SAMSUNG

Samsung Innovation Campus

Coding and Programming

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 4.

Algorithm 1 - Data Structures

Coding and Programming

Chapter Description

Chapter objectives

- ✓ Learners will be able to understand the concept of data structures, define and apply abstract data types including stack, queue, and hash table. Also, learners will be able to understand linear search and binary search to solve search problems, while be able to compare data search methods using hash table and algorithm performance.

Chapter contents

Unit 22.

Stack

Unit learning objective (1/3)

Learning objectives

- ✓ Understand the concept of stack and be able to define stack as abstract data type.
- ✓ Be able to implement stack data structures into classes in Python language.
- ✓ Be able to make applications of stack data structures to solve balanced parentheses problems.

Unit learning objective (2/3)

Learning overview

- ✓ Learn the methods to implement stack data structures using Python lists.
- ✓ Learn the methods to define stack data structures as Python's class.
- ✓ Learn the methods to solve balanced parentheses problems by using stack data structures.

Concepts You Will Need to Know From Previous Units

- ✓ Declaring a list, add and delete items.
- ✓ Define class constructor and define variables and methods.
- ✓ Creating class instance to approach to the variables and call methods.

Keywords

- Abstract Data Type
- Stack
- LIFO
- push
- pop

Mission

1. Real world problem

1.1. Parentheses in source codes

```c
#include <stdio.h>

char *s[] = {"Hello", "World"};

int main() {

    printf("%s, %s!\n", s[0], s[1]);

}
```

- There are different kinds of parentheses in computer program source codes, and they need to be paired.
- The figure on the left shows the source code of "Hello, Word!" program written in C language. The figure below provides a sequential listing of the parentheses included in this source code (square brackets, braces, and round brackets).

[]{}(){([][])}

- If the parentheses above are invalid due to inappropriate pairing, the C language compiler will print an error. Parentheses in Python language should be appropriately paired.
- How should we check if the parentheses are appropriately paired in the given source code?

2. Mission

2.1. Balanced parentheses checker

Make a program to check if the balanced parentheses are valid or invalid when strings consist of square backet, brace, and round bracket are provided as below.

Valid	Invalid
[]{}()	(((
((()))	(((]]]
({[]})	)(][}{

3. Solution

3.1. How the balanced parentheses checker works

The balanced parentheses inputs are then classified into either valid or invalid.

```
1 str = input("Input a string of parentheses: ")
2 print("Valid" if check(str) else "Invalid")
```

```
Input a string of parentheses: [{()}]
Valid
```

```
1 str = input("Input a string of parentheses: ")
2 print("Valid" if check(str) else "Invalid")
```

```
Input a string of parentheses: [{(]})
Invalid
```


3. Solution

3.2. Balanced parentheses checker final code

```
1  def check(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = []
5      for char in expr:
6          if char in opening:
7              stack.append(char)
8          elif char in closing:
9              if len(stack) == 0:
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return len(stack) == 0
```

Key concept

Key concept

UNIT 22

1. Algorithms and Data Structures

1.1. Pre-defined data types and user-defined data types

- All of the programming languages provide defined data types. In Python, there are many different kinds of built-in data types including integer type, real type, logic type, string, list, dictionary, etc.
- Also, all of the programming languages provide a way to define a new data type.
- In Python, users can define a new data type as a class.

FOCUS Abstract data type refers to the data type defined by the user.

1. Algorithms and Data Structures

1.2. Algorithms and data structures

- Algorithm refers to a step-by-step procedure to solve a given problem in an effective manner. Data structure is an abstract concept that forms and organizes data for effective insertion and extraction.
- Thus, an efficient algorithm must be used to define a data structure, and an appropriate data structure should be selected to design an efficient algorithm.

FOCUS Computer program is a set of data structure and algorithm.

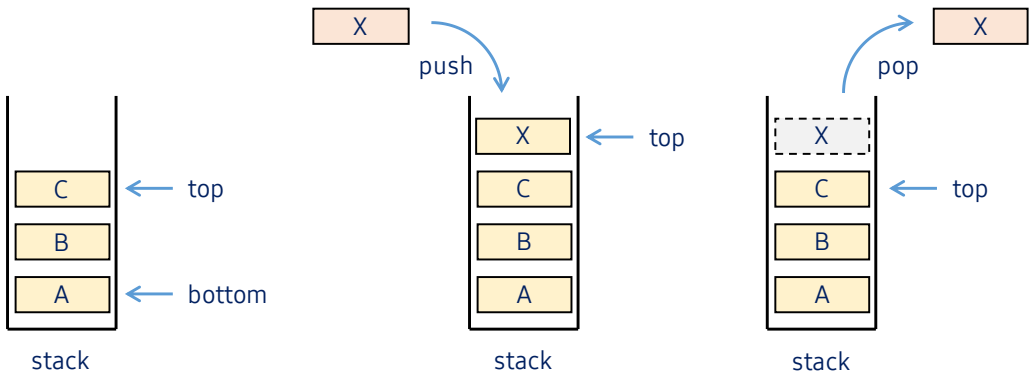
2. What is a Stack?

2.1. Definition of a stack

- Stack is an abstract data type in which an item can be added or deleted only at one side of the top. This data type works by the LIFO (Last-In-First-Out) method where you can remove the item first that was previously entered very last.

Focus Stack must provide the following two interfaces.

- push: Adds a new item on the top.
- pop: Removes an item from the top.



2. What is a Stack?

2.2. Examples of a stack

- Stack is commonly found in real life situations.
- For instance, dishes are stacked while dishwashing and are then taken by one by one for washing.
- Other examples of a stack in real life include stacking books, coins, and shuttlecock container and Pringles.
- What would be other examples of a stack?

2. What is a Stack?

2.2. Examples of a stack

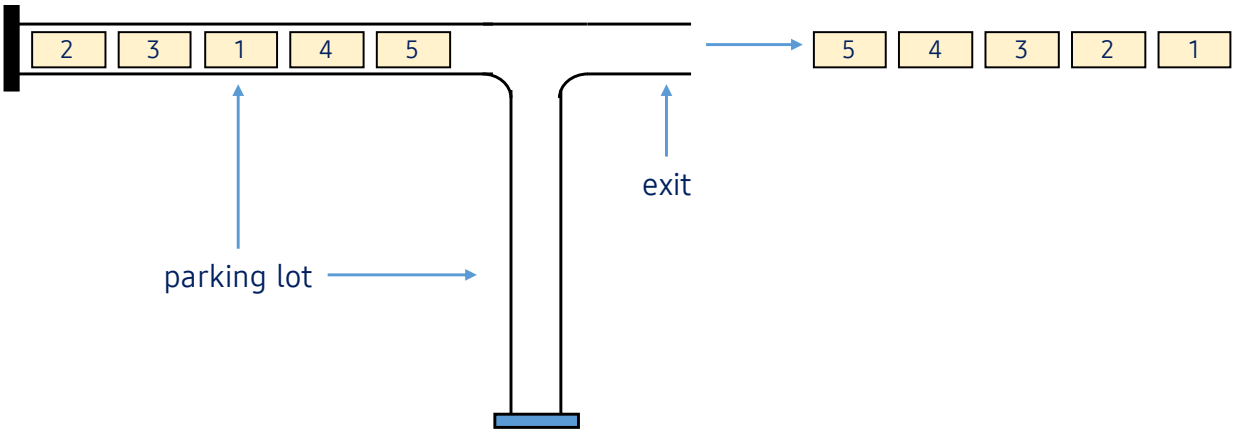
- The web browser accumulates the recently visited website addresses in a stack. When visiting a new page, it will be accumulated on the top of the stack.
- If the user clicks 'back,' the address on the top of the stack is selected which will show previously visited website.

- There's an undo function in the text editor and MS Word.
- This function also accumulates editing operations done by the user into a stack, and the 'undo' will cancel the operations in order from the top of the stack.

2. What is a Stack?

2.3. Applications of a stack

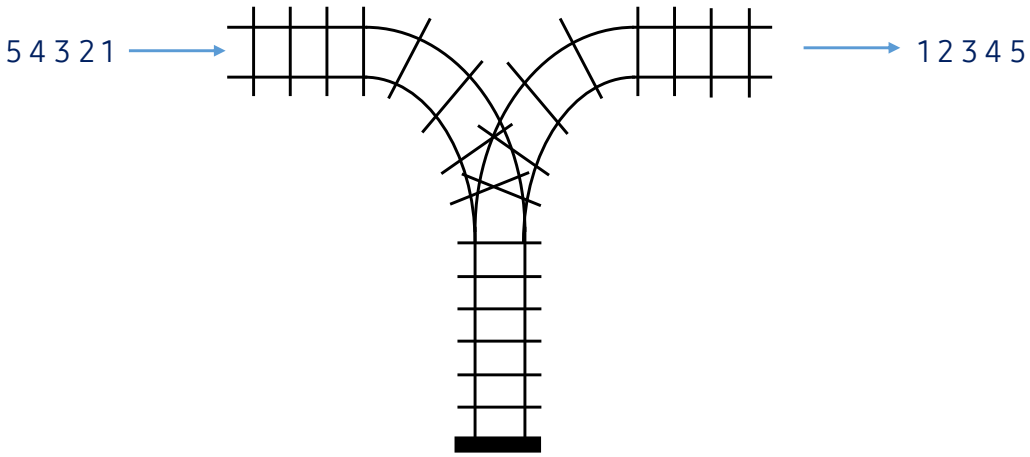
- Take a parking lot in the figure below for an example.
- On the left, there are 5 cars parked along the dead end, and the lane on the bottom is also a dead-end. The drivers of the cars arrived in the order of 2, 3, 1, 4, 5 with random time interval. The cars need to get out to the right in the order of 1, 2, 3, 4, 5.
- If you're a parking lot manager, how would you solve this problem?



2. What is a Stack?

2.3. Applications of a stack

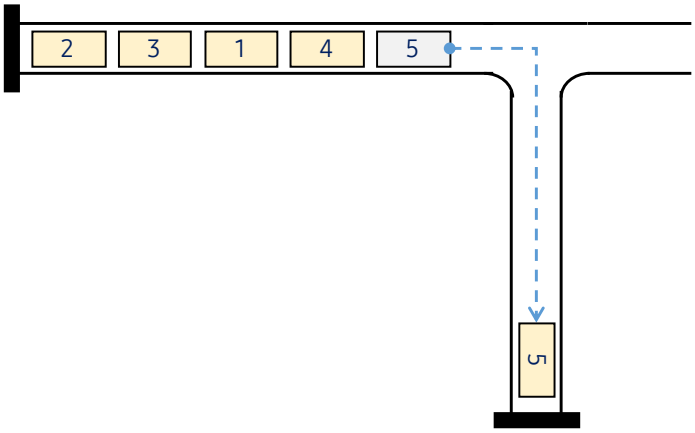
- Before trying to solve the parking lot problem, think of a railroad track that looks like below.
- If a train that has plate numbers from 1 to 5 enters the track from the left and gets out to the right, then the plate number will be in a reversed order.



2. What is a Stack?

2.3. Applications of a stack

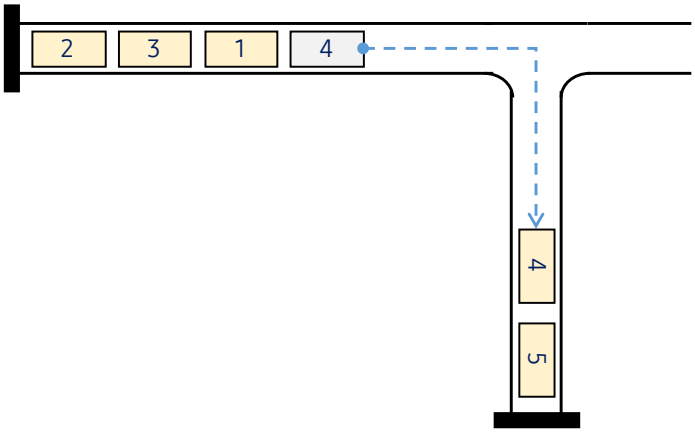
- Solve the parking lot problem as follows.
- First, temporarily park the car no.5 in the bottom alley.



2. What is a Stack?

2.3. Applications of a stack

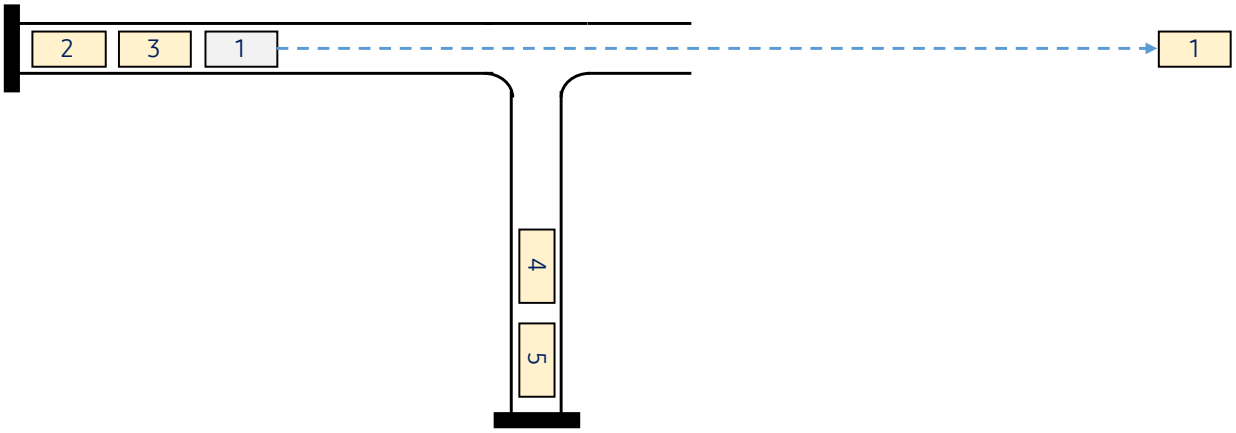
- Next, also park the car no.4 in the bottom alley.



2. What is a Stack?

2.3. Applications of a stack

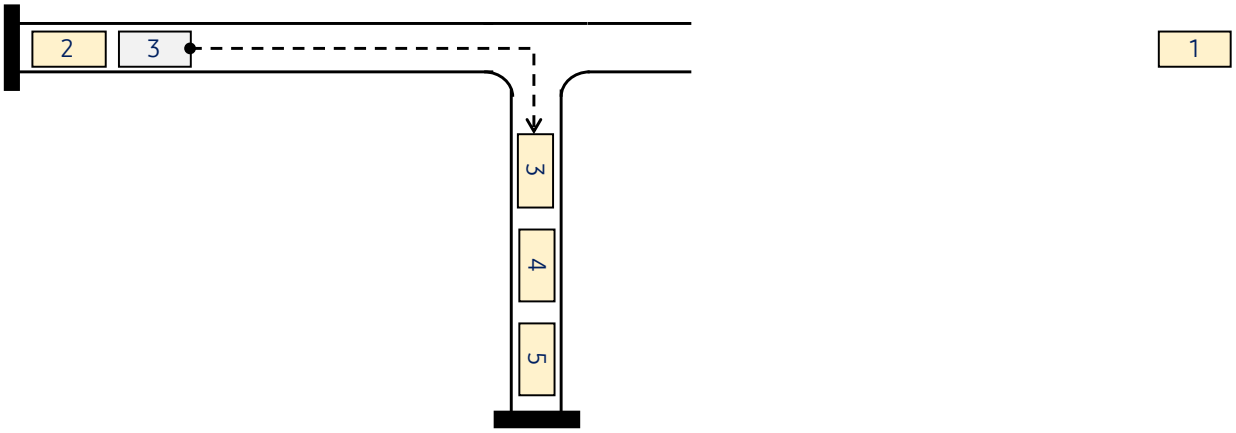
Now, the car no.1 can get out of the parking lot first.



2. What is a Stack?

2.3. Applications of a stack

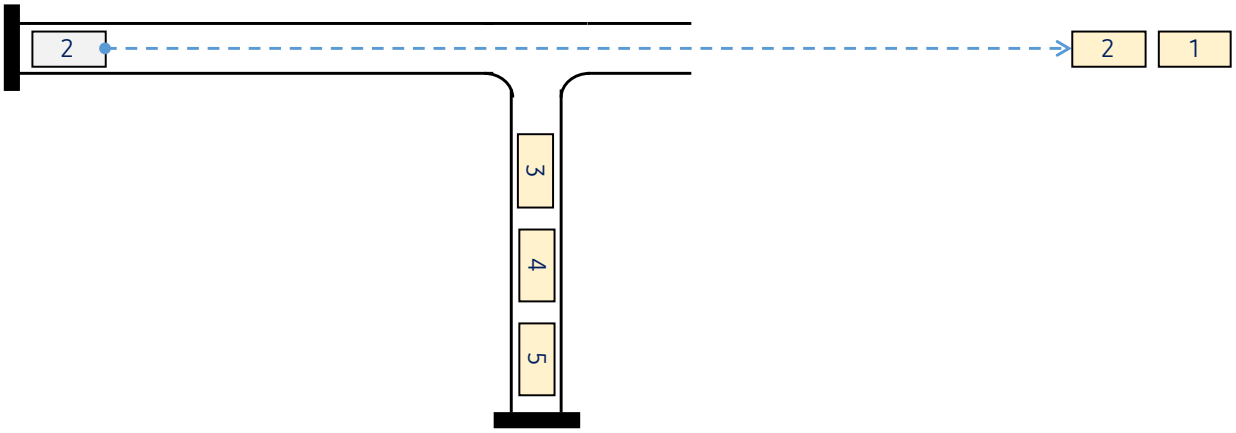
Again, temporarily park the car no.3 in the bottom alley.



2. What is a Stack?

2.3. Applications of a stack

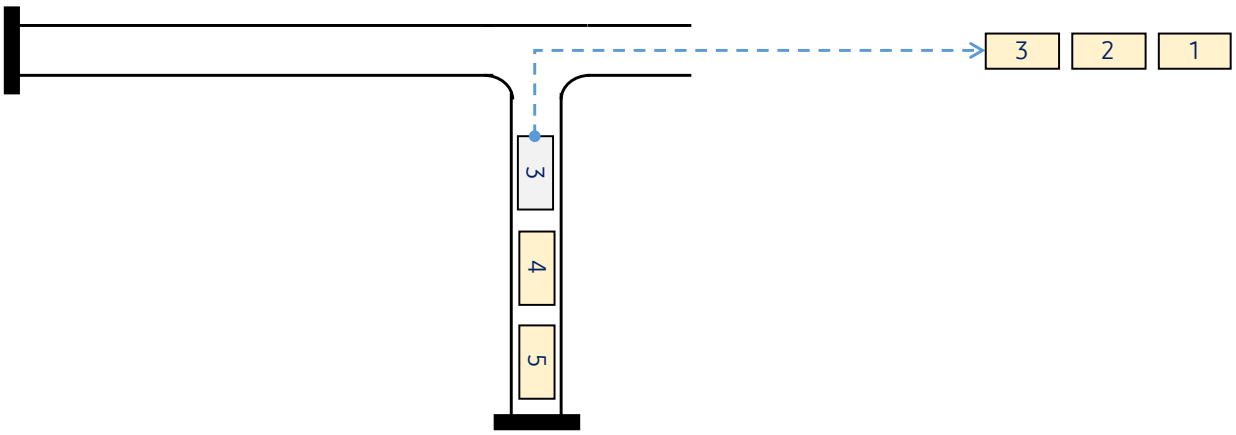
- Then, the car no.2 can get out of the parking lot followed by car no.1.



