VNUHCM - UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

University of Science – VNU-HCM
Faculty of Information Science
Department of Computer Science
MTH083 - Advanced Programming for Artificial Intelligence

Slot 08- Recursion

Advisor:

Dr. Nguyễn Tiến Huy

Dr. Lê Thanh Tùng

Content

Introduction

- Recursion is an extremely powerful problem-solving technique
- Recursion is encountered not only in mathematics, but also in daily life
- An object is said to be recursive if it is defined in terms of a smaller version of itself

Introduction

- This technique provides a way to break complicated problems down into simple problems which are easier to solve
- For example, we can define the operation "find your way home" as:
  - If you are at home, stop moving
  - Take one step toward home
  - “find your way home"

Introduction

- All recursive algorithms must have the following:
  - Base Case (i.e., when to stop) - Halting Condition
  - Work toward Base Case
    - Recursive Call (i.e., call ourselves)

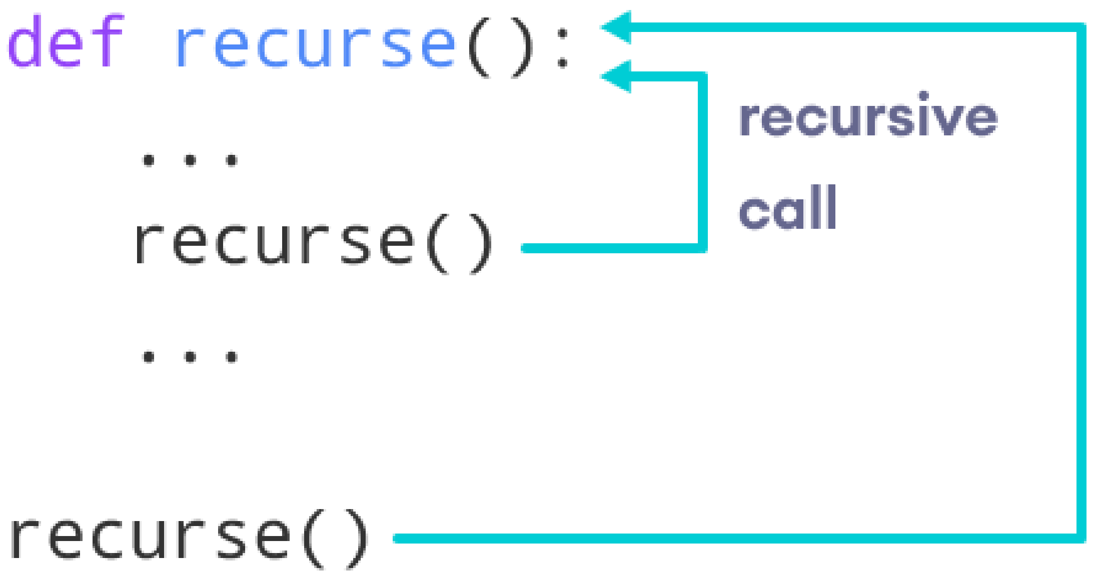
Recursion Example

- Natural numbers
  - 0 is a natural number
  - The successor of a natural number is a natural number
- Fractional:
  - $0! = 1$
  - $n! = n*(n - 1)!$
- Fibonacci numbers:
  - $F(0) = 0, F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

Recursion Example

- Fibonacci numbers:
  - Base case:
    - F(0) = 0, F(1) = 1
  - Work toward base case:
    - F(n) = F(n-1) + F(n-2)
  - Recursive Call: ??

- In Python, we know that a function can call other functions
- It is even possible for the **function to call itself**
- These types of construct are termed as recursive functions



- If the halting condition is not well-defined, the recursion function will be looped infinitely.

```python
def greet():
    print("Hello")
    greet()
```

- We should determine the base case carefully

```python
def greet(n):
    if n == 0: return
    print("Hello")
    greet(n- 1)
```

Recursion Function

- The structure of recursive functions is typically like the following

```
RecursiveFunction():
    if (test for simple case):
        Compute the solution without recursion
        Stop function
    else:
        Break the problem into subproblems of the same form
        Call RecursiveFunction() on each subproblem
        Reassamble the results of the subproblems
```

- 3 “must have” of a recursive algorithm
  - Your code must have a case for **all valid inputs**
  - You must have a **base case** with **no recursive calls**.
  - When you make a recursive case, it should be to a **simpler instance** and make forward progress towards the base case

Recursion Function

- Example: Calculating n!
  - Definition 1:
    - If n = 0: $n! = 1$
    - If n > 0: $\mathrm{n}! = 1 \times 2 \times 3 \times \cdots \times n$

  $\rightarrow$ Can not use recursion
  - Definition 2:
    - If n = 0: $n! = 1$
    - If n > 0: $\mathrm{n}! = (\mathrm{n}-1)! \times n$

