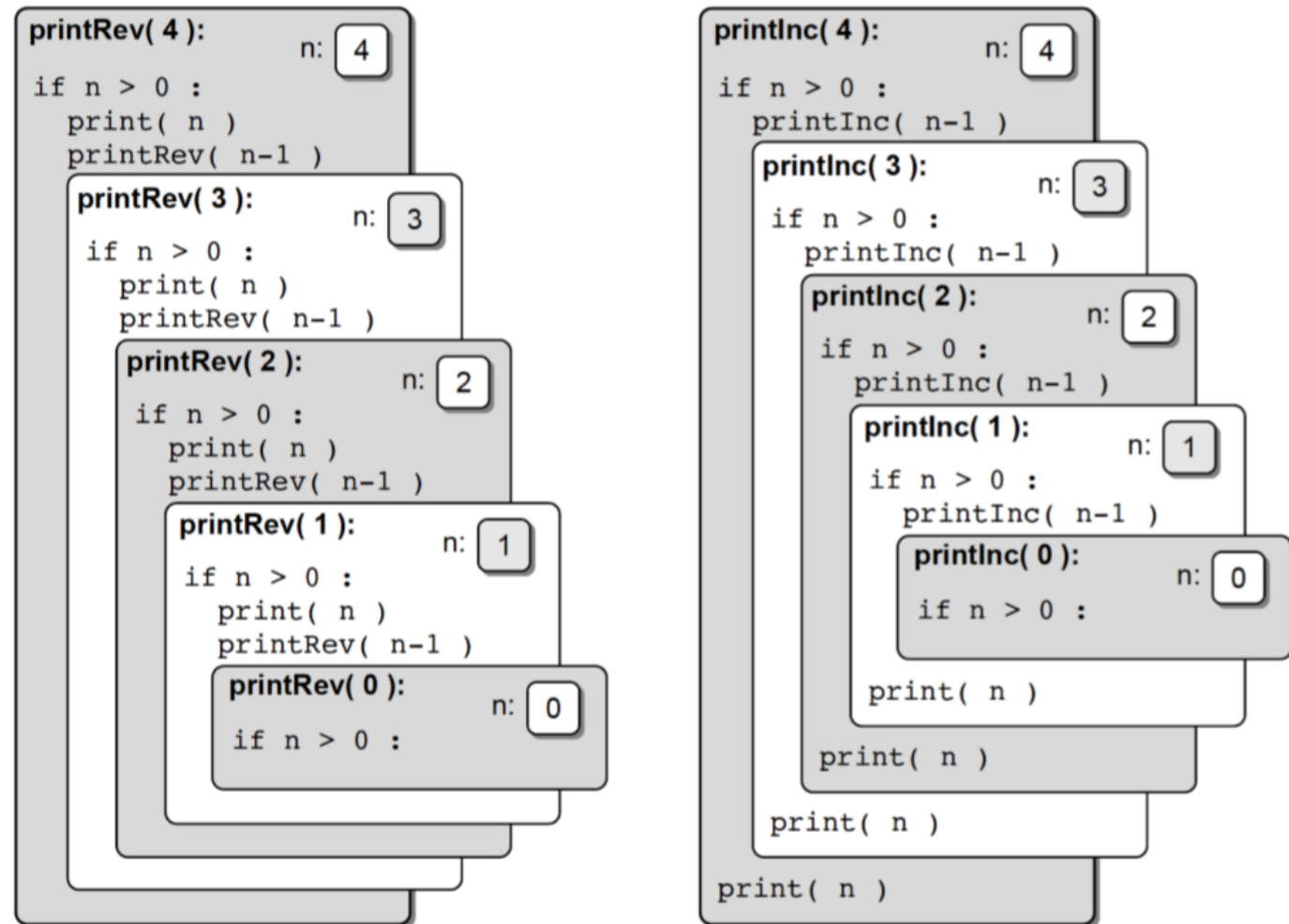
- What is the output of each function?

Fig 2-10 Print function work flow [3]

Recursion Solution

- A recursive solution can:
  1. Subdivides a problem into smaller version of itself.
  2. Find a based case
  3. Find a recursion case

- **Ex**: Factorial of n (n!)
- A recursive solution can:
  1. Subdivides a problem into smaller version of itself.