2. What is a Stack?

2.3. Applications of a stack

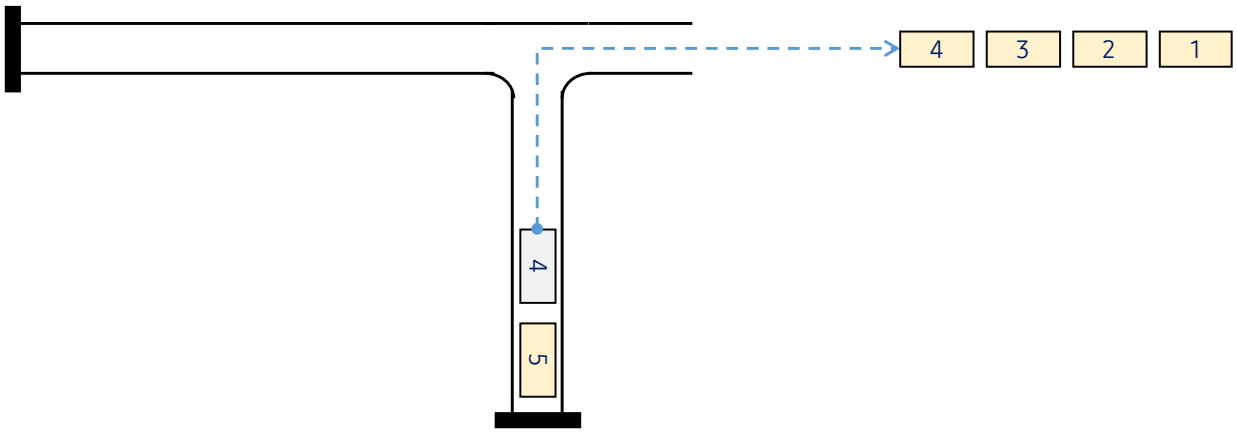
- Because the bottom alley is parked with right order of cars that need to get out, the car no.3 can get out followed by car no.2.
- It is important to understand that the car no.3 is the last car parked in the bottom alley.



2. What is a Stack?

2.3. Applications of a stack

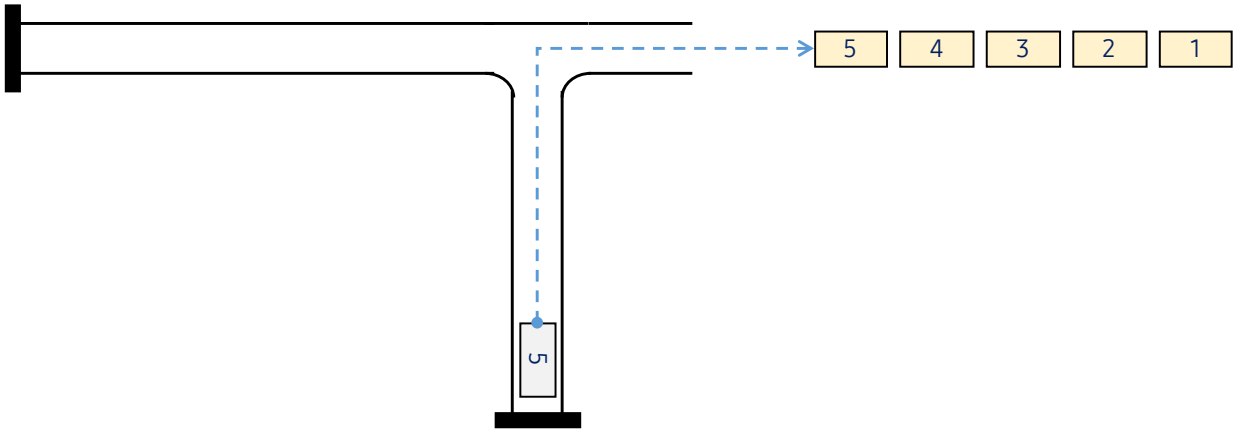
- This time, the last car parked in the bottom alley is car no.4.
- The car no.4 can get out of the parking lot followed by car no.3.



2. What is a Stack?

2.3. Applications of a stack

- Now, if the car no.5 gets out of the parking lot, then the problem is solved.
- The car no.5 that gets out of the parking lot very last is the car that was first parked in the bottom alley.



3. Implement a Stack

3.1. Using a list data type

In Python, a stack can be defined as follows by using the list data type.

```
1 stack = []
2
3 def push(stack, item):
4     stack.append(item)
5
6 def pop(stack):
7     return stack.pop()
```

Line 1

- Define a stack as a list type variable.
- Addition or deletion of an element is possible at any location in the phyton list type, so an additional interface must be provided to perform 'push' and 'pop' for stack implementation.

3. Implement a Stack

3.1. Using a list data type

For convenience, assume that the last element of the stack is 'top.' Then, the 'push' will add the element at the top of the stack.

```
1 stack = []
2
3 def push(stack, item):
4     stack.append(item)
5
6 def pop(stack):
7     return stack.pop()
```

Line 3-4

- The 'push' function receives input of the list type stack and item to be added.
- Since the 'push' will add the item to the top of the stack, use the append() function of the stack list.

3. Implement a Stack

3.1. Using a list data type

The 'pop' will delete the last element of the stack and return the deleted element.

```
1  stack = []
2
3  def push(stack, item):
4      stack.append(item)
5
6  def pop(stack):
7      return stack.pop()
```

Line 6-7

- The list type stack is input of the 'pop' function.
- Since 'pop' deletes and returns the top element of the stack, use pop() function of the stack list.

3. Implement a Stack

3.1. Using a list data type

The following code tests the stack data structure that was defined previously.

```
1  stack = []
2  print(stack)
3  push(stack, "A")
4  print(stack)
5  push(stack, "B")
6  print(stack)
7  pop(stack)
8  print(stack)
9  push(stack, "C")
10 print(stack)
11 pop(stack)
12 print(stack)
13 pop(stack)
14 print(stack)
```

```
[]
['A']
['A', 'B']
['A']
['A', 'C']
['A']
[]
```

3. Implement a Stack

3.1. Using a list data type

When the 'pop' is asked in an empty stack, IndexError occurs because the stack has nothing to delete.

IndexError

```
1 stack = []
2 pop(stack)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-29-f40cd90df1a6> in <module>
      1 stack = []
----> 2 pop(stack)

<ipython-input-21-3270d1cc050e> in pop(stack)
      5
      6 def pop(stack):
----> 7     return stack.pop()

IndexError: pop from empty list
```


3. Implement a Stack

3.1. Using a list data type

To solve the IndexError problem in an empty stack, first create the function 'is_empty()' to check if the stack is empty.

```
1 def is_empty(stack):
2     return True if len(stack) == 0 else False
3
4 stack = []
5 print(is_empty(stack))
```

```
True
```

Line 1-2

- The is_empty() function returns False if the length of the stack as a parameter is 0, and True if not.
- When initializing a stack, the stack length is 0. So, is_empty() returns True.

3. Implement a Stack

3.1. Using a list data type

To solve the IndexError problem in an empty stack, first create the function 'is_empty()' to check if the stack is empty.

```
1  def pop(stack):
2      return None if is_empty(stack) else stack.pop()
3
4  stack = []
5  print(pop(stack))
```

```
None
```

Line 1-2

- The pop() function deletes and returns the None object if the stack is empty, and if the stack is not empty, it deletes and returns the top element of the stack list.
- The initialized stack is empty, so pop() returns None object and no IndexError occurs.

3. Implement a Stack

3.2. Define a stack as a class type

In Python, abstract data type can be defined as a class.

The abstract stack data type that was implemented as a list can be defined as a class as shown below.

```
1  class Stack:
2
3      def __init__(self):
4          self.stack = []
5
6      def is_empty(self):
7          return True if len(self.stack) == 0 else False
8
9      def push(self, item):
10         self.stack.append(item)
11
12     def pop(self):
13         return None if self.is_empty() else self.stack.pop()
```

Line 1-4

- Define the stack data structure as a stack class.
- __init__(self) constructor initializes the stack member field into an empty list.

3. Implement a Stack

3.2. Define a stack as a class type

The 'push' can be implemented as the method of the stack class as shown below.

```
1  class Stack:
2
3      def __init__(self):
4          self.stack = []
5
6      def is_empty(self):
7          return True if len(self.stack) == 0 else False
8
9      def push(self, item):
10         self.stack.append(item)
11
12     def pop(self):
13         return None if self.is_empty() else self.stack.pop()
```

Line 9-10

- The push() method receives the 'self' and 'item' to be added to the stack as parameters.
- The 'push' adds the item to the top of the self.stack list which is the stack class member field.

3. Implement a Stack

3.2. Define a stack as a class type

The 'pop' can be implemented as the method of the stack class as shown below.

```
1  class Stack:
2
3      def __init__(self):
4          self.stack = []
5
6      def is_empty(self):
7          return True if len(self.stack) == 0 else False
8
9      def push(self, item):
10         self.stack.append(item)
11
12     def pop(self):
13         return None if self.is_empty() else self.stack.pop()
```

Line 12-13

- The pop() method receives 'self' as a parameter.
- The 'pop' returns None if the self.is_empty() is True, and if not, it deletes and returns the top element of the self.stack.

3. Implement a Stack

3.2. Define a stack as a class type

The is_empty() method that checks if a stack is empty should also be defined as a stack class method.

```
1  class Stack:
2
3      def __init__(self):
4          self.stack = []
5
6      def is_empty(self):
7          return True if len(self.stack) == 0 else False
8
9      def push(self, item):
10         self.stack.append(item)
11
12     def pop(self):
13         return None if self.is_empty() else self.stack.pop()
```

Line 6-7

- The is_empty() method returns True if the self.stack length is 0, and if not, it returns False.

3. Implement a Stack

3.2. Define a stack as a class type

The following code tests the stack data structure that was defined as a class previously.

```
1  stack = Stack()
2  stack.push("A")
3  stack.push("B")
4  print(stack.pop())
5  stack.push("C")
6  print(stack.pop())
7  print(stack.pop())
8  print(stack.pop())
```

```
B
C
A
None
```

Line 1-8

- The stack variable saves the object instance of a stack class that was created by calling Stack() constructor.
- The push() and pop() methods of the stack object can be called by using a reference operator '.'
- Print the returned value of the stack.pop() function into the print() function to identify the order of removal from the stack.

One More Step

- The pop() function used the pop() method of the list to delete the top element of the stack and then returned it. How would you implement the peek() function that lets you know which element is on the top of the stack?
- For the peek() method, simply return the top element without deleting the element.

```
1  def peek(stack):
2      return None if is_empty(stack) else stack[-1]
```

- It is also possible to define the peek() method as a member method of the stack class that was previously defined as follows.

```
15     def peek(self):
16         return None if is_empty(self.stack) else self.stack[-1]
```

TIP

- When designating the index value as -1 in the Python list, it is possible to approach to the top element of the list.
- stack[-1] == stack[len(stack) – 1]

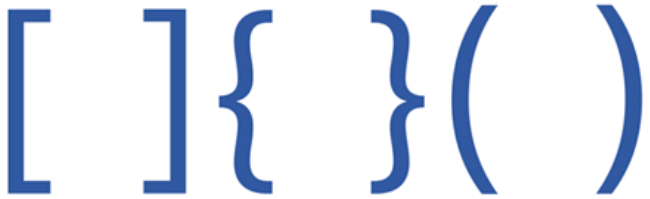
Let's code

1. Balanced Parentheses Problem

1.1. Problem definition

- Write an algorithm that checks if the parentheses are valid when a string consists of parenthesis is given.
- Print Valid if the string parentheses are valid, and if not, print Invalid.
- Assume that there are three different kinds of parentheses that can be included in the input string.
  - Square bracket: [], Brace: {}, Round bracket: ()
- Parentheses are valid only if satisfying the following conditions.

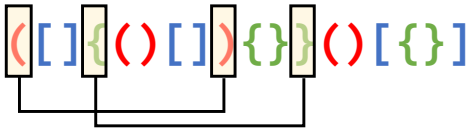
1. The right parentheses type should be paired up.
2. The parentheses should have a right order.
3. Consider the string without parentheses as valid parentheses.



1. Balanced Parentheses Problem

1.2. Input & output examples

- The following shows examples of the string containing valid and invalid parentheses.

Valid	Invalid
[]{}()	(
((()))	(]
({[]})	)(
[]{}()({[]})((()))	([()})
([]{()[]{}}())[{}]	

1. Balanced Parentheses Problem

1.3. Codes for the solution

- The check_parentheses() function receives the string consists of user-entered parentheses as input and prints valid if the parentheses are appropriately paired up and prints invalid if not.
- Use this function to print valid or invalid as shown below.

```
1 str = input("Input a string of parentheses: ")
2 print("Valid" if check_parentheses(str) else "Invalid")
```

```
Input a string of parentheses: ([]{()[]{}}())[{}]
Valid
```

```
1 str = input("Input a string of parentheses: ")
2 print("Valid" if check_parentheses(str) else "Invalid")
```

```
Input a string of parentheses: ([]{()[]){}}()[{}]
Invalid
```


1. Balanced Parentheses Problem

1.3. Codes for the solution

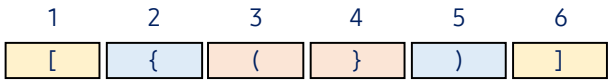
- The following shows final results of the check_parentheses() function.

```
1  def check_parentheses(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = Stack()
5      for char in expr:
6          if char in opening:
7              stack.push(char)
8          elif char in closing:
9              if stack.is_empty():
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return stack.is_empty()
```

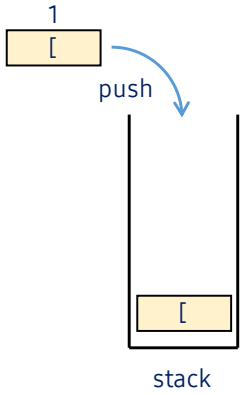
1. Balanced Parentheses Problem

1.4. Solving the problem using a stack

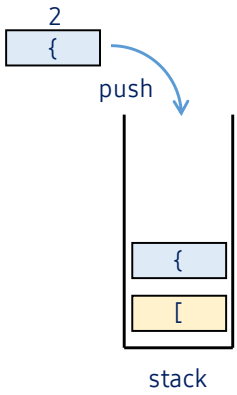
Use a stack data structure to easily check if the parentheses are appropriately paired up. For instance, pair up the string provided below.



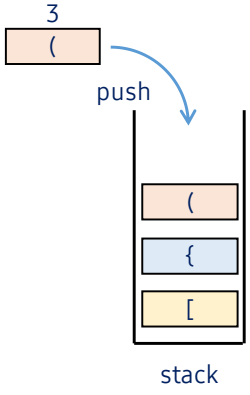
1. Push the open parenthesis into the stack.



2. Push the open parenthesis into the stack.

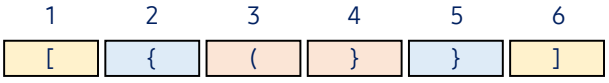


3. Push the open parenthesis into the stack.

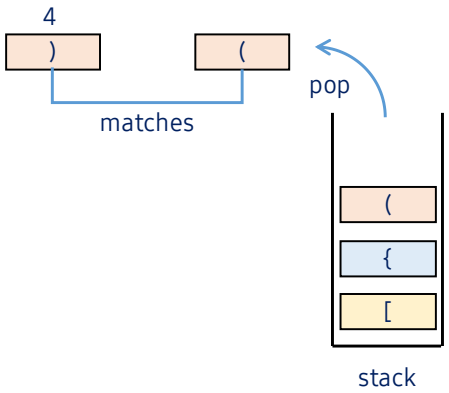


1. Balanced Parentheses Problem

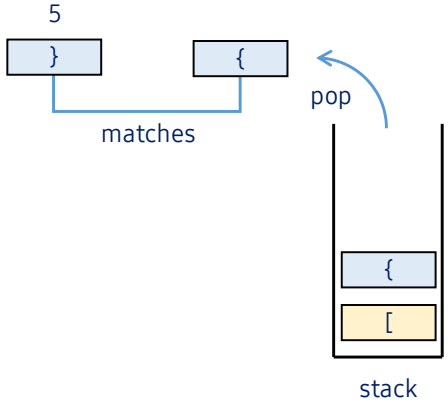
1.4. Solving the problem using a stack



4. When there is a close parenthesis, pop out the parenthesis from the stack for matching.

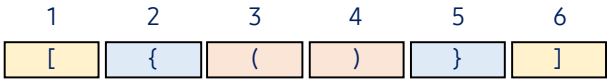


5. When there is a close parenthesis, pop out the parenthesis from the stack for matching.

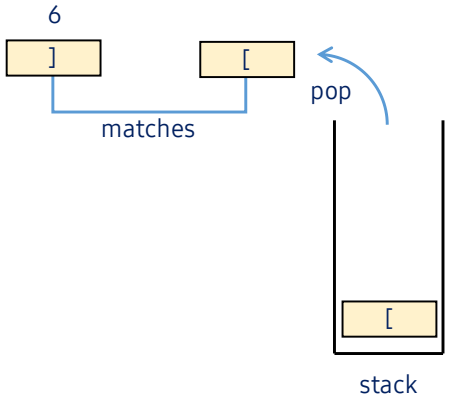


1. Balanced Parentheses Problem

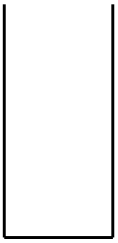
1.4. Solving the problem using a stack



6. When there is a close parenthesis, pop out the parenthesis from the stack for matching.



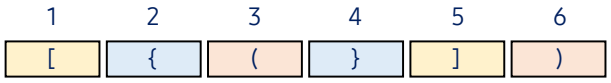
7. If the stack is empty after checking all parentheses of the string, then you get balanced parentheses.



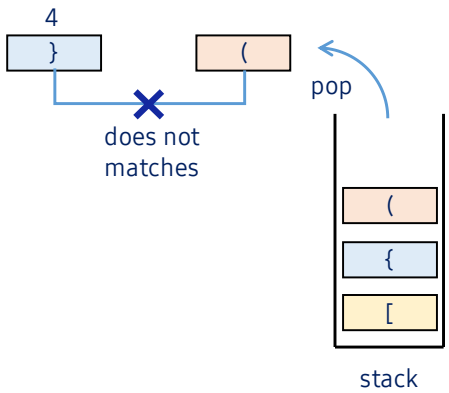
1. Balanced Parentheses Problem

1.4. Solving the problem using a stack

The following shows unmatched parentheses.



Case 1. The close parenthesis does not match with the parenthesis popped out of the stack.