  $\rightarrow$ Can use recursion

Recursion Function

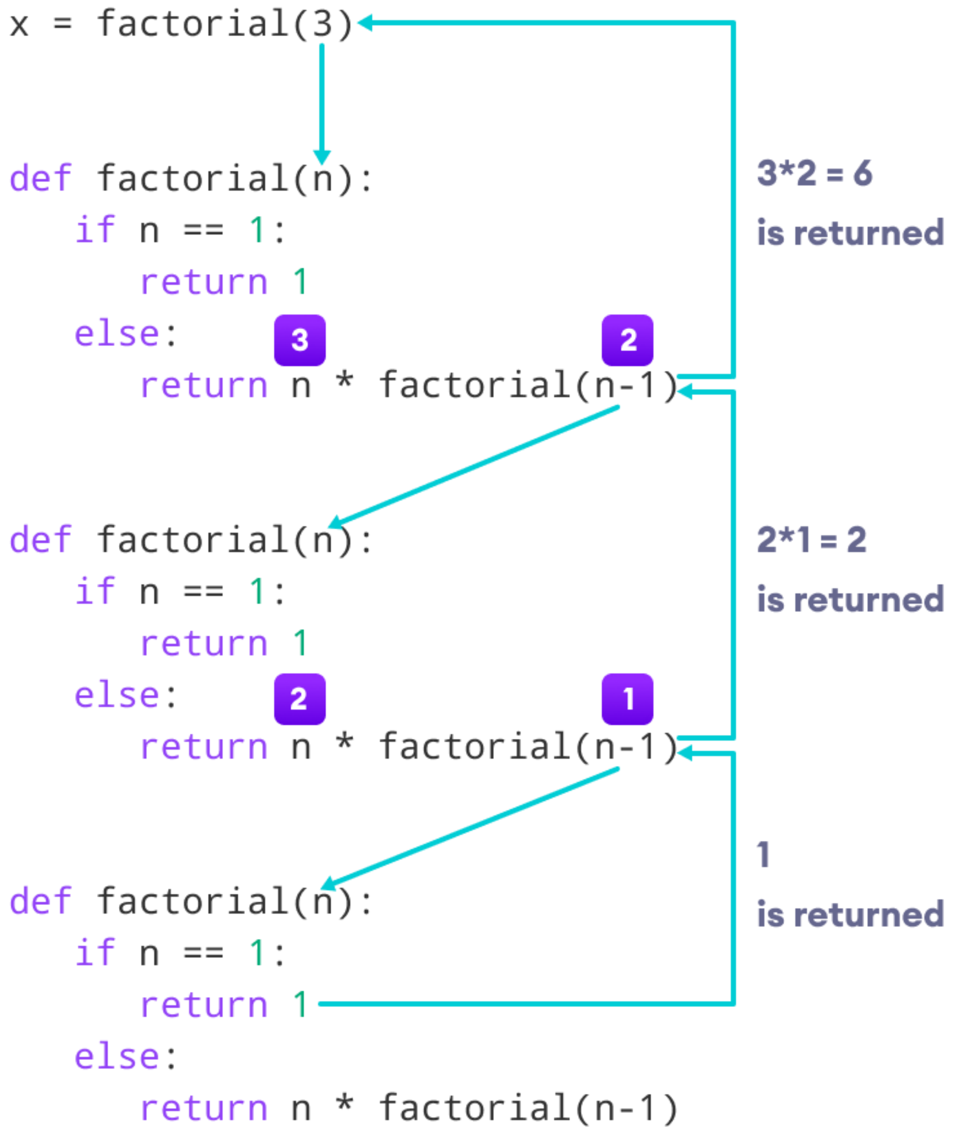
- Example: Calculating n!
  - Definition 2:
    - If n = 0: $n! = 1$
    - If n > 0: $\mathrm{n}! = (\mathrm{n}-1)! \times n$

```python
def calcFactorial(n: int)-> int:
    if n == 0: # base case
        return 1
    else:
        # recursion call
        return n*calcFactorial(n - 1)
```

Recursion Progress

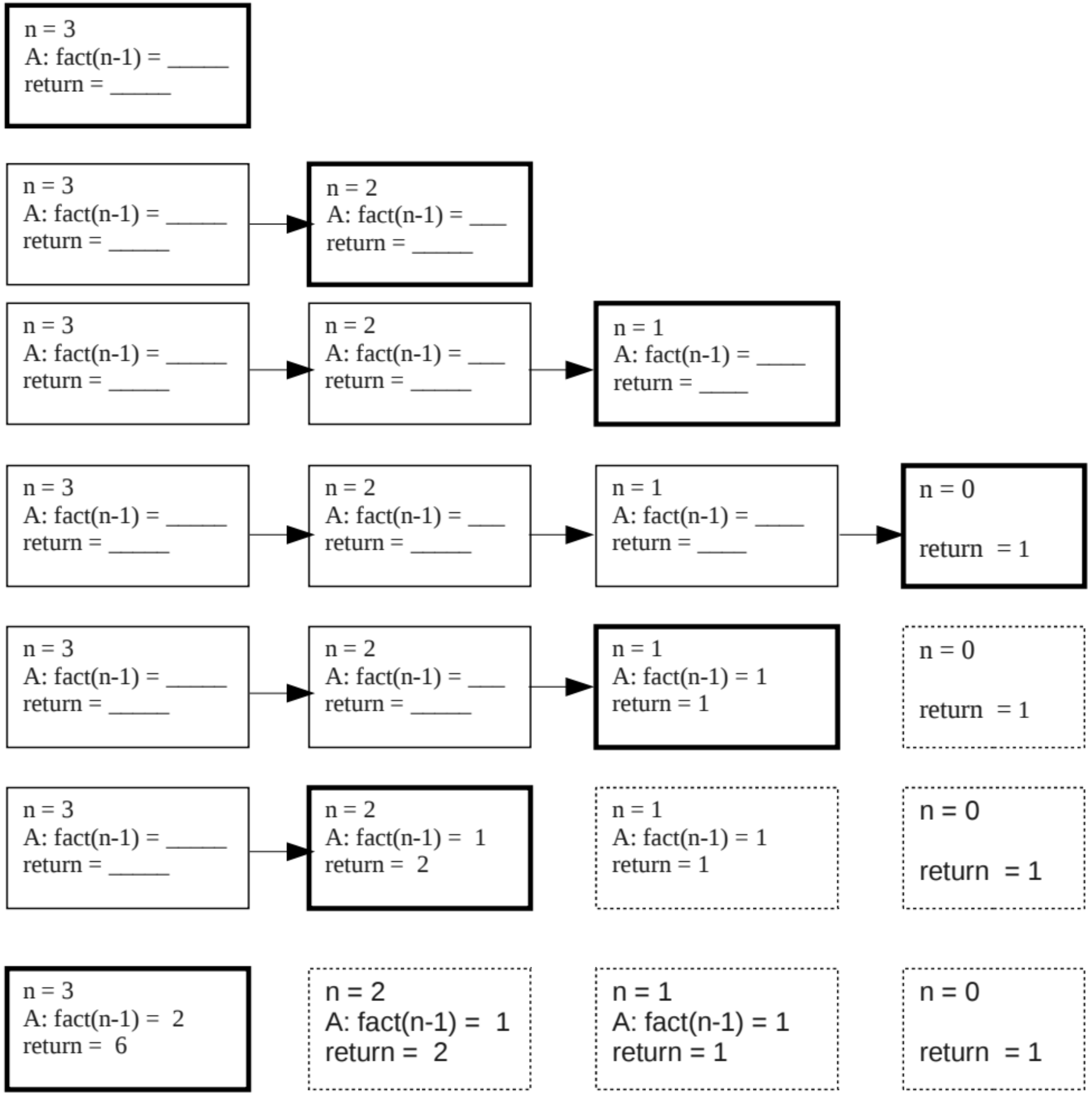

```python
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```



Tracing Recursion

- Read more: https://codeahoy.com/learn/recursion/ch8/



Advantages/Disadvantages

- **Advantages:**
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.
- **Disadvantages:**
  - Sometimes the logic behind recursion is hard to follow through.
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
  - Recursive functions are hard to debug.

Type of Recursion