     5 x 4 x 3 x 2 x 1
  2. Find a based case

     n =0, n! = 1
  3. Find a recursion case

     n! = n(n-1)!
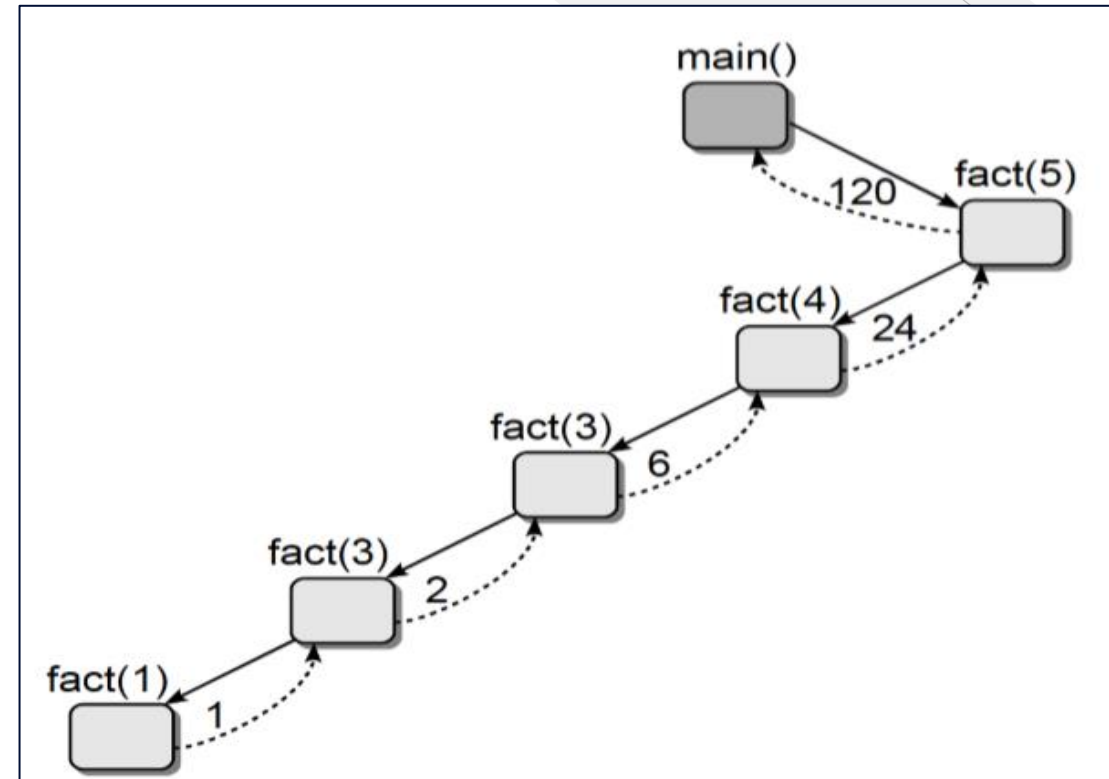

```
1   # Compute n!
2   def fact( n ):
3       assert n >= 0, "Factorial not defined for negative values."
4       if n < 2 :
5           return 1
6       else :
7           return n * fact(n - 1)
```