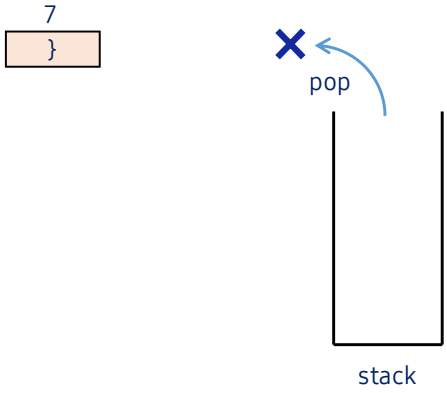


1. Balanced Parentheses Problem

1.4. Solving the problem using a stack

The following shows unmatched parentheses.

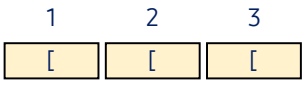
Case 2. If there is a close parenthesis but the stack is empty.



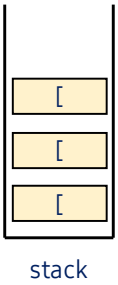
1. Balanced Parentheses Problem

1.4. Solving the problem using a stack

The following shows unmatched parentheses.



Case 3. If the stack is not empty even after checking all string.



1. Balanced Parentheses Problem

1.5. Implementation and coding

Define the function as follows to check balanced parentheses.

```
1  def check_parentheses(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = Stack()
5      for char in expr:
6          if char in opening:
7              stack.push(char)
8          elif char in closing:
9              if stack.is_empty():
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return stack.is_empty()
```

Line 1

- The chek_parentheses() function receives the 'expr' string as input parameter.
- This function returns True if there are matching parentheses in the expr string, and if not, it returns False.

1. Balanced Parentheses Problem

1.5. Implementation and coding

Initialize the data structure required for algorithm implementation as follows.

```
1  def check_parentheses(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = Stack()
5      for char in expr:
6          if char in opening:
7              stack.push(char)
8          elif char in closing:
9              if stack.is_empty():
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return stack.is_empty()
```

Line 2-4

- The 'opening' and 'closing' initializes open and close parentheses respectively. Notice that the matched parentheses are located in the same position of the string. The string index will check for matches.
- The stack variable is initialized to the object instance of the stack class that was previously defined.

1. Balanced Parentheses Problem

1.5. Implementation and coding

Implement a routine that processes open parentheses of each character in the given string.

```
1  def check_parentheses(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = Stack()
5      for char in expr:
6          if char in opening:
7              stack.push(char)
8          elif char in closing:
9              if stack.is_empty():
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return stack.is_empty()
```

Line 5-7

- Check every character (char) in the expr string.
- If the char is open parenthesis, push it to the stack.

1. Balanced Parentheses Problem

1.5. Implementation and coding

Implement a routine to process close parentheses as follows.

```
1  def check_parentheses(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = Stack()
5      for char in expr:
6          if char in opening:
7              stack.push(char)
8          elif char in closing:
9              if stack.is_empty():
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return stack.is_empty()
```

Line 8-12

- If the 'char' is close parenthesis rather than an open parenthesis, check if it matches with another parenthesis.
- The parentheses do not match if there's a close parenthesis and empty stack. (Case 2)
- The parentheses do not match if there's a close parenthesis and the character position taken from the stack and string position of char are different. (Case 1)

1. Balanced Parentheses Problem

1.5. Implementation and coding

After checking all of the string, return the results as follows.

```
1  def check_parentheses(expr):
2      opening = "[{("
3      closing = "]})"
4      stack = Stack()
5      for char in expr:
6          if char in opening:
7              stack.push(char)
8          elif char in closing:
9              if stack.is_empty():
10                 return False
11             if opening.index(stack.pop()) != closing.index(char):
12                 return False
13     return stack.is_empty()
```

Line 13

- The parentheses do not match if the stack is not empty after checking all characters of the expr string. (Case 3)
- If the stack is empty, there are matching parentheses so return the result of the empty stack.

1. Balanced Parentheses Problem

1.5. Implementation and coding

Examine various inputs of the check_parentheses() function through the following code.

```
1  str = input("Input a string of parentheses: ")
2  print("Valid" if check_parentheses(str) else "Invalid")
```

```
Input a string of parentheses:
```

Pop quiz

Pop quiz

UNIT 22

Quiz. #1

Write the output of the codes from the example that uses a stack.

```
stack = Stack()
stack.push("Banana")
stack.push("Apple")
stack.push("Tomato")
stack.pop()
stack.push("Strawberry")
stack.push("Grapes")
stack.pop()
print(stack.stack)
```

Quiz. #2

Write the output of the codes from the example that uses a stack.

```
stack = Stack()
items = [10 * i for i in range(1, 10)]
for item in items:
    stack.push(item)
    if (item // 10) % 2 == 0:
        stack.pop()
print(stack.stack)
```


Pair programing

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair programming UNIT 22

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. The HTML document consists of many tags as shown below. Write a program that matches HTML document tags.

Matching HTML Tags

```
<html>
<body>
<h1>Hello, World!</h1>
<p> We are learning the art of coding
with Python programming language.
Here we are learning ... </p>
<ul>
<li> Data Structures, </li>
<li> Algorithms, </li>
<li> and Computational Thinking,
    eventually. </li>
</ul>
</body>
</html>
```

Hello, World!

We are learning the art of coding with Python programming language. Here we are learning ...

- Data Structures,
- Algorithms,
- and Computational Thinking, eventually.

Reference

An HTML element is defined by a start tag, some content, and an end tag.

HTML Elements

The HTML **element** is everything from the start tag to the end tag:

```
<tagname>Content goes here...</tagname>
```

Examples of some HTML elements:

```
<h1>My First Heading</h1>

<p>My first paragraph.</p>
```

https://www.w3schools.com/html/html_elements.asp

TIP

- In Python, the find() method is used to locate the start position of the substring. Find the '<' and '>' to distinguish HTML document tags.
- If the distinguished tag includes '/,' then it is an open tag, and if not, it is a close tag. Check the HTML tag matching in a similar way of using a stack to match parentheses.

```
1  text = input("Input the string of HTML document: ")
```

```
Input the string of HTML document: <html> <body> <h1>Hello, World!</h1> <p> We are learning the art of coding
with Python programming language. Here we are learning ... </p>  <ul> <li> Data Structures, </li> <li> Algorit
hms, </li> <li> and Computational Thinking,      eventually. </li>  </ul> </body> </html>
```

```
1  start = text.find('<')
2  while start != -1:
3      end = text.find('>', start + 1)
4      tag = text[start:end+1]
5      print(tag, end = ' ')
6      start = text.find('<', end + 1)
```

```
<html> <body> <h1> </h1> <p> </p> <ul> <li> </li> <li> </li> <li> </li> </ul> </body> </html>
```


Unit 23.

Queue

Unit learning objective (1/3)

Learning objectives

- ✓ Understand the concept of a queue and be able to define a queue as abstract data type.
- ✓ Be able to implement the queue data structure into a class by using python language.
- ✓ Be able to apply the queue data structure to solve Josephus problem.

Unit learning objective (2/3)

Learning overview

- ✓ Learn how to define the queue data structure as a class
- ✓ Learn how to use the queue class with its methods
- ✓ Learn how to solve the Josephus problem by using the queue data structure

Concepts You Will Need to Know From Previous Units

- ✓ How to declare a list, add and delete items
- ✓ How to define class constructor and define variables and methods
- ✓ How to create class instance to approach to the variables and call methods

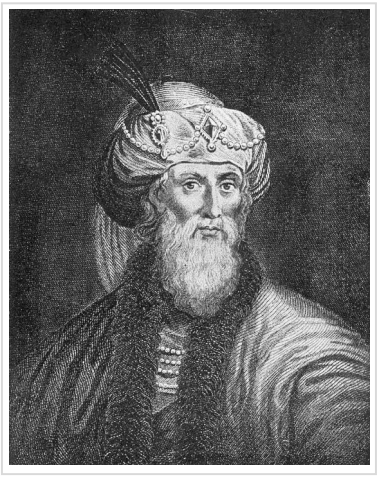
Keywords

- Abstract Data Type
- Queue
- FIFO
- enqueue
- dequeue

Mission

1. Real world problem

1.1. Josephus story in the history



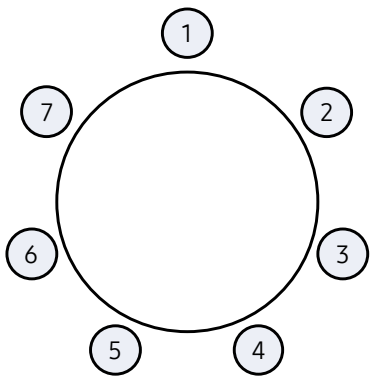
https://en.wikipedia.org/wiki/Josephus#/media/File:Josephus.jpg

- The Josephus story in the history has become an interesting coding problem.
- Josephus was a Jewish historian in the 1st century. After being defeated from the war with Rome, all of the 41 people including him escaped into a cave.
- In the cave, they stood in a circle and made an agreement that every 3rd person to commit suicide until no one survives.
- Josephus thought killing themselves was meaningless, so he did calculation and occupied the position that will allow him to survive until the end. Then, the story says that he finally survived by convincing other remaining people.

1. Real world problem

1.1. Josephus story in the history

- Let's say a total of 7 people are sitting around the table.
- Each of them is given with a number from 1 to 7 in clockwise. The following method will be used to select a representative.



- Exclude every 3rd person from the table by starting with no.1 in clockwise.
- Repeat the same method from the person who sits next to the one that was previously excluded.

- If the above method is used to select one representative that remains the last, then who will be the one?

1. Real world problem

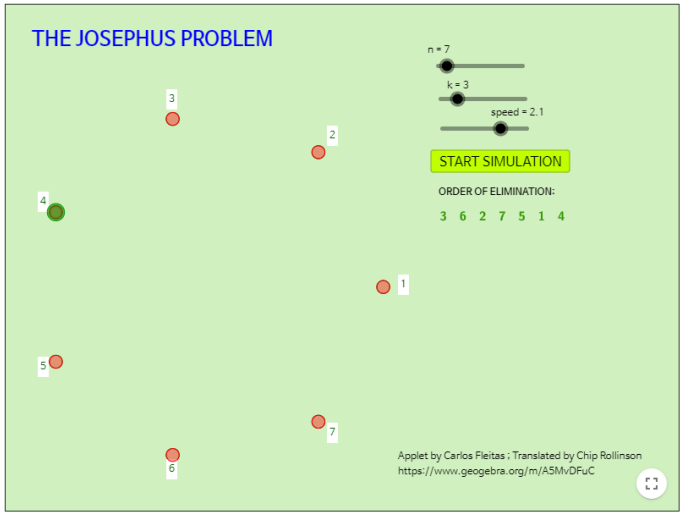
1.1. Josephus story in the history

Josephus problem can be defined as follows.

- N people gather around a table and each of them receives a number from 1 to N in clockwise.
- Select K that is smaller or identical to N, and exclude the Kth person from no.1.
- Exclude the Kth person in clockwise until $(N-1)$ people are eliminated.
- Calculate the number of the person who will survive until the last when two positive integers N and $K(\leq N)$ are given.

1. Real world problem

1.2. Josephus sequence generator



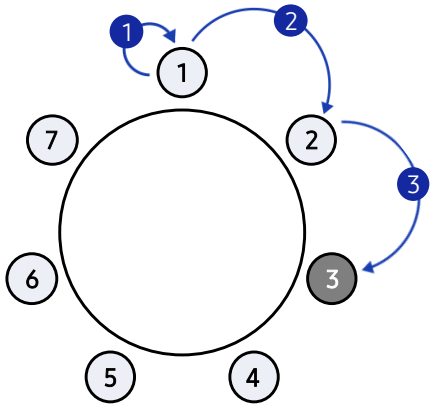
https://www.geogebra.org/m/ExvvrBbR

- GeoGebra provides an applet for simulating Josephus problem.
- In the history, Josephus participated in a game in which every 3rd person kills oneself among 41 people sitting in a circle.
- So, in Josephus problem, N=41 and K=3. Then, which number did Josephus have?
- Josephus survived with another person who also survived until the last. Then, what would be his number?
- Use the applet to check those two people who survived from the Josephus story.

2. Mission

2.1. Josephus problem

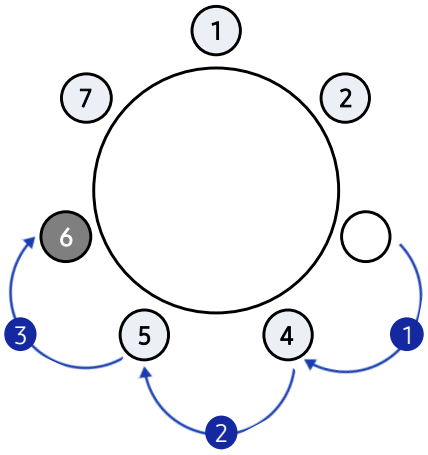
- The first problem that we had is Josephus problem where N=7, K=3.
- To solve this problem, start with no.1 where 7 people are sitting around the table and exclude every 3rd person.



2. Mission

2.1. Josephus problem

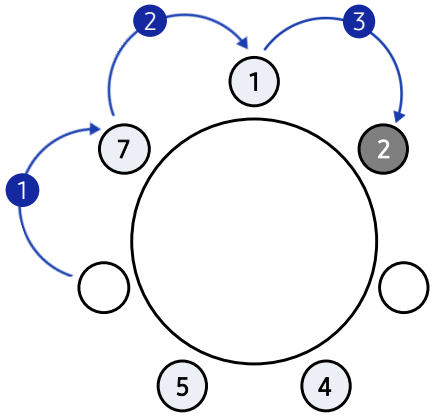
- After no.3 is eliminated, the no.6 who is the 3rd person from no.4 should be excluded.



2. Mission

2.1. Josephus problem

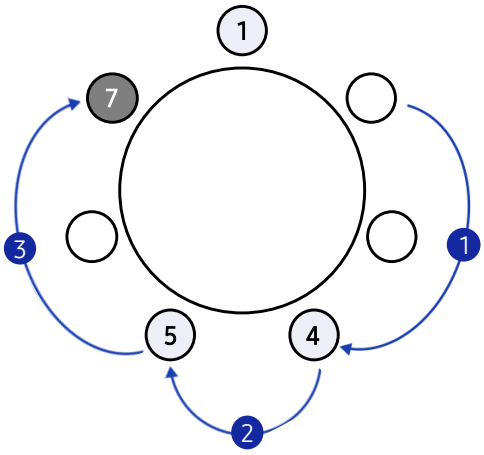
- Once no.6 is eliminated, then the next 3rd person is no.2.
- Exclude no.2 from the table.



2. Mission

2.1. Josephus problem

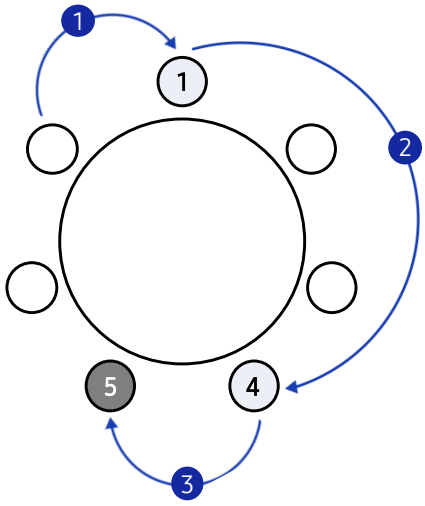
- Skip the person who is already excluded.
- After no.2 is eliminated, the next 3rd person to be excluded is no.7. Eliminate no.7.



2. Mission

2.1. Josephus problem

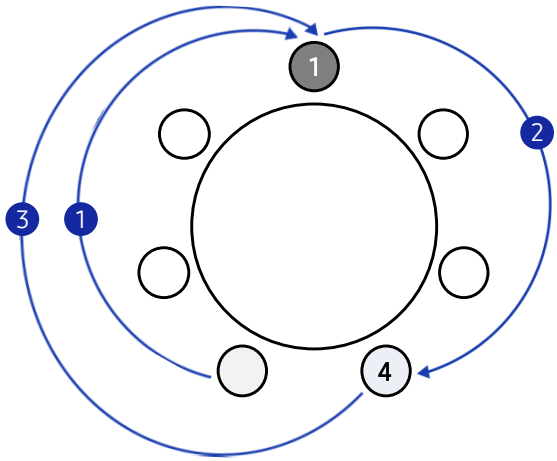
- The same principle applies, and the next person to be eliminated next to no.7 is no.5.
- Exclude no.5.



2. Mission

2.1. Josephus problem

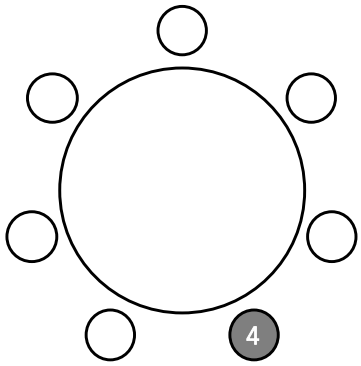
- Only two people remain, so when starting from no.1, the 3rd person becomes no.1.
- Thus, eliminate no.1.



2. Mission

2.1. Josephus problem

After excluding no.1, only one person remains. The final representative is no.4.



3. Solution

3.1. How to solve the Josephus problem

Receive N and K as user input to print Josephus sequence.

```
1 N = int(input("Input the number of people(N): "))
2 K = int(input("Input the number to be skipped(K): "))
3 print(josephus_sequence(N, K))
```

```
Input the number of people(N): 7
Input the number to be skipped(K): 3
[3, 6, 2, 7, 5, 1, 4]
```

3. Solution

3.2. Josephus problem final code

```python
def josephus_sequence(n, k):
    sequence = []
    queue = []
    for i in range(1, n + 1):
        queue.append(i)
    j = 1
    while len(queue) > 1:
        item = queue.pop(0)
        if j % k == 0:
            sequence.append(item)
        else:
            queue.append(item)
        j += 1
    sequence.append(queue.pop(0))
    return sequence
```


Key concept

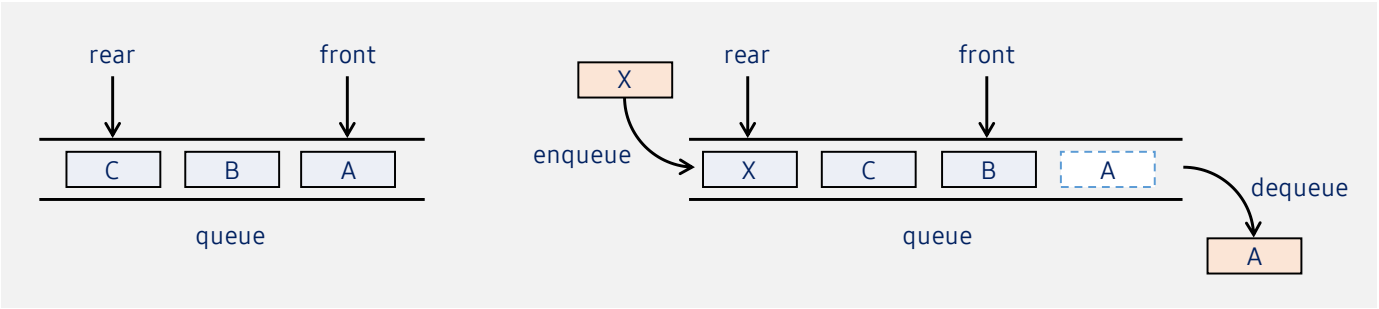
1. What is a Queue?