```python
#direct recursion
def fact(x: int) -> int:
    if x == 0:
        return 1
    else:
        return x*fact(x - 1)
```

```python
#indirect recursion
def isOdd(x: int) -> bool:
    return not(isEven(x))

def isEven(x: int) -> bool:
    if x == 0:
        return True
    else:
        return isOdd(x - 1)
```

- **Tail Recursion:** A recursive call is said to be tail-recursive if it is the last statement to be executed inside the function

```python
#Tail Recursion
def printN(n: int):
    print(n)
    if n > 0:
        printN(n - 1)
```

```python
#Non-tail Recursion
def printN_NonTail(n: int):
    if n > 0:
        printN(n - 1)
    print(n)
```

Type of Recursion

```python
#Linear recursion
def fact(x: int) -> int:
    if x == 0:
        return 1
    else:
        return x*fact(x - 1)
```

```python
#Binary recursion
def fibo(x: int) -> int:
    if x < 2:
        return x
    else:
        return fibo(x - 1) + fibo(x - 2)
```

- Nested Recursion

$$h(n) = \begin{cases} 0 & if\ n = 0 \\ n & if\ n > 4 \\ h(2 + h(2n)) & if\ n \le 4 \end{cases}$$

```python
#Nested recursion
def func(n: int) -> int:
    if n == 0: return 0
    elif n > 4: return n
    else:
        return func(2 + func(2 * n))
```

Example 01

- Write a Python program to calculate the sum of a list of numbers with recursion
- **Input: li = [1, 2, 3, 4, 5]**
  - **Output: 15**

Example 01

- Calculating the sum of all elements in a list of integers:

$$a_0 + a_1 + \cdots + a_{n-1}$$

- **Recursion definition:**

$$S_n = a_0 + a_1 + \cdots + a_{n-1}$$
$$S_n = S_{n-1} + a_{n-1}$$

  - **Base case:** $S_1 = a_0$

- Calculating the sum of all elements in an array:

```python
def calcSumOfList(a: list, n: int) -> int:
    if n == 0:
        return 0
    else:
        return a[n - 1] + calcSumOfList(a, n-1)
```

Recursion Example

- Calculating the sum of all elements in an array:

```python
def calcSumOfList(a: list) -> int:
    if len(a) == 0:
        return 0
    else:
        return a[-1] + calcSumOfList(a[:-1])
```

Example 02

- Verifying if the elements in an array are in ascending order

$$a_0 \leq a_1 \leq \cdots \leq a_{n-1}?$$

- The above array is in ascending order if it satisfies two conditions
  - The first $n - 1$ elements are in ascending order, and
  - $a_{n-2} \leq a_{n-1}$
- If the array contains only one element (a_0), it must be in ascending order

The most common error

- Stack Overflow: means that you've tried to make too many function calls recursively and the memory in stack is full
- If you get this error, one clue would be to look to see if you have infinite recursion
  - This situation will cause you to exceed the size of your stack -- no matter how large your stack is

Recursion and Iteration

- While recursion is very powerful
  - It should not be used if iteration can be used to solve the problem in a maintainable way (i.e., if it isn't too difficult to solve using iteration)
  - So, think about the problem. Can loops do the trick instead of recursion?

Recursion and Iteration

- Why select iteration versus recursion
  - Every time we call a function a stack frame is pushed onto the program stack and a jump is made to the corresponding function
  - This is done in addition to evaluating a control structure (such as the conditional expression for an if statement to determine when to stop the recursive calls
  - With iteration all we need is to check the control structure (such as the conditional expression for the while, do-while, or for) $\rightarrow$ effiency

Recursion and Iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a **looping construct**
  - A recursive algorithm uses a **branching structure**
- Recursive solutions are often less efficient, in terms of both **time** and **space**, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in **shorter**, more easily understood source code

Exercise 03

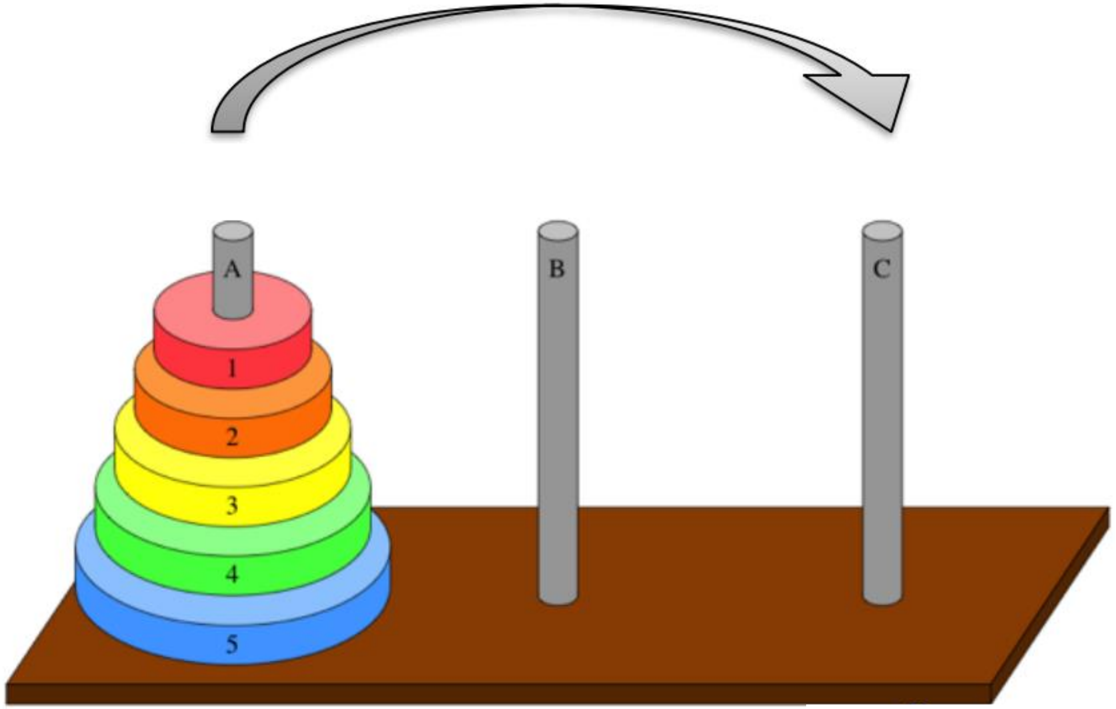
- Verifying if a string is a palindrome