Runtime Stack

- The base case will be called last but must be firstly computed.

```
def main():
  y = fact( 2 )
main()
```
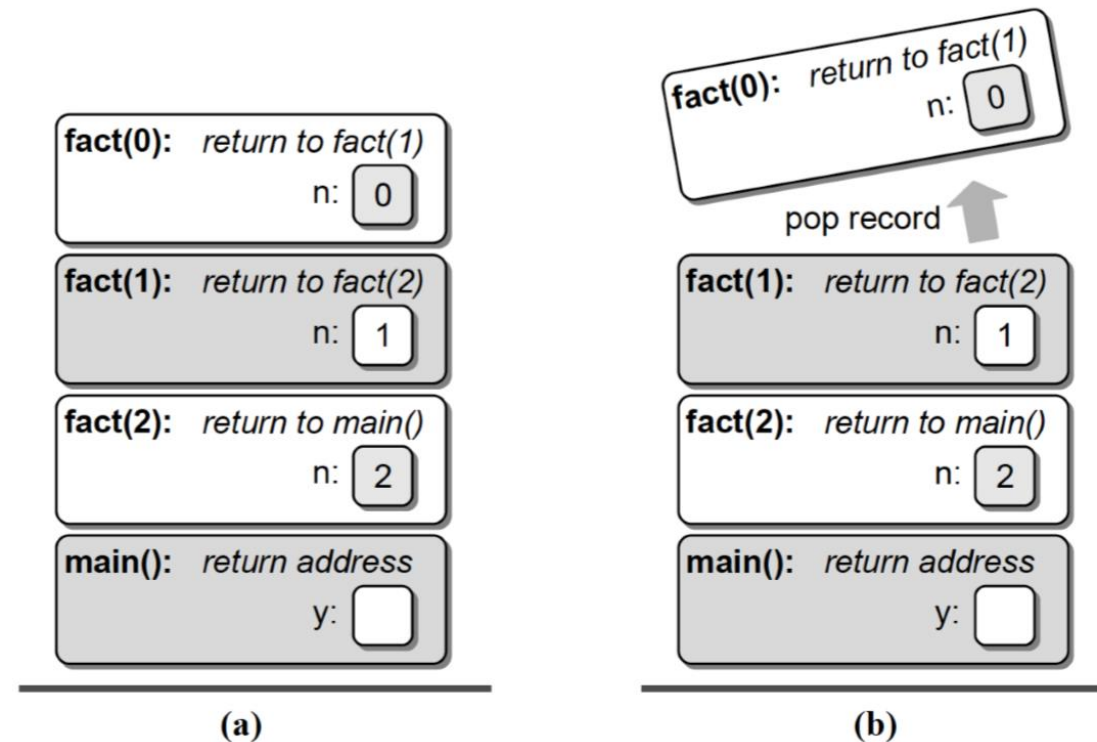

Fig 2-11 Stack runtime [3]

Multiple Recursion

- Some problem - like Fibonacci, require multiple recursive function.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

$$\text{fib}(n) = \begin{cases} 1, n = 1,2 \\ fib(n-1) + fib(n-2), others \end{cases}$$

```
fib(n): //assuming n >= 1
    if n=1 or n=2:
        return 1
    else:
        return fib(n - 1)+fib(n-2)
```