1.1. Definition of a queue

- A queue is an abstract data type where only deletion is possible at the front and only addition is possible at the rear.
- Because the first added item is also removed the first, a queue is a data structure that works with the FIFO (First-In-First-Out) method.

Focus A queue should provide two important interfaces.

- enqueue: adds a new item to the rear of the queue
- dequeue: deletes an item from the front of the queue



1. What is a Queue?

1.2. Examples of a queue in our life

- A queue is frequently found in our life.
- For example, for flight booking at an airport, people need to wait in a queue, and the first person who entered the line receives the service first.
- Other than this, queues are commonly found in restaurants and banks.

2. Implement a Stack

2.1. Define a queue as a class type

In Python, list data type can be used to define the queue data structure as a class.

```
1  class Queue:
2
3      def __init__(self):
4          self.queue = []
5
6      def is_empty(self):
7          return True if len(self.queue) == 0 else False
8
9      def enqueue(self, item):
10         self.queue.append(item)
11
12     def dequeue(self):
13         return None if self.is_empty() else self.queue.pop(0)
```


2. Implement a Stack

2.1. Define a queue as a class type

Similar to the stack class, the queue class defines a queue as a list data type through the constructor method.

```
1  class Queue:
2
3      def __init__(self):
4          self.queue = []
5
6      def is_empty(self):
7          return True if len(self.queue) == 0 else False
8
9      def enqueue(self, item):
10         self.queue.append(item)
11
12     def dequeue(self):
13         return None if self.is_empty() else self.queue.pop(0)
```

Line 1-4

- Define the queue data structure as a queue class.
- The __init__(self) constructor initiates the queue member field into an empty list.

2. Implement a Stack

2.1. Define a queue as a class type

The 'enqueue' can be implemented as a method of the queue class as shown below.

```
1  class Queue:
2
3      def __init__(self):
4          self.queue = []
5
6      def is_empty(self):
7          return True if len(self.queue) == 0 else False
8
9      def enqueue(self, item):
10         self.queue.append(item)
11
12     def dequeue(self):
13         return None if self.is_empty() else self.queue.pop(0)
```

Line 9-10

- The enqueue() method receives the 'self' and item to be added to the queue as parameters.
- The 'enqueue' adds an item to the end of the self.queue list which is the queue class member field.

2. Implement a Stack

2.1. Define a queue as a class type

The 'dequeue' can be implemented as a method of the stack class as shown below.

```
1  class Queue:
2
3      def __init__(self):
4          self.queue = []
5
6      def is_empty(self):
7          return True if len(self.queue) == 0 else False
8
9      def enqueue(self, item):
10         self.queue.append(item)
11
12     def dequeue(self):
13         return None if self.is_empty() else self.queue.pop(0)
```

Line 12-13

- The dequeue() method receives 'self' as a parameter.
- The 'dequeue' returns None if self.is_empty() is True, and if not, it returns by deleting the last element of the self.queue.

2. Implement a Stack

2.1. Define a queue as a class type

The is_empty() method that is used to check for an empty queue should be also defined as the queue class method.

```
1  class Queue:
2
3      def __init__(self):
4          self.queue = []
5
6      def is_empty(self):
7          return True if len(self.queue) == 0 else False
8
9      def enqueue(self, item):
10         self.queue.append(item)
11
12     def dequeue(self):
13         return None if self.is_empty() else self.queue.pop(0)
```

Line 6-7

- The is_empty() method returns True if the self.queue length is 0, and if not, it returns False.

2. Implement a Stack

2.1. Define a queue as a class type

The following code tests the queue data structure that is defined as a class.

```
1  queue = Queue()
2  queue.enqueue("A")
3  queue.enqueue("B")
4  print(queue.dequeue())
5  queue.enqueue("C")
6  print(queue.dequeue())
7  print(queue.dequeue())
8  print(queue.dequeue())
```

```
A
B
C
None
```

Line 1-8

- The queue variable saves the object instance of the queue class that is created by calling the Queue() constructor.
- The enqueue() and dequeue() methods of the queue object can be called by using the reference operator '.'
- Print the returned value of the queue.dequeue() function as print() to check the order of removal from the queue.

One More Step

- What would you do if you want to know the number of elements in the queue?
- The size() method that provides current number of elements in the queue can be implemented by returning the queue list length of the current queue class.

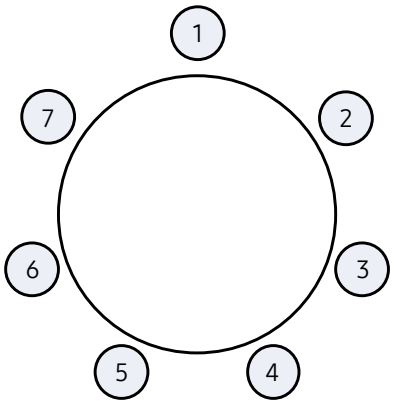
```
class Queue:

    # ......

    def size(self):
        return len(self.queue)
```


Let's code

1. Josephus Problem

1.1. Problem definition

- Calculate number of people being excluded when applying the following rule to two natural numbers N and K.
  - *N* people gather around a table and each of them is assigned with a number from 1 to *N* in clockwise.
  - For K that is smaller or identical to N, exclude the Kth person starting from no.1.
  - Eliminate Kth person in clockwise until everyone is excluded.

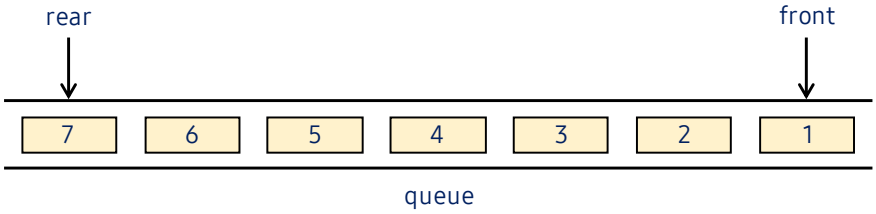


If N=7, K=3,
the order of elimination is [3, 6, 2, 7, 5, 1, 4]

1. Josephus Problem

1.2. Solving the problem using a queue

- Use the queue data structure to easily solve the Josephus problem.
- First, enqueue from 1 to N to the queue in order.
- If N=7, it will look like as follows.
- Because nothing has been eliminated yet, the sequence is defined as an empty list.

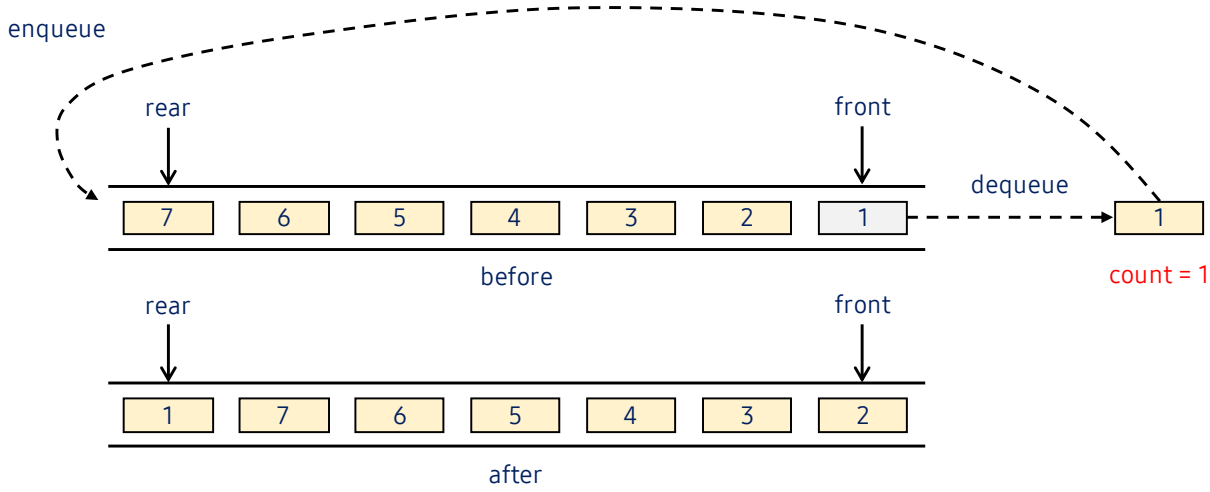


sequence = []

1. Josephus Problem

1.2. Solving the problem using a queue

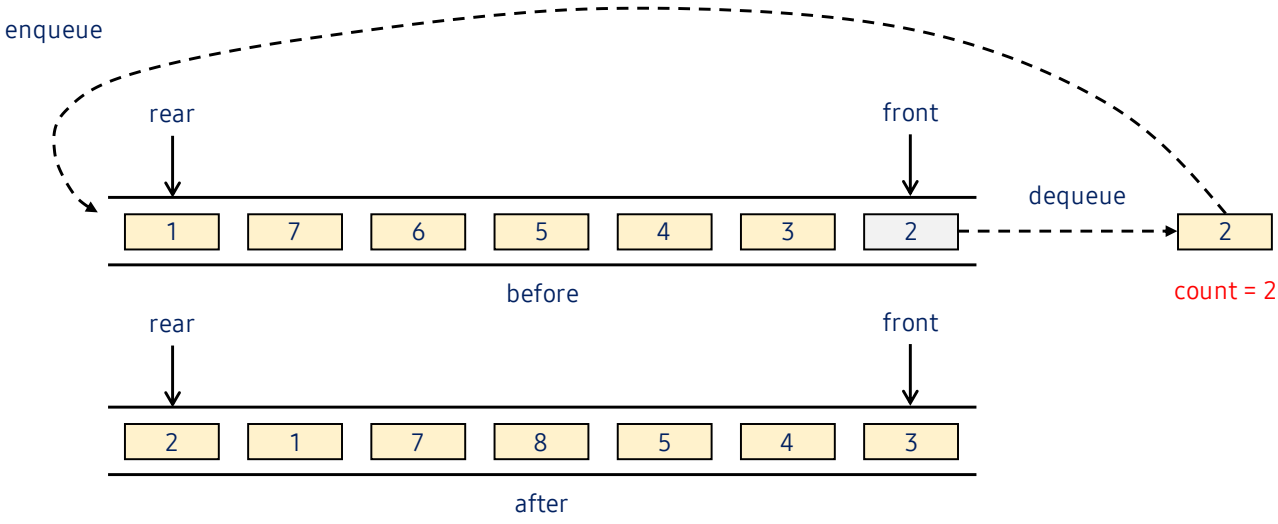
- Take one element from the queue and add it back to the queue. However, do not add the Kth element back to the queue.
- Start by taking the first element and add no.1 back to the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

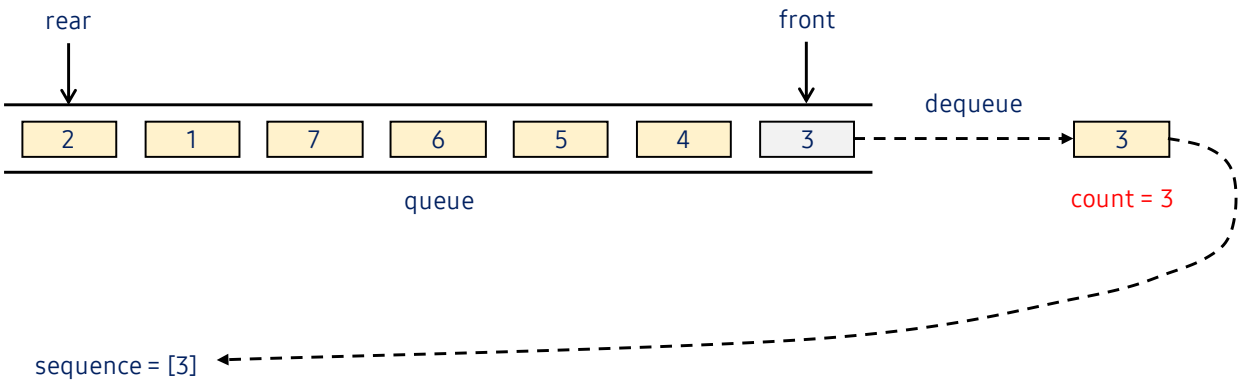
- When taking the 2nd element, add no.2 back to the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

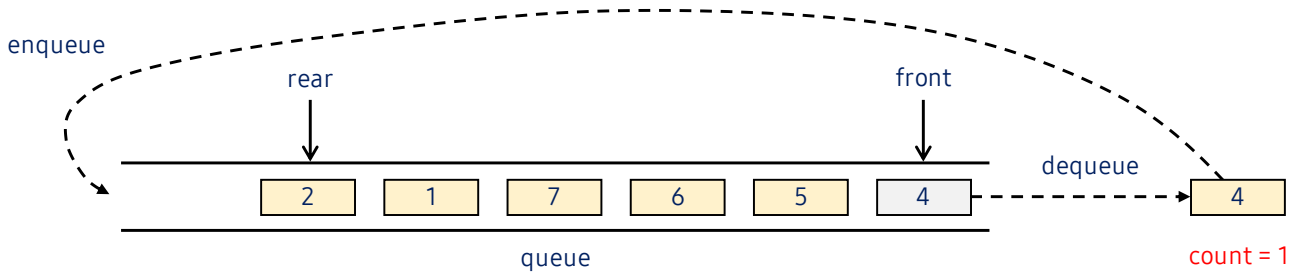
- When taking the 3rd element, eliminate it without adding no.3 back to the queue.(K=3).
- The eliminated no.3 should be added to the sequence list.

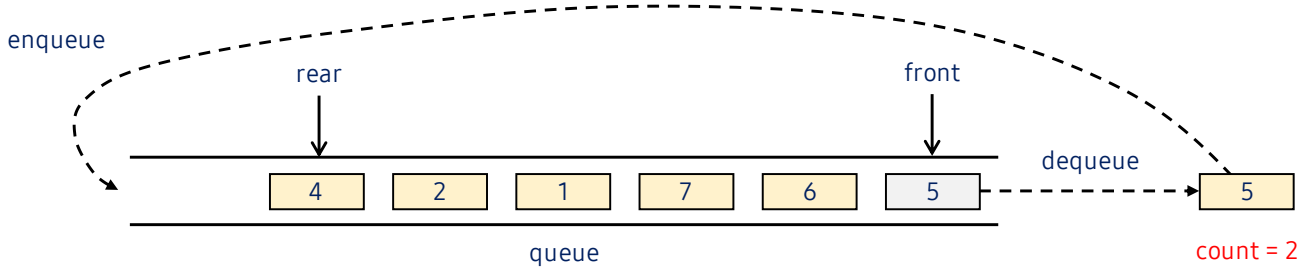


1. Josephus Problem

1.2. Solving the problem using a queue

- Next, take the elements no. 4 and 5 from the queue, then add them back to it.

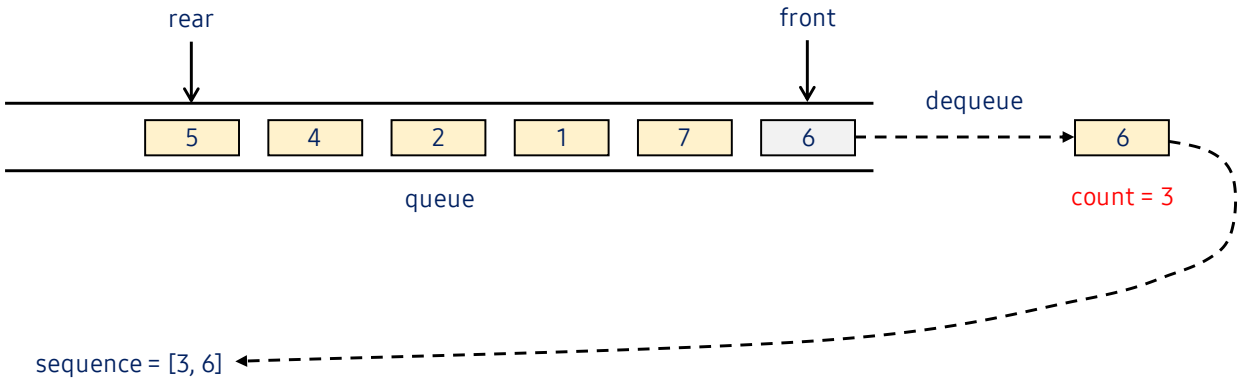




1. Josephus Problem

1.2. Solving the problem using a queue

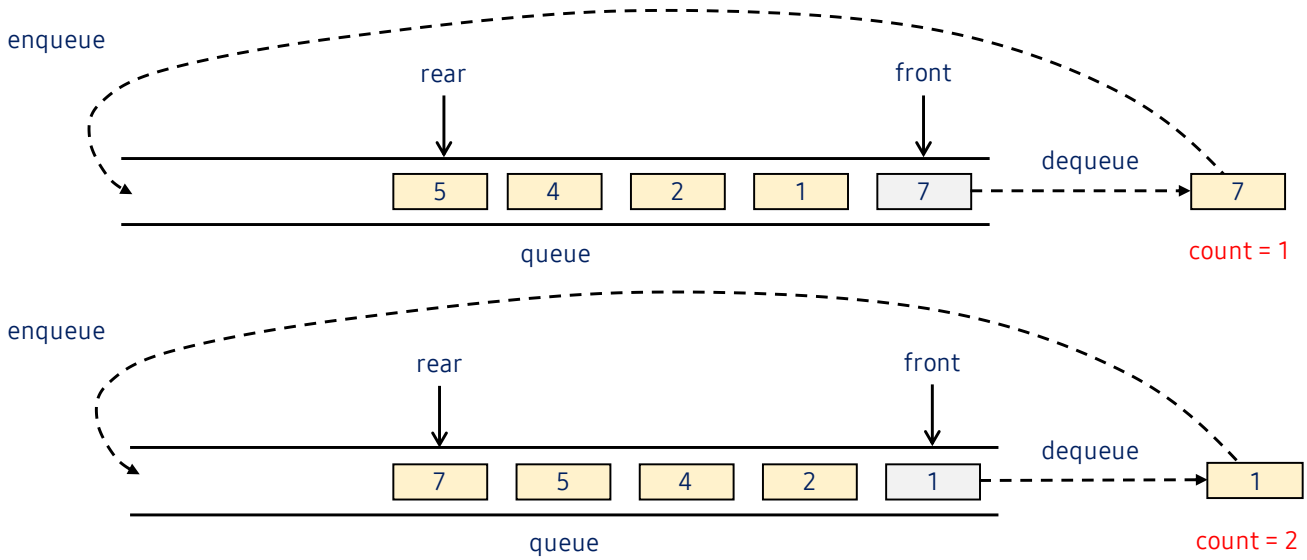
No.6 is the 3rd element, so add it to the sequence list without adding it back to the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