- Formally, a palindrome can be defined as follows:
  - If a string is a palindrome, it must begin and end with the same letter. Further, when the first and last letters are removed, the resulting string must also be a palindrome
  - A string of length 1 is a palindrome
  - The empty string is a palindrome

Exercise 04

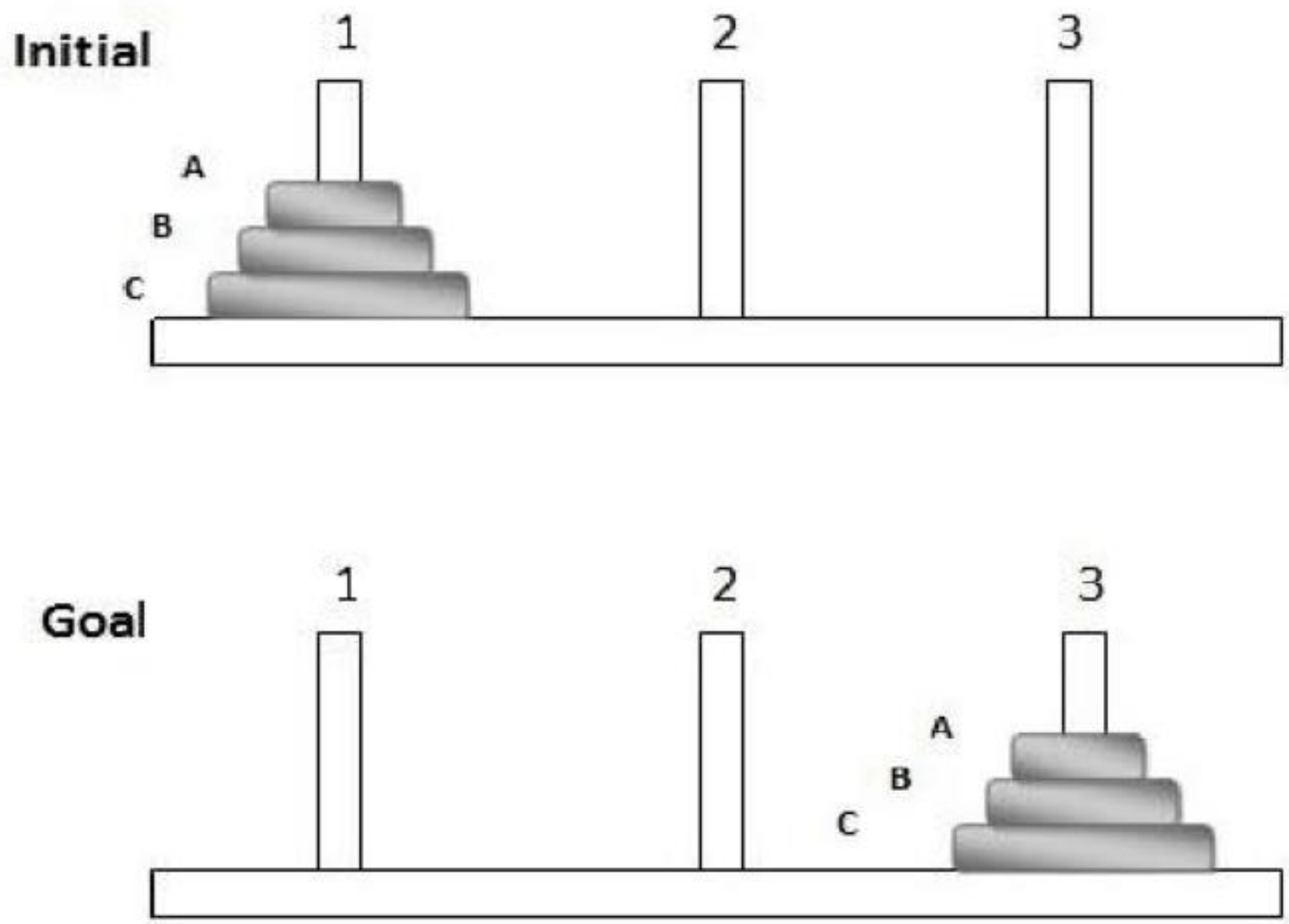
- Implement binary search with recursion

The Towers of Hanoi Puzzle

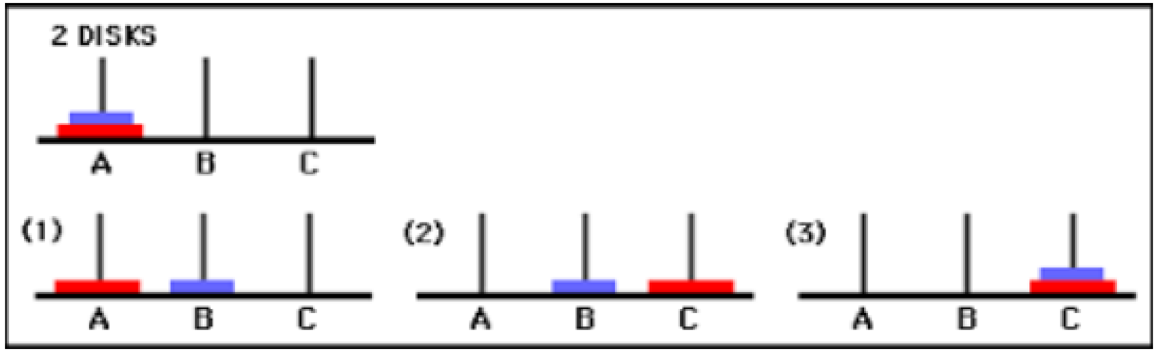
- Given a set of three pegs A, B, C, and n disks, with each disk a different size (disk 1 is the smallest, disk n is the largest)
- Initially, n disks are on peg A, in order of decreasing size from bottom to top.
- The goal is to move all n disks from peg A to peg C
- 2 rules:
  - You can move 1 disk at a time.
  - Smaller disk must be above larger disks



The Towers of Hanoi Puzzle

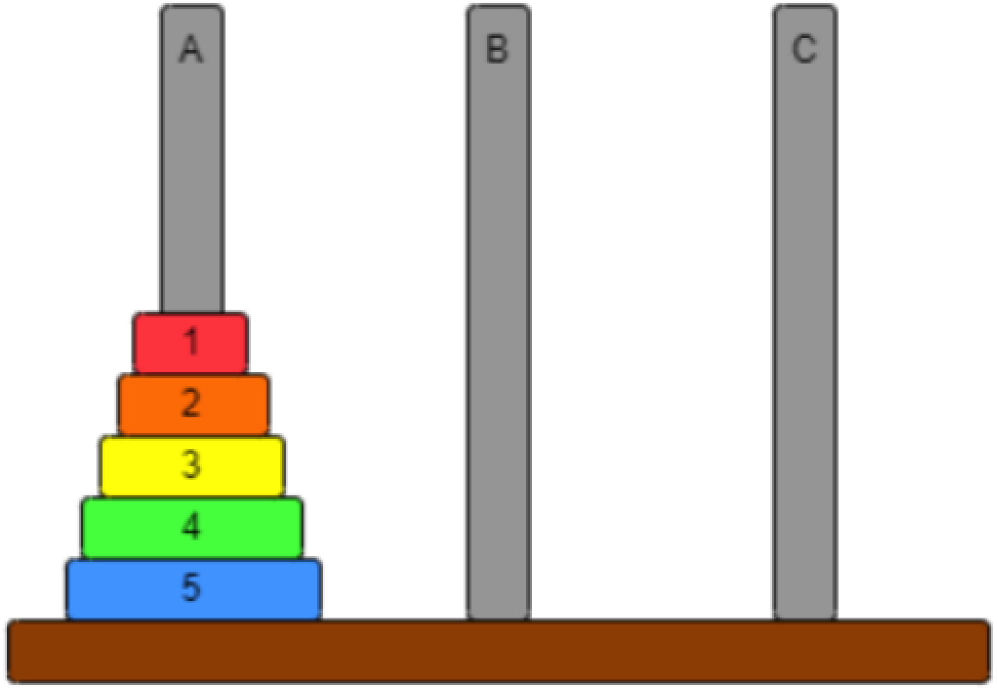


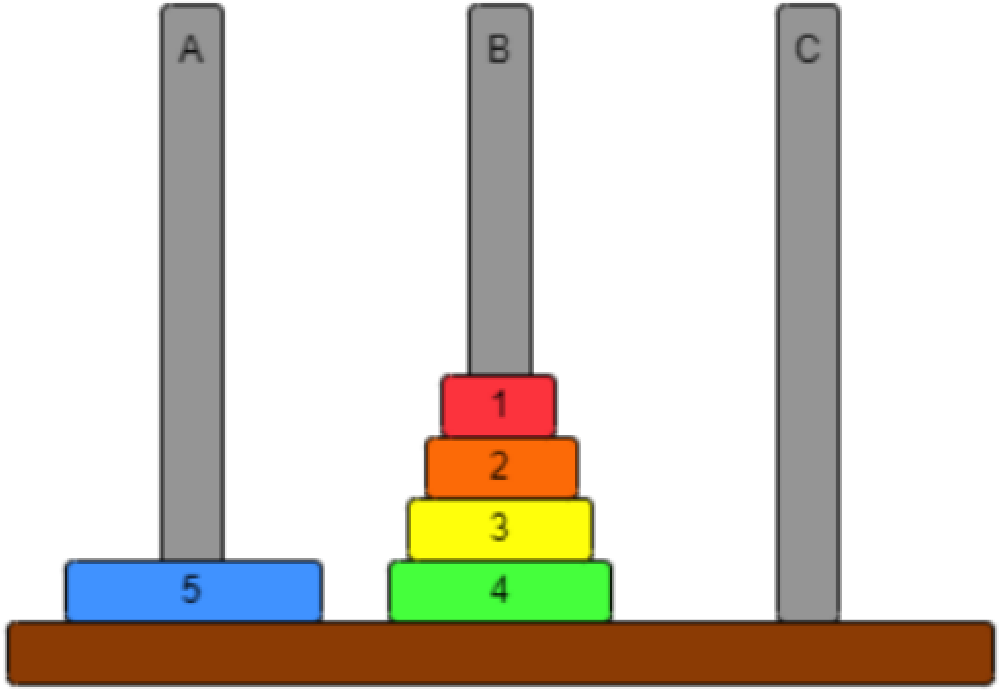
- How to solve this problem recursively?
  - The easiest case: (base case)
    - $n = 1$: just move disk 1 from A to C