Fig 2-12 Example of Fibonacci recursive function [3]
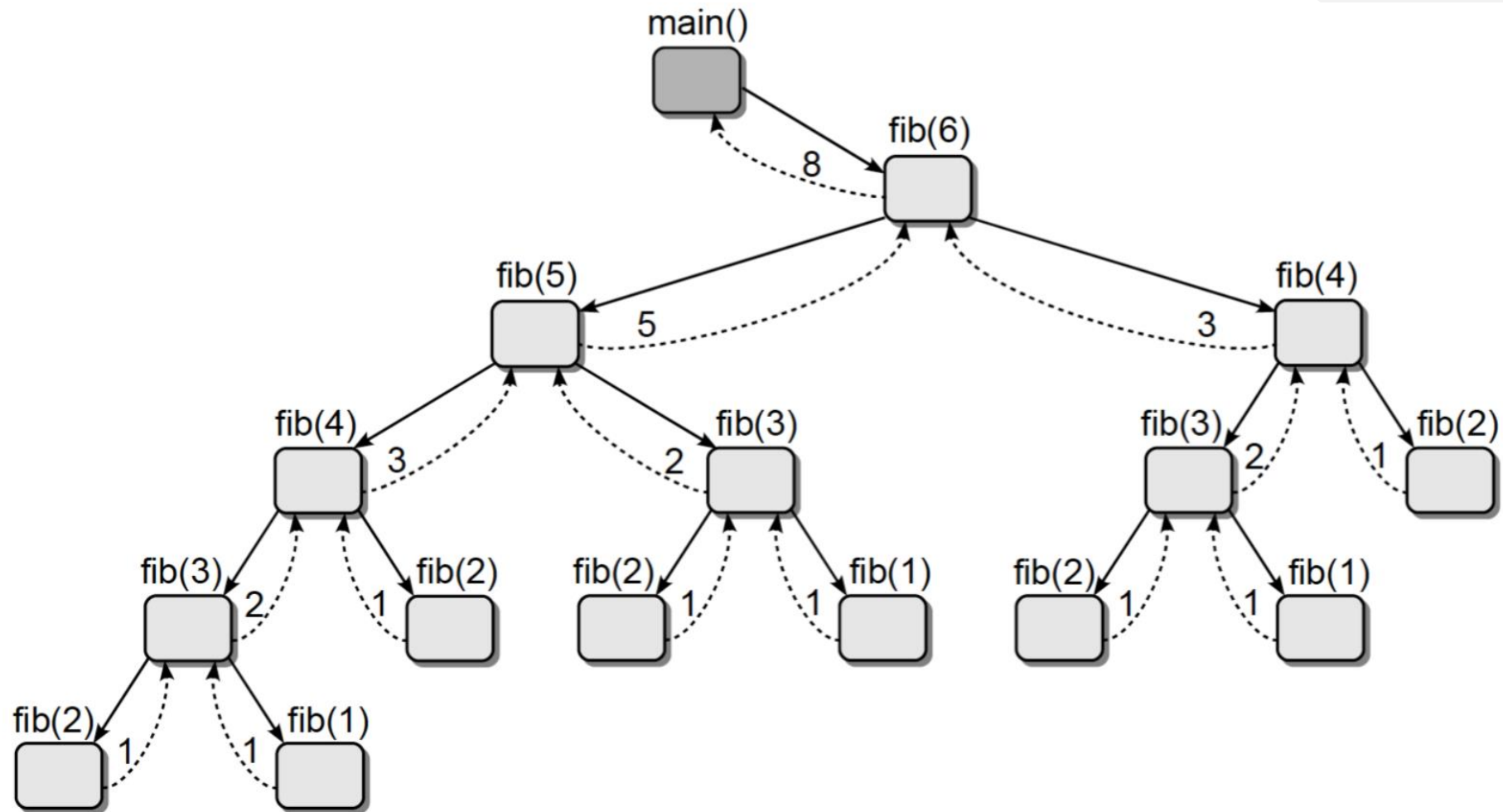
27

Multiple Recursion

Fig 2-13 Logical view of executing Fibonacci recursive function [3]

Recursion Applications

- Many application can be solved by using recursion
    1. Merge sort algorithm
    2. Binary search
    3. Towers of Hanoi
    4. Tic-Tac-Toe

Recursion Applications

[7]

Merge sort algorithm

MERGE(A, p, q, r)

1 $n_L = q - p + 1$ // length of $A[p : q]$

2 $n_R = r - q$ // length of $A[q + 1 : r]$

3 let $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$ be new arrays

4 **for** $i = 0$ **to** $n_L - 1$ // copy $A[p : q]$ into $L[0 : n_L - 1]$

5      $L[i] = A[p + i]$

6 **for** $j = 0$ **to** $n_R - 1$ // copy $A[q + 1 : r]$ into $R[0 : n_R - 1]$

7      $R[j] = A[q + j + 1]$

8 $i = 0$ // i indexes the smallest remaining element in L

9 $j = 0$ // j indexes the smallest remaining element in R

10 $k = p$ // k indexes the location in A to fill

30

Recursion Applications

Merge sort algorithm

11 **//** As long as each of the arrays L and R contains an unmerged element,
  //   copy the smallest unmerged element back into $A[p : r]$.
12 **while** $i < n_L$ and $j < n_R$
13   **if** $L[i] \leq R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16   **else** $A[k] = R[j]$
17          $j = j + 1$
18      $k = k + 1$
19 **//** Having gone through one of L and R entirely, copy the
   //   remainder of the other to the end of $A[p : r]$.