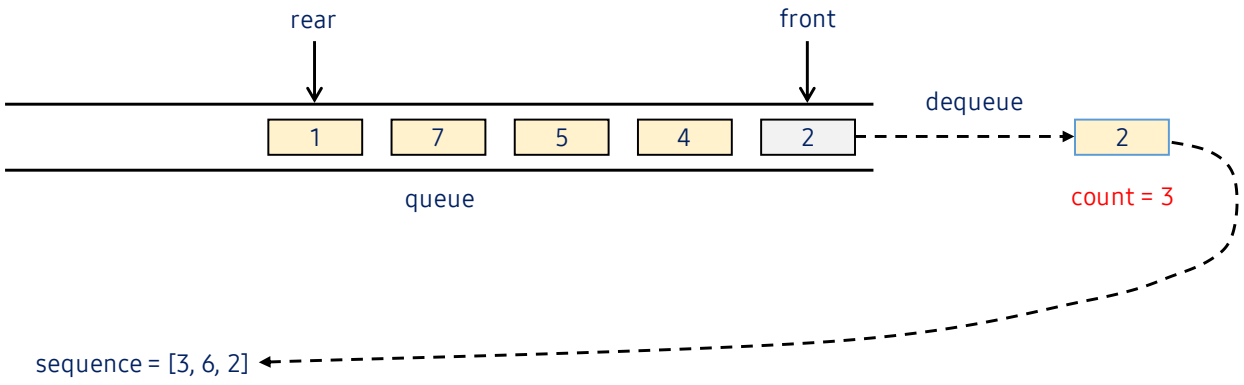
After taking no.7 and 1, add them back to the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

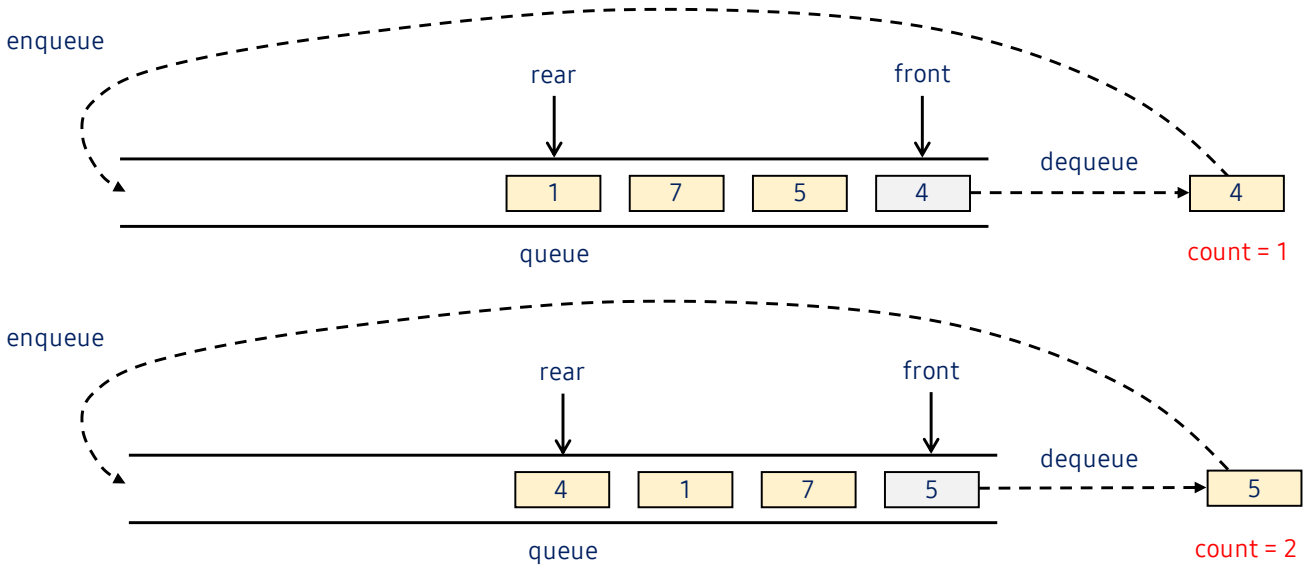
No.2 is the element where K=3, so add it to the sequence list, not to the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

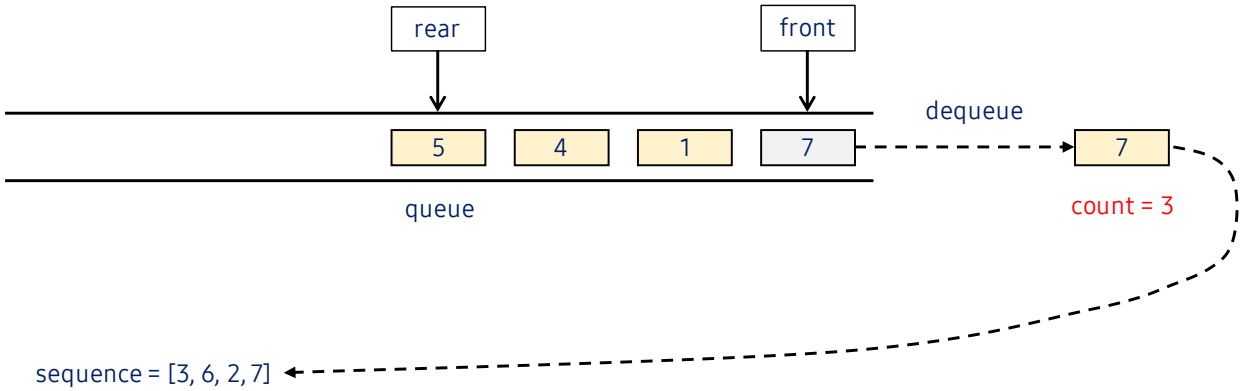
Repeat the same process until one element remains in the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

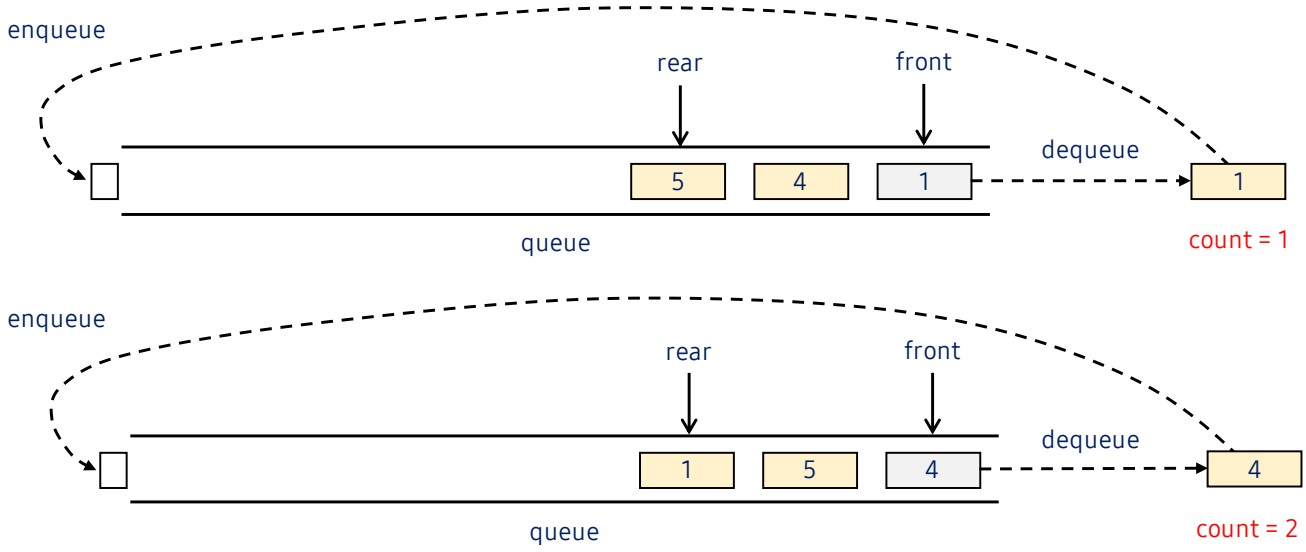
Repeat the same process until one element remains in the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

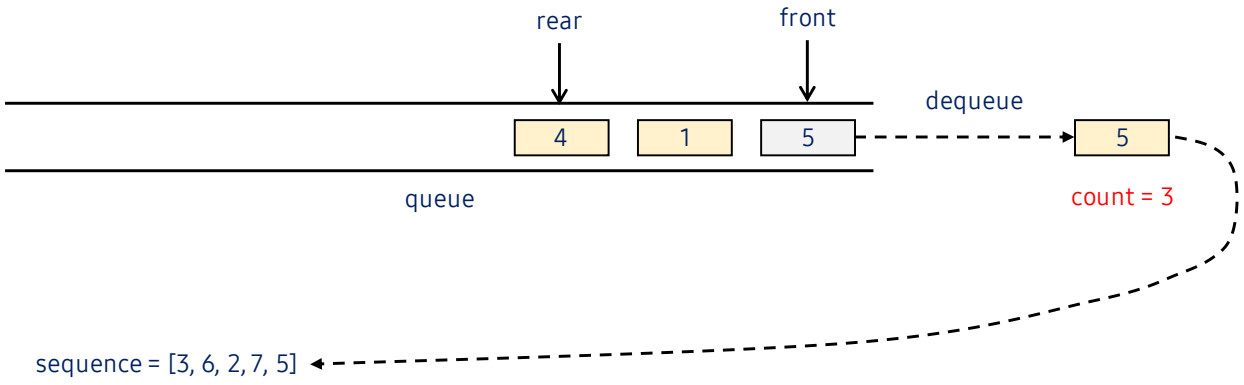
Repeat the same process until one element remains in the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

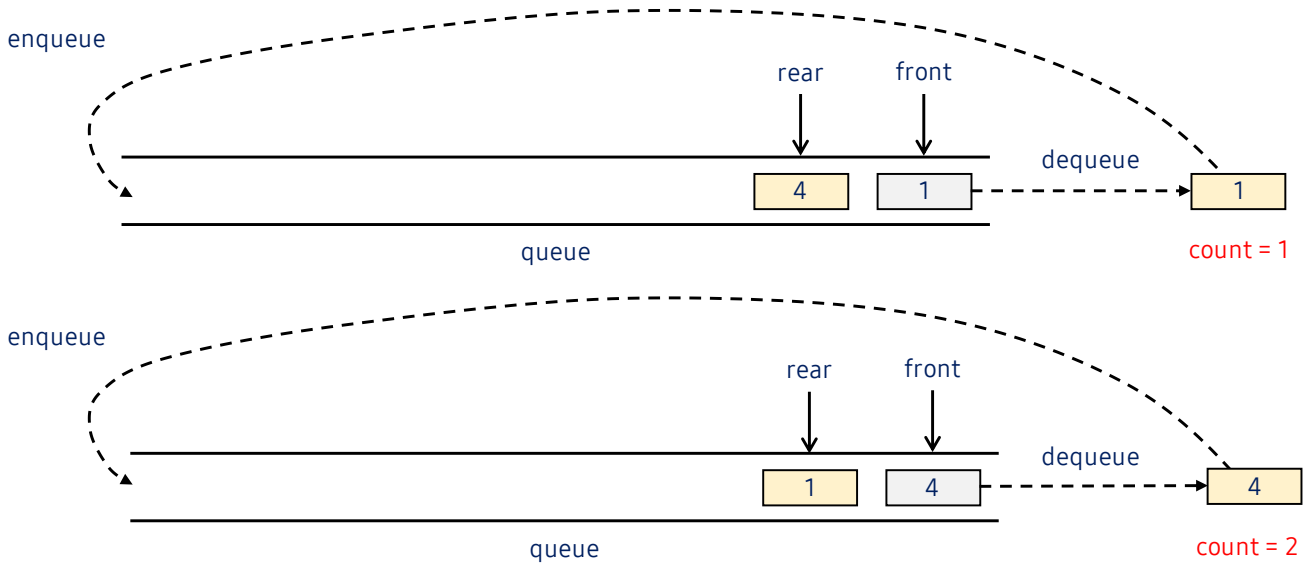
Repeat the same process until one element remains in the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

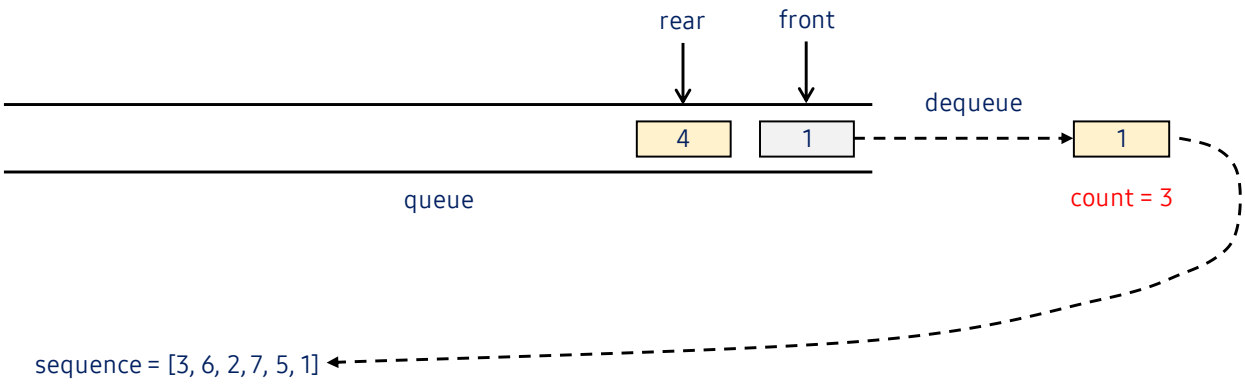
Repeat the same process until one element remains in the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

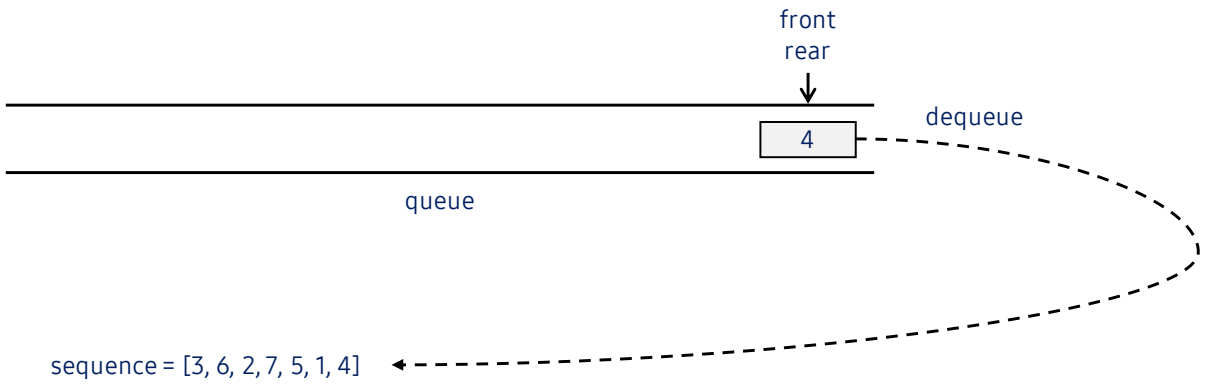
Repeat the same process until one element remains in the queue.



1. Josephus Problem

1.2. Solving the problem using a queue

If only one element remains in the queue, dequeue it and add it as the last element of the sequence list.



1. Josephus Problem

1.3. Implementation and coding

Implement the previous steps into Python codes. First, the josepush() function receives integers n and k as input parameters.

```
1  def josephus(n, k):
2      sequence = []
3      queue = Queue()
4      for i in range(1, n + 1):
5          queue.enqueue(i)
6      j = 1
7      while queue.size() > 1:
8          item = queue.dequeue()
9          if j % k == 0:
10             sequence.append(item)
11         else:
12             queue.enqueue(item)
13         j += 1
14     sequence.append(queue.dequeue())
15     return sequence
```

Line 1-3

- The josephus() function returns Josephus permutation into a list through the input parameters n and k.
- The returning sequence is initialized as an empty list, while the queue is initialized as an object of the queue class that was previously defined.

1. Josephus Problem

1.3. Implementation and coding

```
1  def josephus(n, k):
2      sequence = []
3      queue = Queue()
4      for i in range(1, n + 1):
5          queue.enqueue(i)
6      j = 1
7      while queue.size() > 1:
8          item = queue.dequeue()
9          if j % k == 0:
10             sequence.append(item)
11         else:
12             queue.enqueue(item)
13         j += 1
14     sequence.append(queue.dequeue())
15     return sequence
```

Line 4-5

- Enqueue integers from 1 to N to the first queue.

1. Josephus Problem

1.3. Implementation and coding

```
1  def josephus(n, k):
2      sequence = []
3      queue = Queue()
4      for i in range(1, n + 1):
5          queue.enqueue(i)
6      j = 1
7      while queue.size() > 1:
8          item = queue.dequeue()
9          if j % k == 0:
10             sequence.append(item)
11         else:
12             queue.enqueue(item)
13         j += 1
14     sequence.append(queue.dequeue())
15     return sequence
```

Line 6-7, 13

- 'j' is a counter that takes an item out of the queue. Thus, it processes initialization with 1.
- Then, increase the j value by 1 to repeat enqueue and dequeue until only one item remains in the queue.
- If the remainder of dividing j with k is 0, then this item is the next kth one which will be dequeued.

Let's code UNIT 23

1. Josephus Problem

1.3. Implementation and coding

```
1  def josephus(n, k):
2      sequence = []
3      queue = Queue()
4      for i in range(1, n + 1):
5          queue.enqueue(i)
6      j = 1
7      while queue.size() > 1:
8          item = queue.dequeue()
9          if j % k == 0:
10             sequence.append(item)
11         else:
12             queue.enqueue(item)
13         j += 1
14     sequence.append(queue.dequeue())
15     return sequence
```

Line 8-12

- Dequeue one item from the queue.
- If j % k == 0, then it is the kth item, so add an element to the end of the sequence.
- If j % k != 0, then this item won't be dequeued, so add it back to the queue.
- Repeat the process until only one element remains in the queue, and the dequeued elements are saved in the sequence list.

1. Josephus Problem

1.3. Implementation and coding

```
1  def josephus(n, k):
2      sequence = []
3      queue = Queue()
4      for i in range(1, n + 1):
5          queue.enqueue(i)
6      j = 1
7      while queue.size() > 1:
8          item = queue.dequeue()
9          if j % k == 0:
10             sequence.append(item)
11         else:
12             queue.enqueue(item)
13         j += 1
14     sequence.append(queue.dequeue())
15     return sequence
```

Line 14-15

- Once the while loop is complete, only one element remains in the queue, which will be dequeued for the last.
- Thus, it is possible to print the sequence by taking the element from the queue and adding it to the end of the sequence.