  - When $n = 2$: 3 steps (using B as the spare peg)
    - Move disk 1 from A to B
    - Move disk 2 from A to C
    - Move disk 1 from B to C



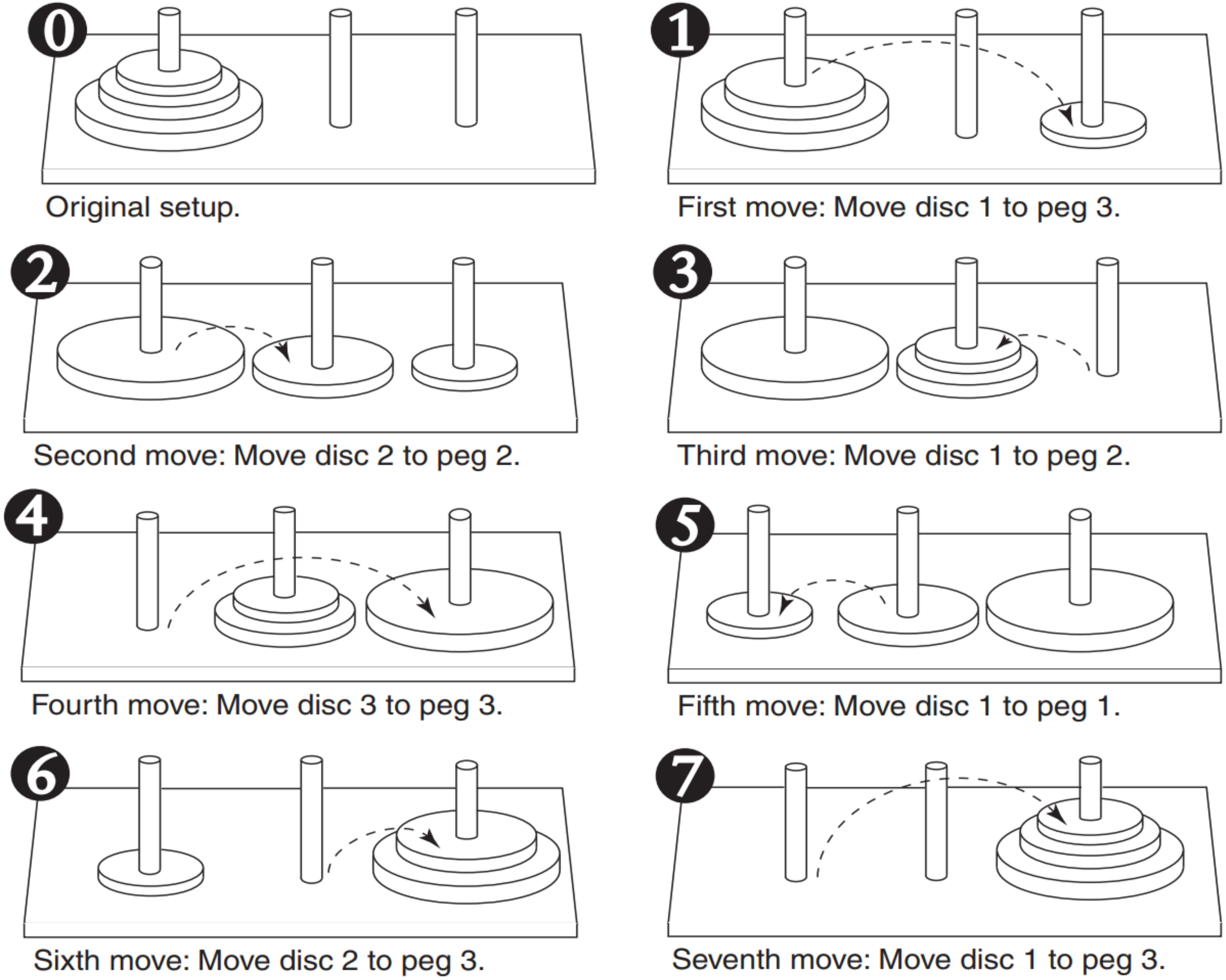
  - When $n = k$:
    - Move disk $1, 2, \dots, k-1$ from A to B
    - Move disk k from A to C
    - Move disk $1, 2, \dots, k-1$ from B to C

- How to solve this problem recursively?
  - n = 5:
    - Step 1: Move disk $1, 2, \ldots, 4$ from A to B

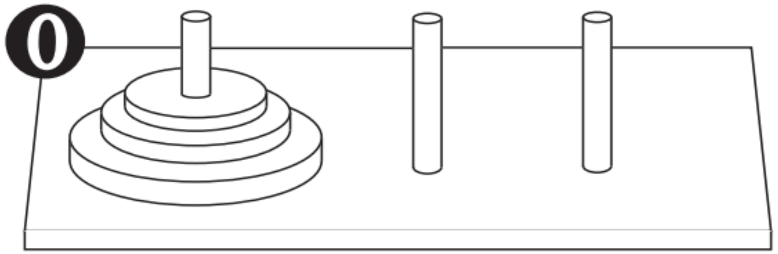




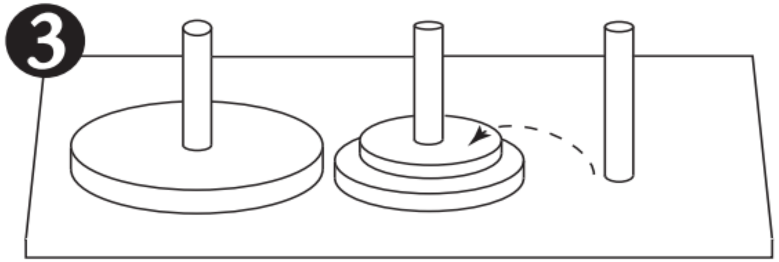
Hanoi Tower Puzzle



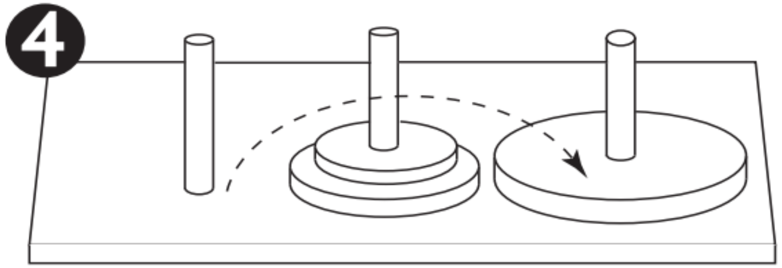
Hanoi Tower Puzzle



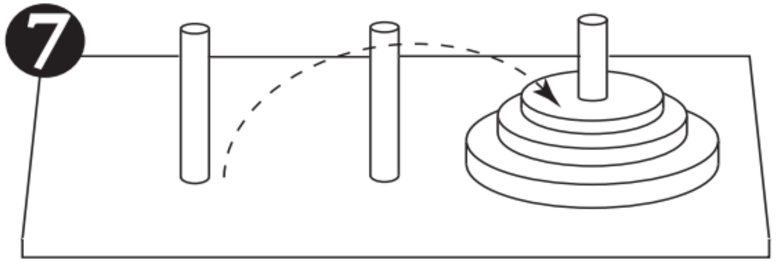
Original setup.



Third move: Move disc 1 to peg 2.



Fourth move: Move disc 3 to peg 3.



Seventh move: Move disc 1 to peg 3.

Move n discs from peg 1 to peg 3 using peg 2 as an auxiliary peg

If n > 0 **then**

Move n – 1 discs from peg 1 to peg 2, using peg 3 as an auxiliary peg

Move the remaining disc from the peg 1 to peg 3

Move n – 1 discs from peg 2 to peg 3, using peg 1 as an auxiliary peg

End If