Recursion Applications

Merge sort algorithm

19 // Having gone through one of L and R entirely, copy the
//   remainder of the other to the end of $A[p : r]$.
20 **while** $i < n_L$

21      $A[k] = L[i]$

22      $i = i + 1$

23      $k = k + 1$

24 **while** $j < n_R$
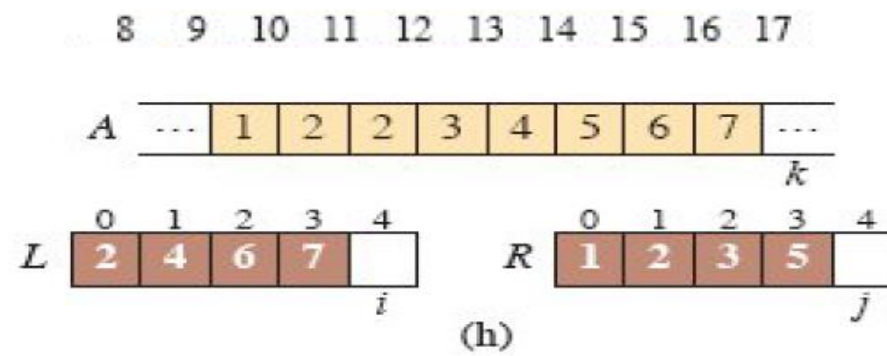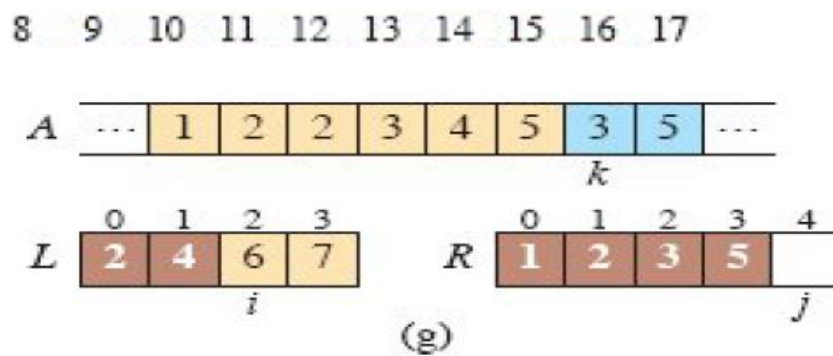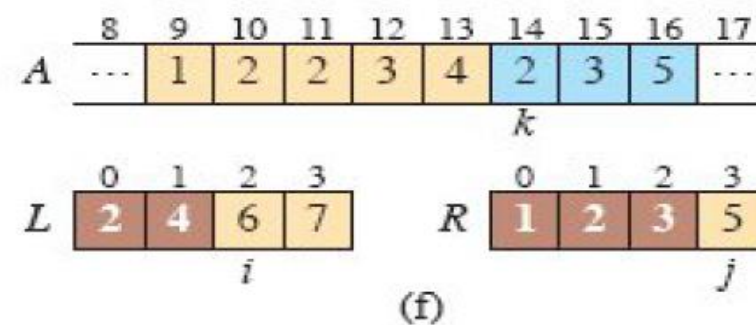
25      $A[k] = R[j]$