1. Josephus Problem

1.3. Implementation and coding

Apply the previous example to the josephus() function to obtain the following results.

```
1  N = int(input("Input the number of people(N): "))
2  K = int(input("Input the number to be skipped(K): "))
3  print(josephus(N, K))
```

```
Input the number of people(N): 7
Input the number to be skipped(K): 3
[3, 6, 2, 7, 5, 1, 4]
```

```
1  N = int(input("Input the number of people(N): "))
2  K = int(input("Input the number to be skipped(K): "))
3  print(josephus(N, K))
```

```
Input the number of people(N): 41
Input the number to be skipped(K): 3
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 1, 5, 10, 14, 19, 23, 28, 32, 37, 41, 7, 13, 20, 26, 34, 40,
8, 17, 29, 38, 11, 25, 2, 22, 4, 35, 16, 31]
```

Pop quiz

Pop quiz

UNIT 23

Quiz. #1

Write the output of the codes from the example that uses a stack.

```
queue = Queue()
queue.enqueue("Banana")
queue.enqueue("Apple")
queue.enqueue("Tomato")
queue.dequeue()
queue.enqueue("Strawberry")
queue.enqueue("Grapes")
queue.dequeue()
print(queue.queue)
```

Quiz. #2

Write the output of the codes from the example that uses a stack.

```
queue = Queue()
items = [10 * i for i in range(1, 11)]
for item in items:
    queue.enqueue(item)
    if (item // 10) % 2 == 0:
        queue.dequeue()
print(queue.queue)
```


Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair programming UNIT 23

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

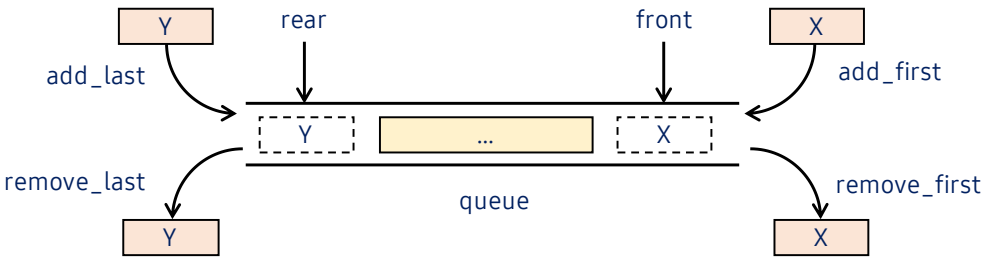
A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Refer to the class definition as follows to complete and test the Deque class.

```
class Deque:

    def __init__(self):
        self.queue = []

    def add_first(self, item):
        ''' Add an item to the front of the queue '''

    def remove_first(self):
        ''' Remove and return the item from the front '''

    def add_first(self, item):
        ''' Add an item to the rear of the queue '''

    def remove_first(self):
        ''' Remove and return the item from the rear '''
```


TIP

A queue is an abstract data type where only addition is possible on one side and only removal is possible on the other side. In contrast, the Double Ended Queue (Deque) is the data structure where addition and removal are possible on both sides.



Unit 24.

Sequential Search

Unit learning objective (1/3)

Learning objectives

- ✓ Understand searching problems and be able to solve them through sequential search.
- ✓ Be able to implement sequential search algorithms through Python language.
- ✓ Be able to apply sequential search algorithms to solve problems that find maximum and 2nd maximum values.

Learning overview

- ✓ Understand searching problems that find a specified element from randomly arranged data.
- ✓ Learn how to solve searching problems through sequential search.
- ✓ Learn how to write sequential search algorithms with Python, and analyze the best, worst, and average time complexity.

Concepts You Will Need to Know From Previous Units

- ✓ How to traverse the list elements through iterator
- ✓ How to compare the size of list elements
- ✓ How to use Big-O notation to express time complexity of the algorithm

Keywords

Searching Problem

Sequential Search

Linear Time

Mission

Mission

UNIT 24

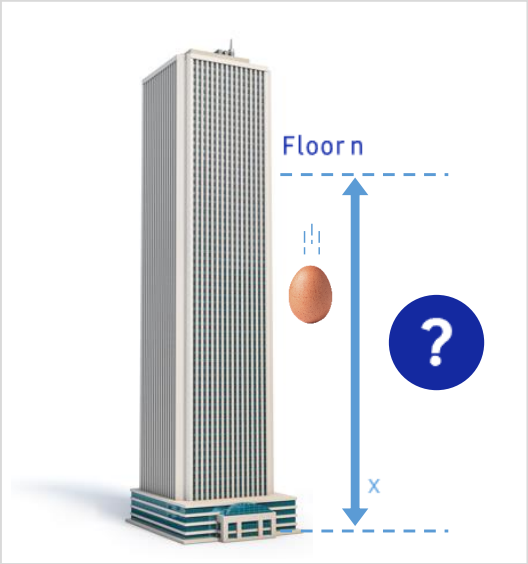
1. Real world problem

1.1. Egg dropping challenge

- The egg dropping challenge is designed for students who like science.
- Students should make an egg-protection device to prevent breaking of the egg even if falling from a high place.
- Imagine that you participated in the egg dropping challenge and made an egg-protection device with a great idea.
- How would you do if you want to calculate the highest floor that won't break your egg on a 100-floor building?
- Let's create an algorithm to solve the egg dropping problem.

2. Mission

2.1. Egg dropping problem

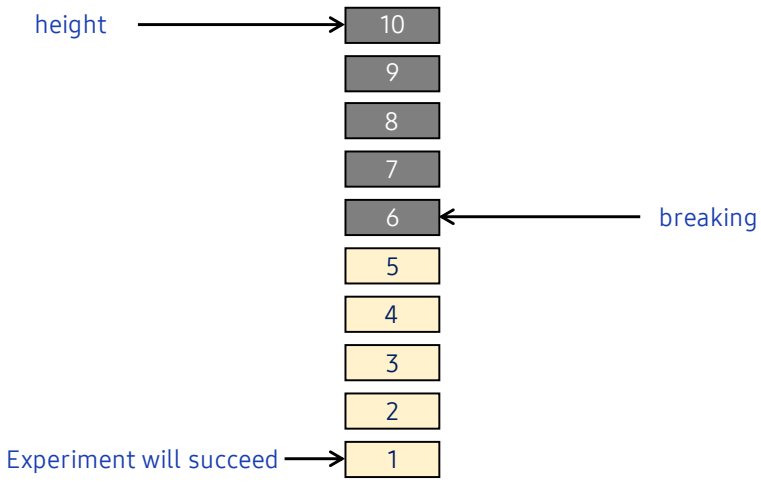


- Find the highest floor for not breaking the egg when the max. building floor is given.
- The highest floor for not breaking the egg is determined by a random number between the 1st to the max. floor of the building through a random function. Because there is only one egg, assume that you cannot proceed the experiment if it breaks.
- If the egg is not broken, you can repeat the experiment. So, test from the 1st to the max. floor and find the floor that is one less than the floor where the egg breaks for the first time.
- However, if the egg doesn't break even from the max. floor, return the max. floor value of the building, and return 0 if the egg breaks from the 1st floor.

2. Mission

2.1. Egg dropping problem

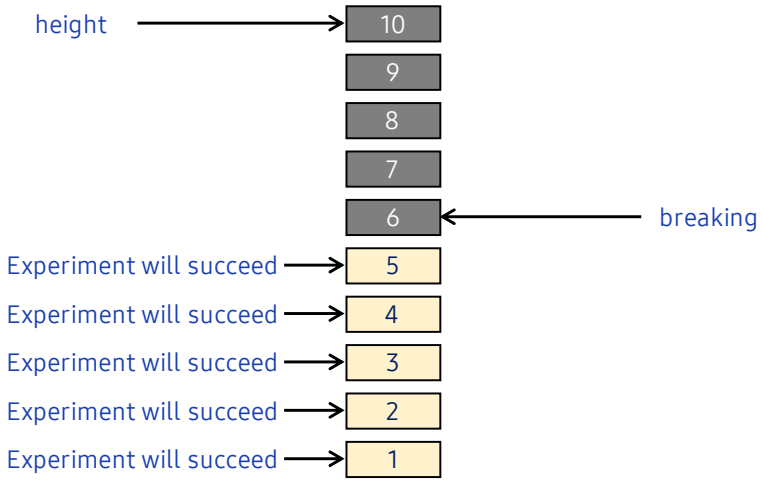
- Experiment on a building where N=10.
- Assume that the egg doesn't break when falling from the 5th floor or lower, and it breaks when falling from the 6th floor or higher. Thus, the egg won't break when doing an experiment on the 1st floor.



2. Mission

2.1. Egg dropping problem

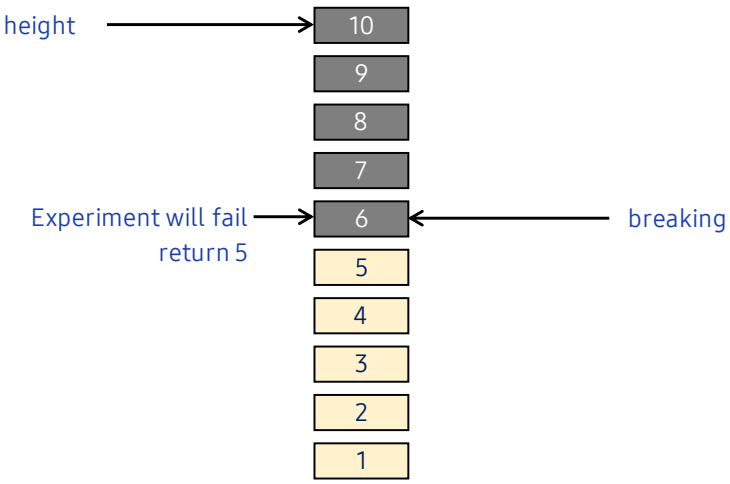
- Proceed the experiment by going up each floor, and the experiment will succeed without breaking the egg.



2. Mission

2.1. Egg dropping problem

- Doing an experiment on the 6h floor will break the egg.
- So, 5^{th} floor is the highest floor where the egg doesn't break.



3. Solution

3.1. How the solution works

The egg dropping problem finds the height where the egg breaks by receiving the building 'height' as user input.

```
from random import randint

height = int(input("Input the number of floors: "))
breaking = randint(1, height)
floor = find_highest_safe_floor(height, breaking)
print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Your egg will safe till the 94-th floor.
```


3. Solution

3.2. Egg dropping problem final code

```
def do_experiment(floor, breaking):
    return floor >= breaking

def find_highest_safe_floor(height, breaking):
    for n in range(1, height + 1):
        if do_experiment(n, breaking):
            return n - 1
    return height
```

Key concept

Key concept

UNIT 24

1. Searching Problem

1.1. Two types of searching

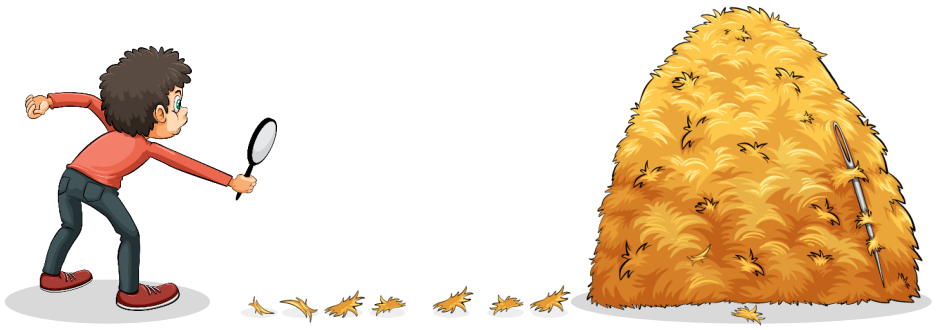
- In computer programming, one of the most frequently done processes is to find a specified item in given data.
- If the given data does not have any rules, you need to check all of it. For instance, consider a problem to find the location of a random value x from the list S where N integers are randomly listed. What if x does not exist in the given list S? You won't be able to find the location of x until checking all of the elements in S.
- If the given data has rules, the searching problem can be more efficiently solved. For example, if the given data is an ordered sequence, the binary search can be used which will be discussed later.

Sequential Search	vs.	Binary Search

1. Searching Problem

1.2. Sequential search

- Sequential search refers to an algorithm that does investigation by checking all of the items one by one in order.
- Sequential search is done for randomly given data. So, because searching is done without any rules, it is also referred to 'finding a needle in a haystack.'



2. Sequential Search

2.1. An algorithm for the sequential search

- Consider a sequential search algorithm that finds a specific integer in the given list of integers.

Focus The following shows the sequential search algorithm problem, and its input and output.

- Problem: Does a random integer x exist in the S, the set of given integers?
- Input: S as the list of integers, x as a random integer to find.
- Output: The index location if x exists in the list S, and -1 if it does not exist.

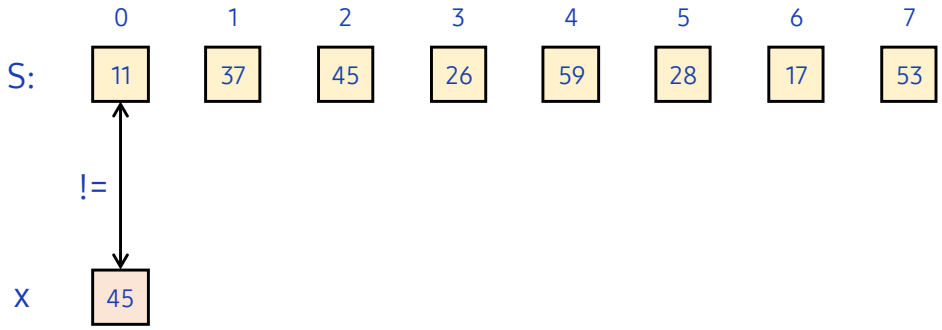
```
S = [11, 37, 45, 26, 59, 28, 17, 53]
x = 53
pos = seq_search(S, x)
print(f"In S, {x} is at position {pos}.")
```

```
In S, 53 is at position 7.
```

2. Sequential Search

2.2. An algorithm for the sequential search

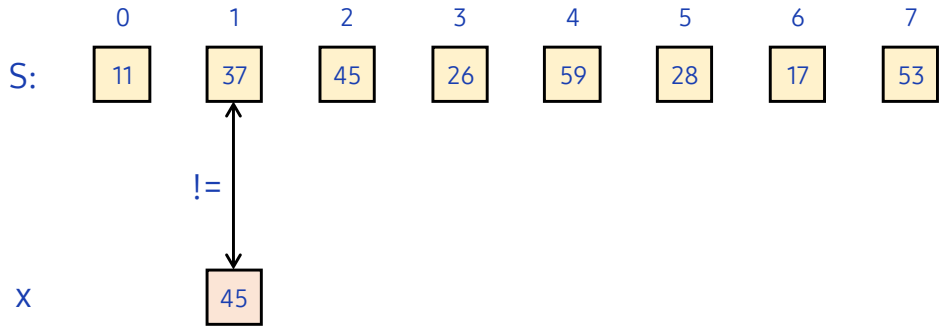
- Sequential search algorithm compares the x value to find with each of the element in the given list S.
- When starting the sequential search, it returns index 0 if the value is the same as the first element, and if not, it makes another comparison with next element.



2. Sequential Search

2.3. An algorithm for the sequential search

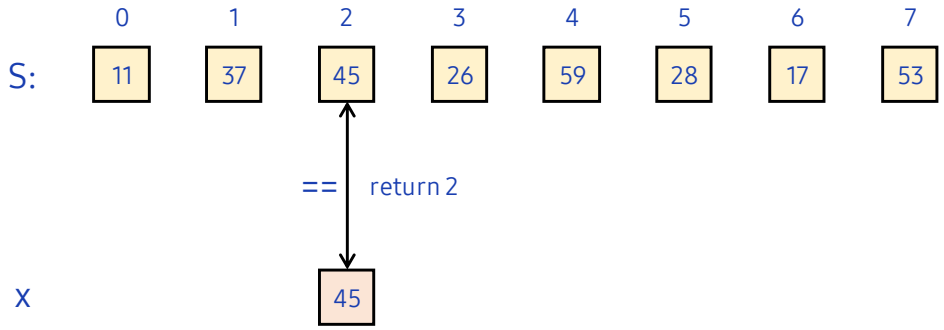
- If the value is not the same as the next element S[1], move to the next one.



2. Sequential Search

2.3. An algorithm for the sequential search

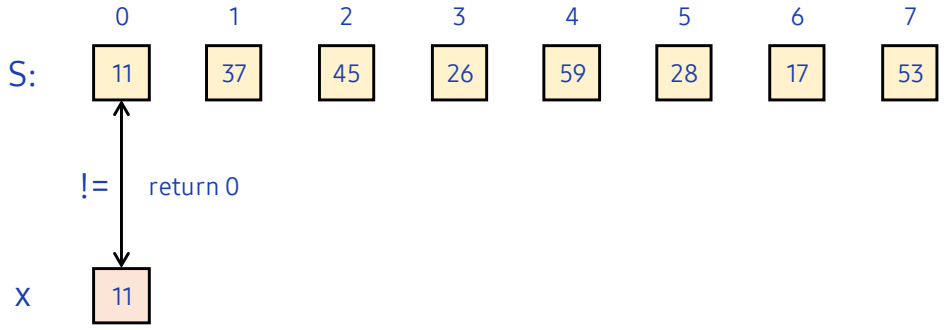
The x value is the same as the element S[2], so it returns 2 and finishes,



2. Sequential Search

2.3. An algorithm for the sequential search

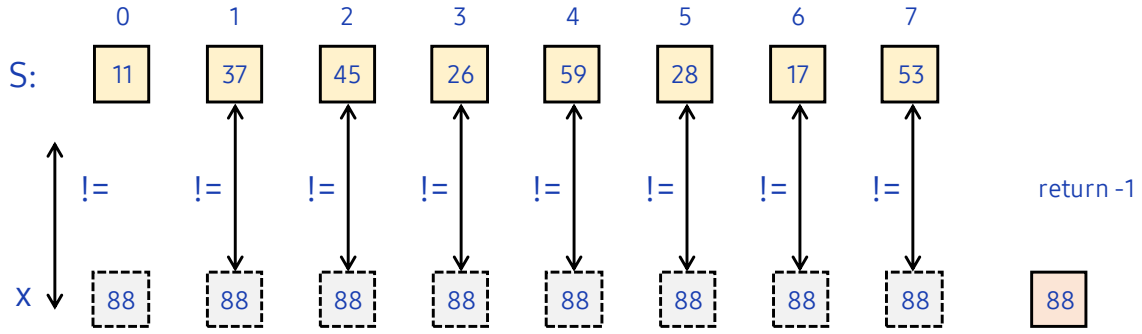
In the sequential search, if the element to find is S[0] which is the first one, only one comparison operation can be done to finish search, which is referred to 'base-case.'



2. Sequential Search

2.3. An algorithm for the sequential search

- In sequential search, if the element to find does not exist in S, comparison must be done to all elements of S.
- If there are n elements in S, a total of n comparison operation should be done. If so, it is called 'worst-case.'



2. Sequential Search

2.4. Implement the sequential search

- Sequential search algorithm can be implemented as follows.
- Receive 'nums' and x as input to proceed with sequential searching of the elements in nums.

```
1  def seq_search(nums, x):
2      for i in range(len(nums)):
3          if x == nums[i]:
4              return i
5      return -1
```

Line 2-5

- Line 2 is the for-loop for sequential searching of elements in nums.
- Line 3 and 4 return 'i' if the nums[i] and x value are the same.
- Line 5 returns -1 because nums does not have x.