```python
def hanoi(n, source, target, auxiliary):
    """Recursive function to solve Tower of Hanoi problem...."""
    if n > 0:
        # Move n-1 disks from source to auxiliary peg
        hanoi(n-1, source, auxiliary, target)

        # Move the n-th disk from source to target peg
        print(f"Move disk {n} from {source} to {target}")

        # Move the n-1 disks from auxiliary to target peg
        hanoi(n-1, auxiliary, target, source)
```

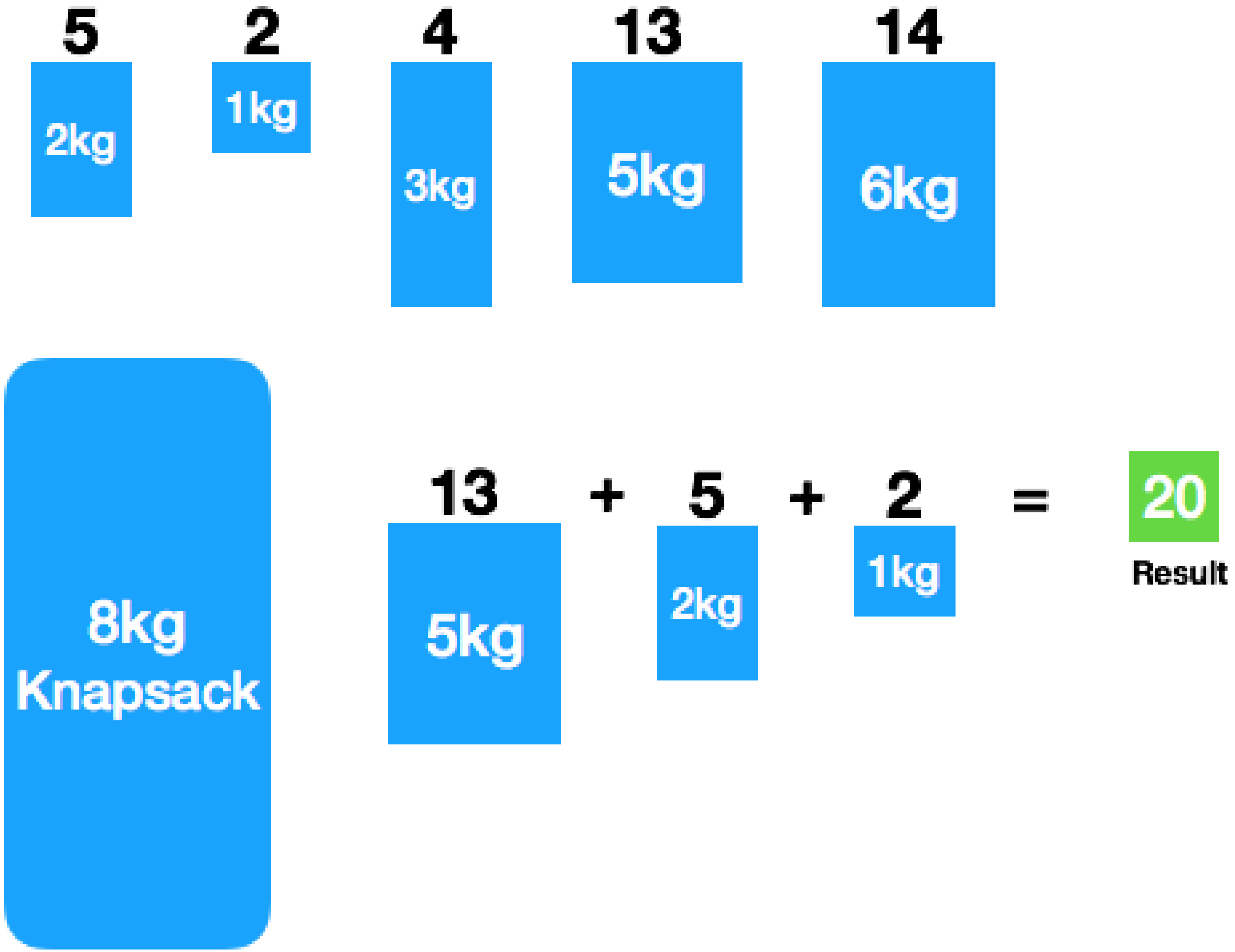
The Knapsack problem

- Problem statement:
  - A thief is robbing a museum and he only has a single knapsack to carry all the items he steals.
  - The knapsack has a capacity for the amount of weight it can hold. Each item in the museum has a weight and a value associated with it

The Knapsack problem – variantion

- 0/1 Knapsack problem
  - Each item is chosen at most once.
  - Decision variable for each item is a binary value (0 or 1)
- Multiple-choice Knapsack problem
  - Each item can be put to the knapsack multiple times.
  - Decision variable for each item is an integer value.
- Bounded Knapsack problem
  - Same with multiple-choice but each item has the max number of times it can be chosen.
- Knapsack problem with fractional items
- Knapsack problem with multiple constraint
- …

0/1 Knapsack problem



0/1 Knapsack problem

- Knapsack's capacity: 10kg
- 5 items can be chosen:
  - Item 1: $6 (2 kg)
  - Item 2: $10 (2 kg)
  - Item 3: $12 (3 kg)
  - Item 4: $16 (4kg)
  - Item 5: $20 (5kg)

0/1 Knapsack problem

- We can also use a ***bottom-up*** approach and memorize the solutions to subproblems to a table → *Dynamic Programming*
  - Row: items
  - Column: remaining weight capacity of the knapsack
  - We fill the table using the following recurrence relation:

$$f(W, i) = \max\big(f(i-1, W-w_i) + x_i, f(i-1, W)\big)$$

	0kg	1kg	2kg	3kg	4kg	5kg	6kg	7kg	8kg	9kg	10kg
1											
2											
3											
4											
5											

0/1 Knapsack problem

- Use only item 1:

→ $f(1,1) = 0, f(1,2) = 6, f(1,3) = 6, \dots, f(1,10) = 6$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10										
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1 & 2:

→ $f(2,2) = \max\left(f(1,0) + 10, f(1,2)\right) = 10, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10								
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

- Use item 1 & 2:

→ $f(2,3) = \max(f(1,1) + 10, f(1,3)) = 10, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	\$6	0	6	6	6	6	6	6	6	6	6
2	2kg	\$10	0	10	10							
3	3kg	\$12										
4	4kg	\$16										
5	5kg	\$20										

- Use item 1 & 2:

→ $f(2,4) = \max\left(f(1,2) + 10, f(1,4)\right) = 16, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16						
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

- Use item 1 & 2:

→ $f(2, i) = \max\big(f(1, W-2) + 10, f(1, W)\big)$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

- Use item 1, 2, 3:

→ $f(3,3) = \max\left(f(2,0) + 12, f(2,3)\right) = 12$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12							
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3:

→ $f(3,4) = \max(f(2,1) + 12, f(2,4)) = 16$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	\$6	0	6	6	6	6	6	6	6	6	6
2	2kg	\$10	0	10	10	16	16	16	16	16	16	16
3	3kg	\$12	0	10	12	16						
4	4kg	\$16										
5	5kg	\$20										

0/1 Knapsack problem

- Use item 1, 2, 3:

→ $f(3,5) = \max(f(2,2) + 12, f(2,5)) = 22$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22					
4	4kg	$16										
5	5kg	$20										

- Use item 1, 2, 3:

→ $f(3, i) = \max\left(f(2, W-3) + 12, f(2, W)\right)$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16										
5	5kg	$20										

- Use item 1, 2, 3, 4:

→ $f(4, i) = \max\big(f(3, W-4) + 16, f(3, W)\big)$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3, 4, 5:

→ $f(5,10) = \max\left(f(4,5) + 20, f(4,10)\right) = 42$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20	0	10	12	16	22	26	30	32	38	42

- Solution:
  - item 5 + item 3 + item 2 → $42 – 10kg

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20	0	10	12	16	22	26	30	32	38	42

- Optimal structure: to find $f(n, W)$:
  1. **Case 1:** including the *n^th* item

     → find $f(n-1, W-w_n)+x_n$ with x_n is the value of item *n^th*
  2. **Case 2:** not include the *n^th* item

     → find $f(n-1, W)$
- Hence, optimal f is calculated by:

$$\boldsymbol{f(n, W)=\max\left(f(n-1, W-w_n)+x_n, f(n-1, W)\right)}$$

→ *This can be solved using **recursion** which is a **top-down** strategy.*

0/1 Knapsack problem

```python
def knapsack01(capacity, weights, values, n):
    """Recursive function to solve the 0/1 Knapsack problem...."""
    # Base case: If there are no items left or the capacity is 0
    if n == 0 or capacity == 0:
        return 0

    # If the weight of the nth item is more than the capacity of the knapsack,
    # then this item cannot be included in the optimal solution
    if weights[n - 1] > capacity:
        return knapsack01(capacity, weights, values, n - 1)
    else:
        # Return the maximum of two cases:
        # 1. The value of the nth item included in the optimal solution
        # 2. The value of the nth item not included in the optimal solution
        return max(values[n - 1] + knapsack01(capacity - weights[n - 1], weights, values, n - 1),
                   knapsack01(capacity, weights, values, n - 1))
```

Exercise 05

- Find the sum of all digits in the positive number n
- Input: n = 1243
- Output: 10

Exercise 06

- Find the biggest digit in the positive number n
- Input: n = 1243
- Output: 4

Exercise 06

- Check whether a positive number is a prime number of not without using iteration

THANK YOU

for YOUR ATTENTION