26      $j = j + 1$

27      $k = k + 1$
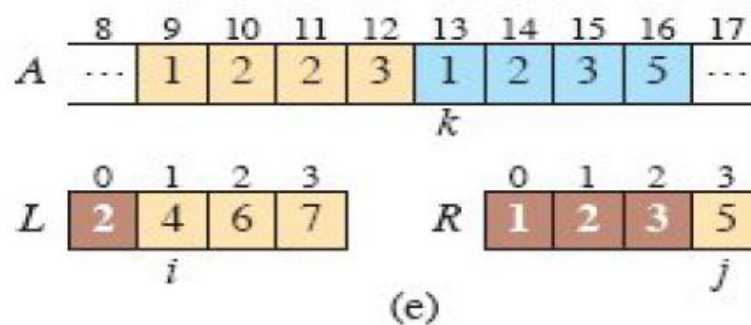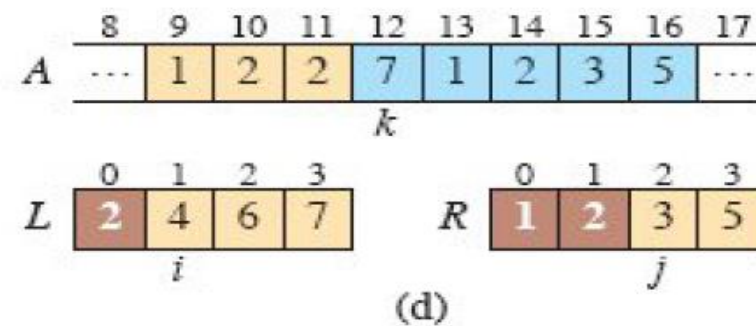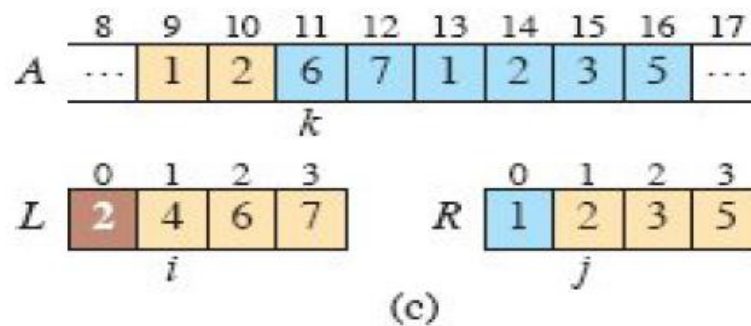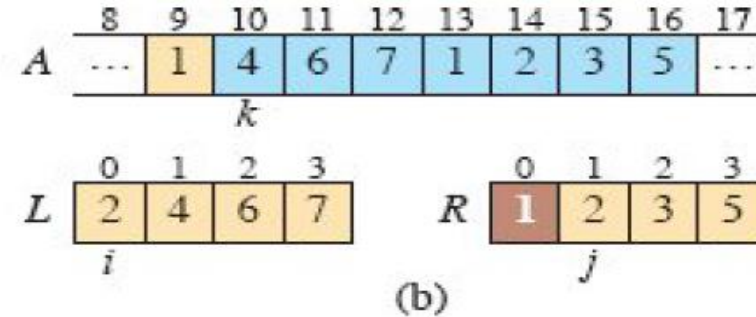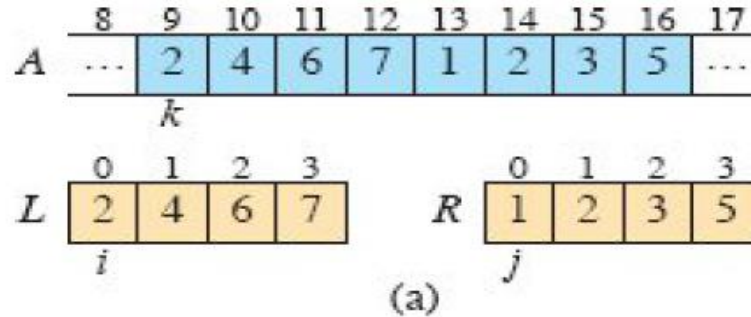
Recursion Applications

Merge sort algorithm

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0, \\ D(n) + aT(n/b) + C(n) & \text{otherwise}. \end{cases}$$

MERGE-SORT(A, p, r)