2. Sequential Search

2.4. Implement the sequential search

- The seq_search() function that was implemented previously can be applied to find list elements as follows.

```
S = [11, 37, 45, 26, 59, 28, 17, 53]
x = 11
pos = seq_search(S, x)
print(f"In S, {x} is at position {pos}.")
```

```
In S, 11 is at position 0.
```

```
S = [11, 37, 45, 26, 59, 28, 17, 53]
x = 77
pos = seq_search(S, x)
print(f"In S, {x} is at position {pos}.")
```

```
In S, 77 is at position -1.
```


One More Step

- What is the time complexity of sequential search? Analyze with Big-O notation.
- First, in the best-case, searching is finished after a single comparison, so it is certainly O(1). Also, comparison for n times is required in the worst-case, so it is clearly O(n).
- How is the average time complexity of sequential search analyzed? Assume that S has N elements. When searching every element of S one by one, calculate how many comparison operations (==) should be done.
- Assign 1 for the first element and 2 for the second element, and so on. For Nth element, N times of comparison operations should be done. Thus, the number of comparison operation required to search every element once is N(N+1)/2.

$$1 + 2 + \cdots + N = \frac{N(N+1)}{2}$$

- So, the average search time T(n) to all elements is O(n), which is linear search time.

$$T(N) = \frac{N(N+1)}{2} \div N = \frac{N+1}{2} \in O(n)$$

3. Finding the Largest

3.1. An algorithm for finding the largest key

Consider an algorithm that finds the largest key in the list of given integers.

Focus The following is the input and output of the algorithm problem to find the largest key.

- Problem: Where is the largest key in the S, a set of given integers?
- Input: S as the list of integers
- Output: Largest index among the elements in S

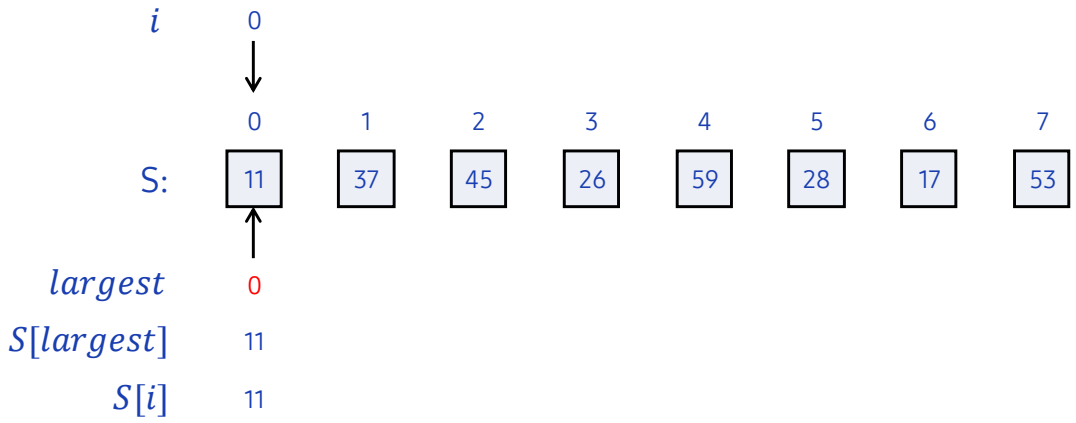
```
1 nums = [11, 37, 45, 26, 59, 28, 17, 53]
2 pos = find_largest(nums)
3 print(f"The largest is {nums[pos]} at {pos}")
```

```
[11, 37, 45, 26, 59, 28, 17, 53]
The largest is 59 at 4
```


3. Finding the Largest

3.1. An algorithm for finding the largest key

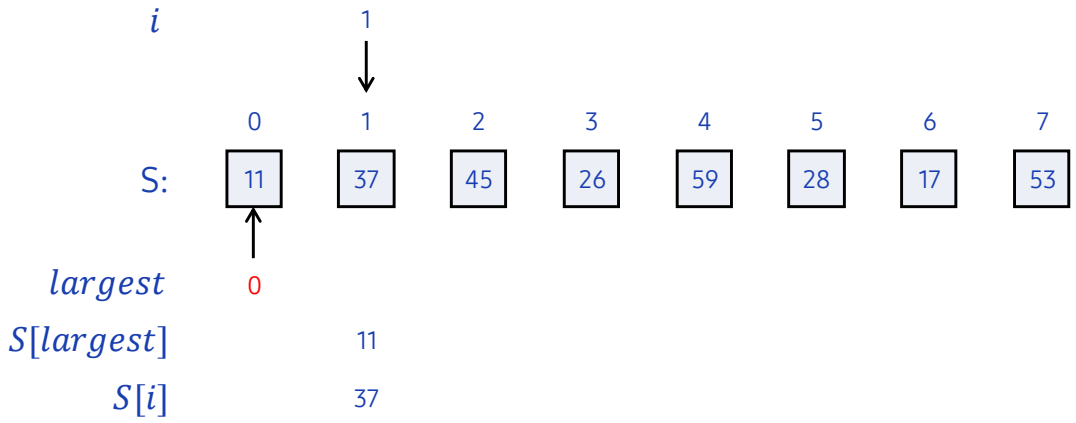
The largest key finding algorithm can be implemented in a similar way as the sequential search algorithm, First, assume that the location of the largest key is the first element of S.



3. Finding the Largest

3.1. An algorithm for finding the largest key

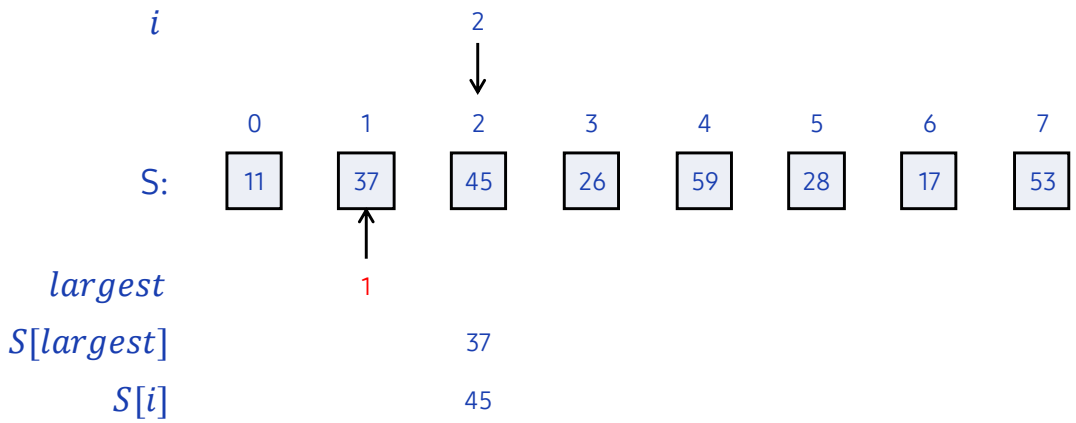
When comparing S[1] with the current largest S[largest], because S[1] is larger than S[0], the location of the largest key should be changed to 1.



3. Finding the Largest

3.1. An algorithm for finding the largest key

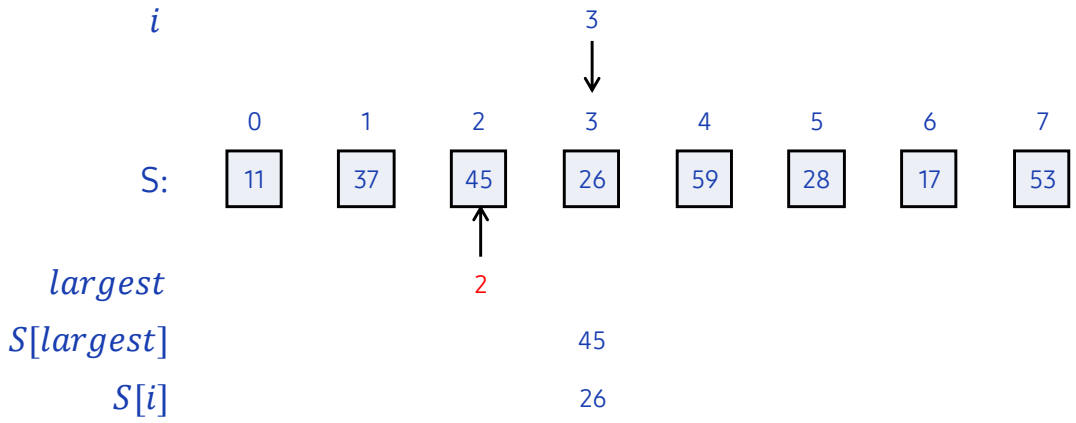
When comparing S[2] with the current largest S[largest], because S[2] is larger than S[1], the location of the largest key should be changed to 2.



3. Finding the Largest

3.1. An algorithm for finding the largest key

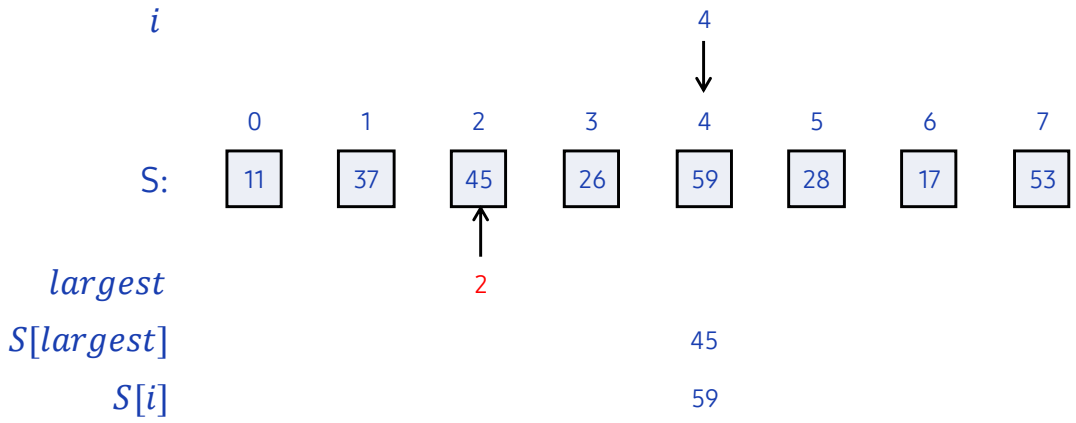
When comparing S[3] with the current largest S[largest], because S[3] is smaller than S[2], the location of the largest key does not change.



3. Finding the Largest

3.1. An algorithm for finding the largest key

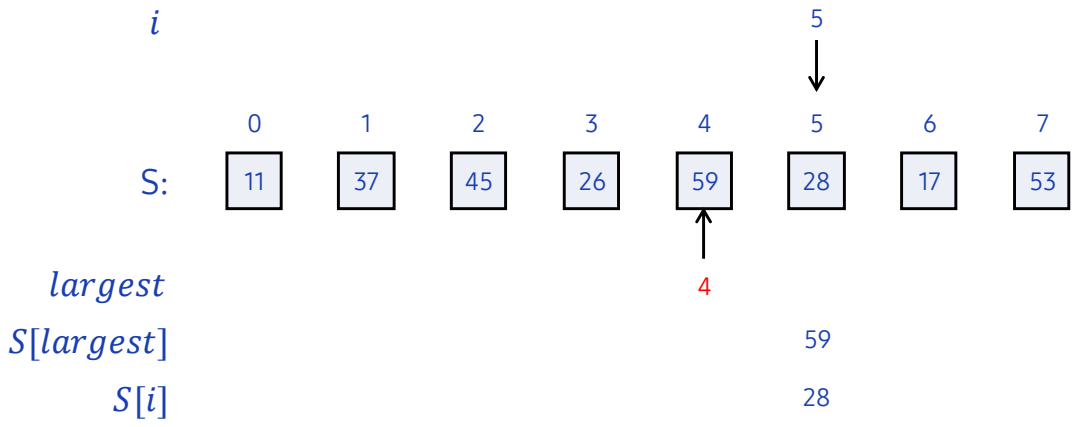
When comparing S[4] with the current largest S[largest], because S[4] is larger than S[2], the location of the largest key should be changed to 4.



3. Finding the Largest

3.1. An algorithm for finding the largest key

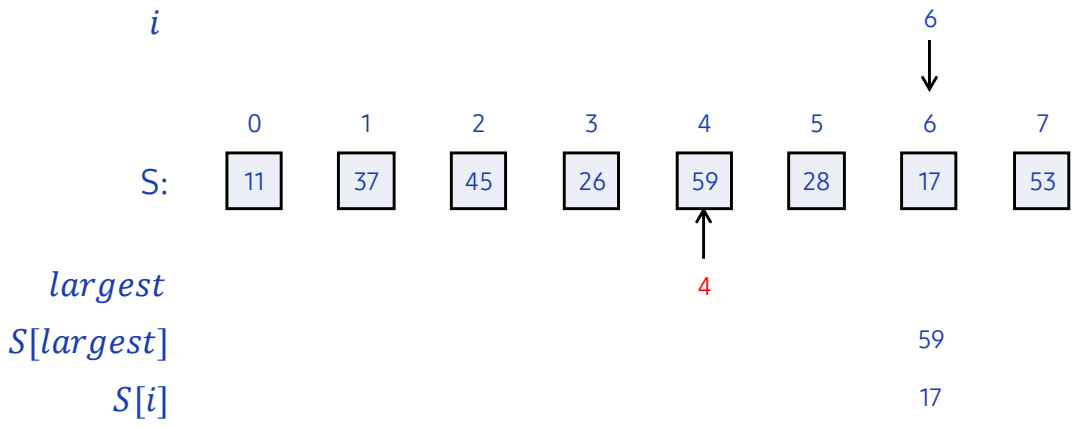
If there is no value that is larger than S[largest], the location of the largest key remains the same.



3. Finding the Largest

3.1. An algorithm for finding the largest key

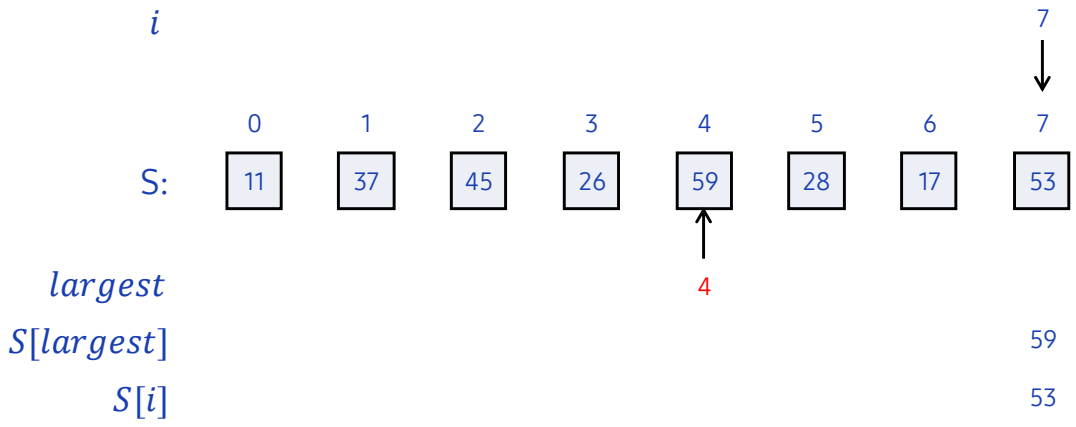
If there is no value that is larger than S[largest], the location of the largest key remains the same.



3. Finding the Largest

3.1. An algorithm for finding the largest key

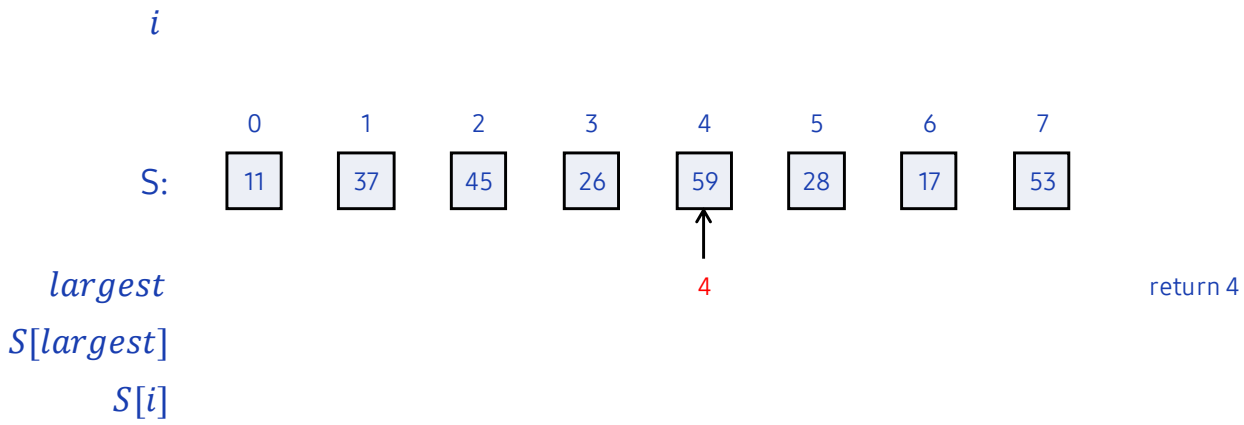
If there is no value that is larger than S[largest], the location of the largest key remains the same.



3. Finding the Largest

3.1. An algorithm for finding the largest key

The largest value has max. index after finish searching all elements, so it returns largest.



3. Finding the Largest

3.2. Implementation and coding

Apply the previously used strategy to implement the find_largest() function as follows.

```
1 def find_largest(nums):
2     largest = 0
3     for i in range(1, len(nums)):
4         if nums[largest] < nums[i]:
5             largest = i
6     return largest
```

Line 2-5

- Initialize the index of the first element in the nums list into the largest value.
- Compare the current nums[largest] value and nums[i] value from the 2nd element to the last element of the nums list.
- If the 'i' value is greater than the largest value, update the largest into 'i.'

3. Finding the Largest

3.2. Implementation and coding

```
1 def find_largest(nums):
2     largest = 0
3     for i in range(1, len(nums)):
4         if nums[largest] < nums[i]:
5             largest = i
6     return largest
```

Line 6

- The largest value after finishing searching is the index value of the largest element in the nums.

Let's code

Let's code

UNIT 24

1. Egg Dropping Problem

1.1. Problem Definition

- Write an algorithm to find the highest floor that does not break the egg when dropping it if the building height is given.
- The algorithm implementation conditions are as follows.

1. The 'breaking,' which is the lowest floor where the egg breaks, is randomly selected between 1 and height.
2. The egg dropping experiment is expressed as do_experiment(floor).
   - Return True if the egg breaks when dropping from the floor.
   - If not, return False.
3. However, the program must be closed when the do_experiment() function calls True.

1. Egg Dropping Problem

1.2. Implementation and Coding

Receive the user input regarding the building 'height' through the input() function.

```
1 from random import randint
2
3 height = int(input("Input the number of floors: "))
4 breaking = randint(1, height)
5 floor = find_highest_safe_floor(height, breaking)
6 print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Your egg will safe till the 94-th floor.
```

Line 3

- Convert the input type received by input() function into int() function and allocate it to variable 'height.'