1 **if** $p \geq r$ *// zero or one element?*
2   **return**
3 $q = \lfloor (p + r)/2 \rfloor$ *// midpoint of $A[p : r]$*
4 MERGE-SORT(A, p, q)   *// recursively sort $A[p : q]$*
5 MERGE-SORT($A, q + 1, r$)   *// recursively sort $A[q + 1 : r]$*
6 *// Merge $A[p : q]$ and $A[q + 1 : r]$ into $A[p : r]$.*
7 MERGE(A, p, q, r)
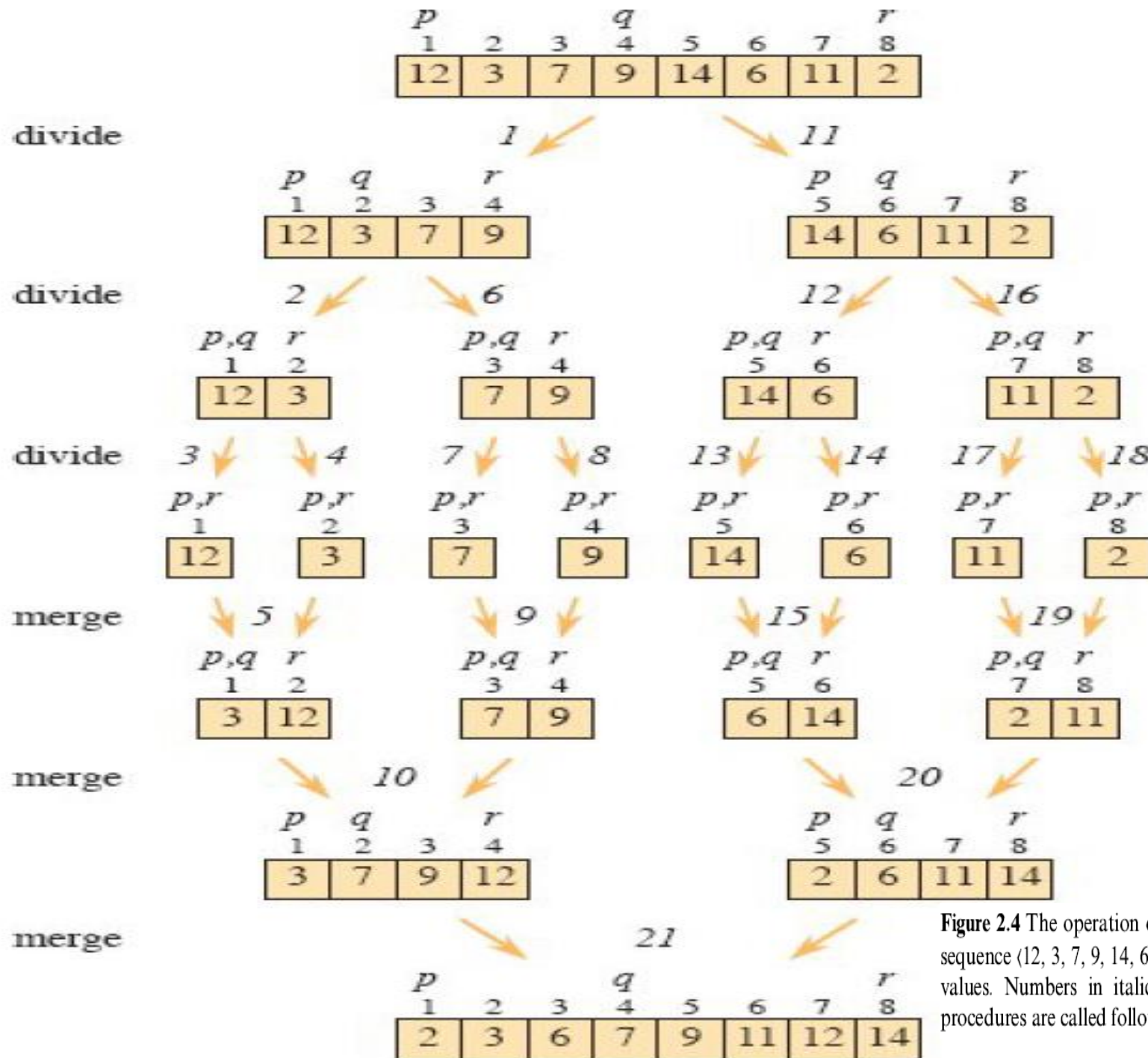
Figure 2.4 The operation of merge sort on the array A with length 8 that initially contains the sequence $\langle 12, 3, 7, 9, 14, 6, 11, 2 \rangle$. The indices p, q, and r into each subarray appear above their values. Numbers in italics indicate the order in which the MERGE-SORT and MERGE procedures are called following the initial call of MERGE-SORT(A, 1, 8).

Analysis of merge sort

Here's how to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: Recursively solving two subproblems, each of size $n/2$, contributes $2T(n/2)$ to the running time (ignoring the floors and ceilings, as we discussed).

Combine: Since the MERGE procedure on an n-element subarray takes $\Theta(n)$ time, we have $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n. That is, it is roughly proportional to n when n is large, and so merge sort's dividing and combining times together are $\Theta(n)$. Adding $\Theta(n)$ to the $2T(n/2)$ term from the conquer step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = 2T(n/2) + \Theta(n). \tag{2.3}$$

Figure 2.5 How to construct a recursion tree for the recurrence (2.4). Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels. Each level above the leaves contributes a total cost of $c_2 n$, and the leaf level contributes $c_1 n$. The total cost, therefore, is $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.

To compute the total cost represented by the recurrence (2.4), simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels. The levels above the leaves each cost $c_2 n$, and the leaf level costs $c_1 n$, for a total cost of $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.

Recursion Applications

Binary Search

- The problem of searching can be divided into smaller version of itself by:
  - Each splitting, the list will be cut into two parts.
  - This will be repeated until the target key is found or there is no more item to be searched.

Recursion Applications

Binary Search

```
recBinarySearch(target, A, first, last):

    //if the sequence cannot be subdivided further, we are done.
    if first > last:
        return False

    else:
        mid <-- (last + first)/2
        if A[mid] = target:
            return True

        else if target < A[mid]:
            return recBinarySearch(target, A, first, mid-1)

        else:
            return recBinarySearch(target, A, mid+1, last)
```
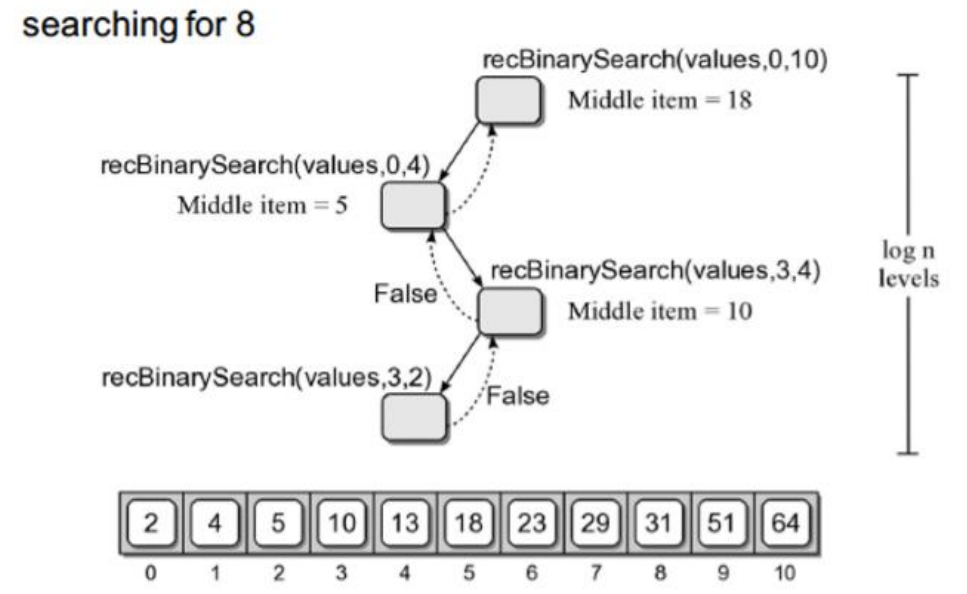

searching for 8

recBinarySearch(values,0,10)
Middle item = 18

recBinarySearch(values,0,4)
Middle item = 5

recBinarySearch(values,3,4)
Middle item = 10

recBinarySearch(values,3,2)
False

False

log n levels

| 2 | 4 | 5 | 10 | 13 | 18 | 23 | 29 | 31 | 51 | 64 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Fig 2-14 Binary search as recursive function [3]

References

Texts | Integrated Development Environment (IDE)

[1] Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Willy & Sons Inc., 2013

[2] Data Structures: A Pseudocode Approach with C++, Richard F. Gilberg and Behrouz A. Forouzan, Brooks/Cole, 2001

[3] Data Structures and Algorithms Using Python, Rance D. Necaise, John Winley & Sons, Inc., 2011.

[4] Problem Solving in Data Structures & Algorithms Using Python: Programming Interview Guide, 1st Edition, Hermant Jain, Thiftbooks, March 2017.

[5] https://trinket.io/features/python3

[6] https://colab.research.google.com/

[7] Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein,, Fourth Edition, The MIT Press, 2022.