1. Egg Dropping Problem

1.2. Implementation and Coding

Create a random number between 1 and height and allocate it to the variable 'breaking.'

```
1 from random import randint
2
3 height = int(input("Input the number of floors: "))
4 breaking = randint(1, height)
5 floor = find_highest_safe_floor(height, breaking)
6 print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Your egg will safe till the 94-th floor.
```

Line 1, 4

- For random creation of integers in a range, use the randint() function in the random module.
- The randint(a, b) function returns a random integer in the range from a and b (same or greater than 'a' and same or smaller than 'b').

1. Egg Dropping Problem

1.2. Implementation and Coding

Implement a function that returns the highest floor where the egg does not break.

```
1 from random import randint
2
3 height = int(input("Input the number of floors: "))
4 breaking = randint(1, height)
5 floor = find_highest_safe_floor(height, breaking)
6 print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Your egg will safe till the 94-th floor.
```

Line 5-6

- The find_highest_safe_floor() function returns the highest floor where the egg does not break.
- Print the string format by using f-string. The f-string adds 'f' in front of the string, and applies {} on the converted variable within the string.

1. Egg Dropping Problem

1.2. Implementation and Coding

Implement the do_experiment() function for egg dropping experiment.

```
1 def do_experiment(floor, breaking):
2     return floor >= breaking
3
4 def find_highest_safe_floor(height, breaking):
5     for n in range(1, height + 1):
6         if do_experiment(n, breaking):
7             return n - 1
8     return height
```

Line 1-2

- The 'breaking' variable is the lowest floor where the egg breaks.
- Thus, if the floor value as input parameter is same or greater than the breaking value, it returns True, and if not, it returns False.

1. Egg Dropping Problem

1.2. Implementation and Coding

- Find the highest floor where the egg does not break in the egg dropping experiment by using 'height' and 'breaking' information.

```
1  def do_experiment(floor, breaking):
2      return floor >= breaking
3
4  def find_highest_safe_floor(height, breaking):
5      for n in range(1, height + 1):
6          if do_experiment(n, breaking):
7              return n - 1
8      return height
```

Line 4-8

- Repeat the egg dropping experiment from 1st to 'height' floor, and if the egg breaks, then return the n-1 floor.
- If experiment is successful at all floors, then return the highest floor 'height.'

One More Step

- What is the time complexity of the egg dropping problem?
- If there is only one egg, similar to the sequential search, the best-case is O(1) while the worst-case is O(n). The average time complexity will be also O(n).
- What if there are infinite number of eggs so that breaking an egg during the experiment doesn't matter?
- When there are infinite number of eggs, it is possible to use the binary search which will be discussed later. The average time complexity will be O(logn).
- How many experiments should be done at least if there are two eggs?
- This problem is called two-eggs problems, and at least 14 experiments can be done on a 100th floor building as the worst-case.

Pop quiz

Pop quiz

UNIT 24

Quiz. #1

Guess the output of the following algorithm. Analyze the code and describe the behavior of this algorithm.

```python
def find_two(nums):
    x = y = 0
    for i in range(1, len(nums)):
        if nums[x] < nums[i]:
            x = i
        elif nums[y] > nums[i]:
            y = i
    return x, y
```

```python
nums = [11, 37, 45, 26, 59, 28, 17, 53]
i, j = find_two(nums)
print(nums[i], nums[j])
```

Quiz. #2

How many comparisons should be done for the find_two() that was implemented in quiz #1?

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair programming UNIT 24

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Write a program to count how many times a specific word was used in the sentence entered by the user.

- In the sentence entered by the user, a word is distinguished by the presence of spacing.
- Use the input().split() function to create the list S that has each string as its element.
- Use the input() function to receive the word that will be searched by the user and save it to x.
- The word_count() function receives S and x as parameters and provides a return by counting how many x are present in the S.

```
1 S = list(input("Input a sentence: ").split())
2 x = input("Input a word to search: ")
3 count = word_count(S, x)
4 print(f"In S, {x} is appeared in {count} times.")
```

```
Input a sentence: the quick brown fox jumps over the lazy dog
Input a word to search: the
In S, the is appeared in 2 times.
```


Unit 25.

Binary Search

Unit learning objective (1/3)

Learning objectives

- ✓ Understand the searching problems and be able to solve them through binary search.
- ✓ Be able to implement binary search algorithm with Python language.
- ✓ Be able to understand and explain the time complexity of binary search algorithm.

Unit learning objective (2/3)

Learning overview

- ✓ Understand searching problems that find a specified element from orderly arranged data.
- ✓ Learn how to solve searching problems of ordered data through binary search.
- ✓ Learn how to write binary search algorithms with Python, and analyze the best, worst, and average time complexity.

Concepts You Will Need to Know From Previous Units

- ✓ How to compare the size of list elements
- ✓ How to make recurrence relation and solve a problem with a recursive function
- ✓ How to express the time complexity of an algorithm by using Big-O notation

Keywords

Searching Problem

Binary Search

Logarithmic Time

Mission

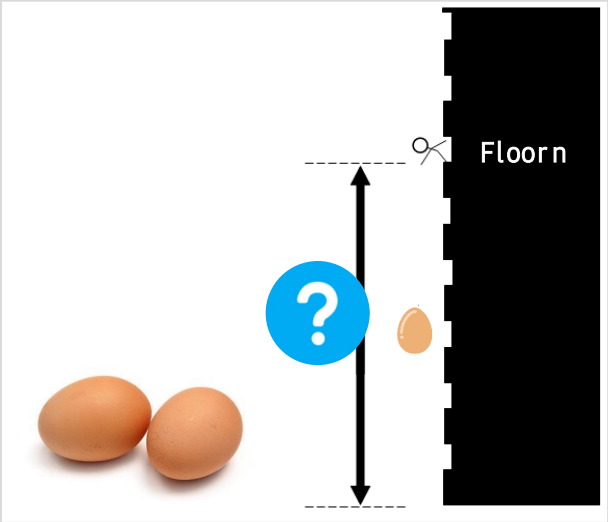
1. Real world problem

1.1. Egg dropping challenge revisited

- Let's participate in the egg dropping challenge again.
- In the last challenge, there was only one egg, so we tried to calculate the highest floor where the egg does not break for the first time.
- This time, there are sufficient number of eggs, meaning that you can keep doing experiments regardless of breaking eggs.
- If there are sufficient eggs, the egg dropping problem can be more effectively solved compared to the sequential search.
- Try to solve the egg dropping problem with binary search.

2. Mission

2.1. Egg dropping problem with infinite eggs

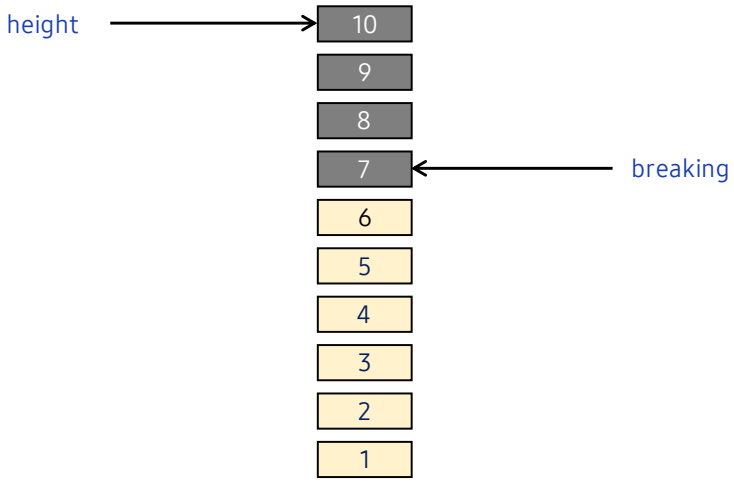


- If the building's highest floor and the lowest floor where the egg breaks are given, find the highest floor where the egg does not break.
- Since there are infinite eggs, assume that you can continue experimenting even if eggs break.
- If you can continue experimenting even if eggs break, if an egg breaks at the mid-floor of the building, then the solution will be downstairs. Likewise, if an egg does not break at the mid-floor of the building, then the solution will be upstairs.
- Apply the same method to halve the space for finding the solution.
- This kind of method is called binary search.

2. Mission

2.1. Egg dropping problem with infinite eggs

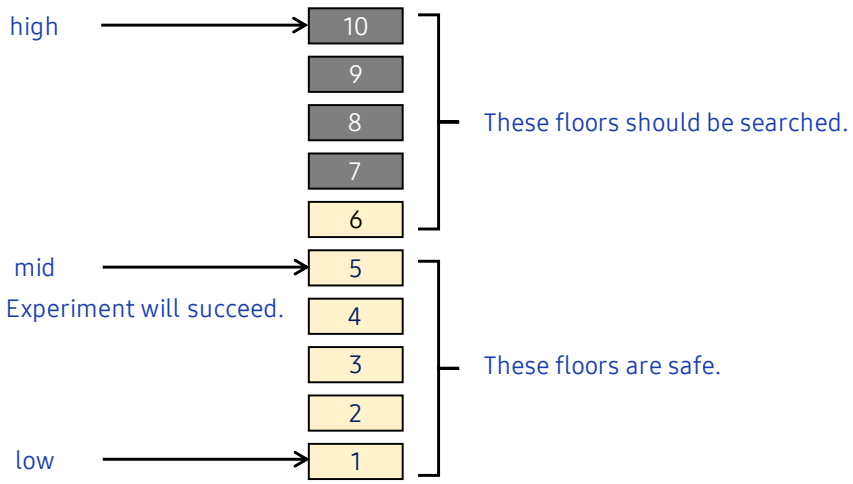
- Do an experiment at a building where N=10.
- Assume that dropping an egg at floors lower than the 6^{th} floor does not break an egg, and dropping it at floors higher than the 7^{th} floor breaks an egg.



2. Mission

2.1. Egg dropping problem with infinite eggs

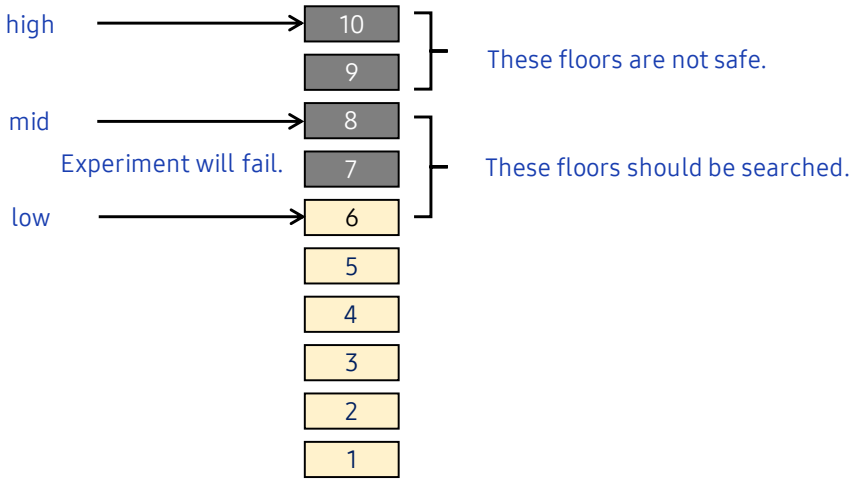
- First, let's do an experiment at 5^{th} floor that is the midpoint of the 1^{st} and 10^{th} floors.
- If the egg doesn't break, the floor where the egg breaks will be higher than the 5^{th} floor.



2. Mission

2.1. Egg dropping problem with infinite eggs

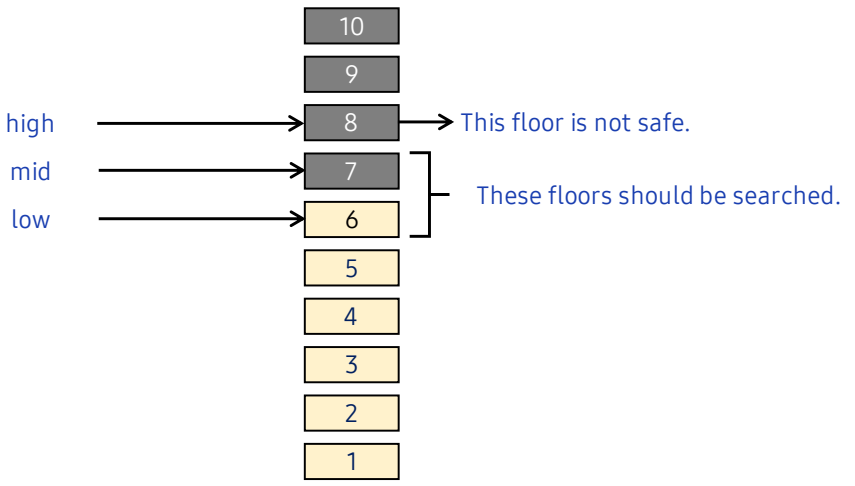
- Now, the section we will investigate is from 6th to 10th floors.
- Similarly, if an egg breaks at the 8th floor which is the midpoint, then the first floor where the egg breaks would be the 8th floor or downstairs.



2. Mission

2.1. Egg dropping problem with infinite eggs

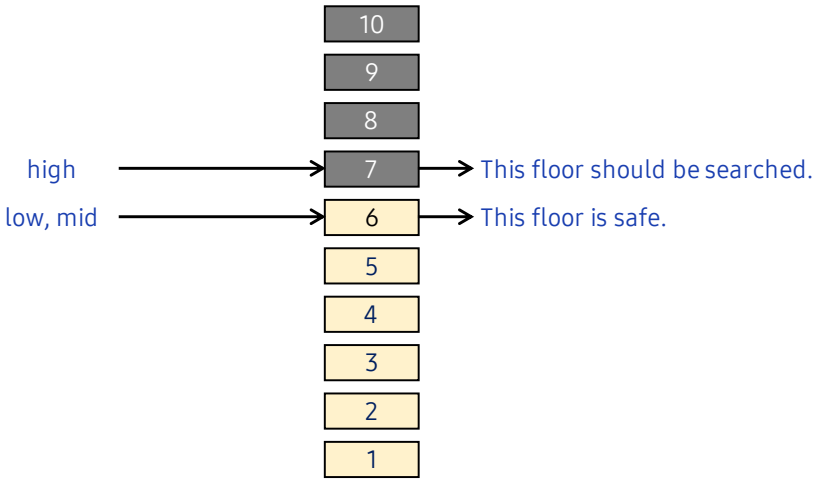
- The next section we will investigate is from 6th to 8th floors.
- Similarly, if an egg breaks at the 7th floor which is the midpoint, then the first floor where the egg breaks would be the 7th floor or downstairs.



2. Mission

2.1. Egg dropping problem with infinite eggs

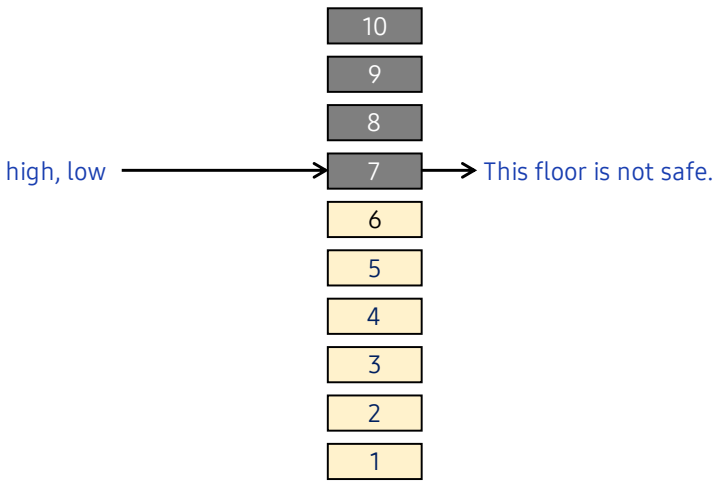
- We will now try 6th and 7th floors.
- Assume that the midpoint is 6th floor, and if an egg does not break from the 6th floor, then the first floor where the egg breaks will be higher than the 6th floor.



2. Mission

2.1. Egg dropping problem with infinite eggs

- The next section is 7th floor.
- Only one floor is remained to check, and since we've been searching for the lowest floor where an egg breaks, the 6th floor is considered as the highest floor where the egg does not break.



3. Solution

3.1. How the solution works

Receive the user input regarding the building height and the floor where the egg breaks to find the highest floor where the egg does not break.

```
height = int(input("Input the number of floors: "))
breaking = int(input("Input the first breaking floor: "))
floor = find_highest_safe_floor2(height, breaking)
print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 10
Input the first breaking floor: 7
Your egg will safe till the 6-th floor.
```


3. Solution

3.2. Egg dropping problem final code

```
def do_experiment(floor, breaking):
    return floor >= breaking

def find_highest_safe_floor2(height, breaking):
    low, high = 1, height
    while low < high:
        mid = (low + high) // 2
        if do_experiment(mid, breaking):
            high = mid
        else:
            low = mid + 1
    return low - 1
```

Key concept

Key concept

UNIT 25

1. Binary Search

1.1. An algorithm for the binary search

Consider the binary search algorithm that finds a specific integer from the list of ordered integers.

Focus The following is the binary search algorithm problem, and its input and output.

- Problem: Does a random integer x exist in the S, a set of ordered integers?
- Input: S as the list of integers arranged in ascending order, x as the random integer to find
- Output: The index location if x exists in the list S, and -1 if it does not exist.

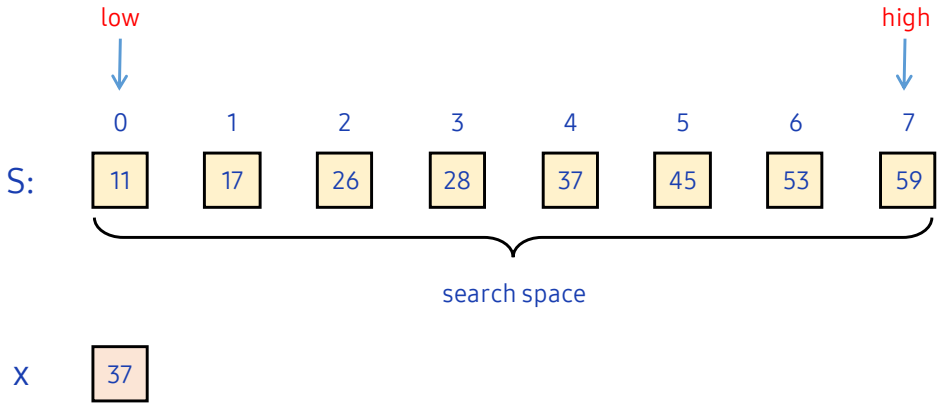
```
S = [11, 17, 26, 28, 37, 45, 53, 59]
x = int(input("Input the number to search: "))
pos = bin_search(S, x)
print(f"In S, {x} is at position {pos}.")
```

```
Input the number to search: 53
In S, 53 is at position 6.
```

1. Binary Search

1.1. An algorithm for the binary search

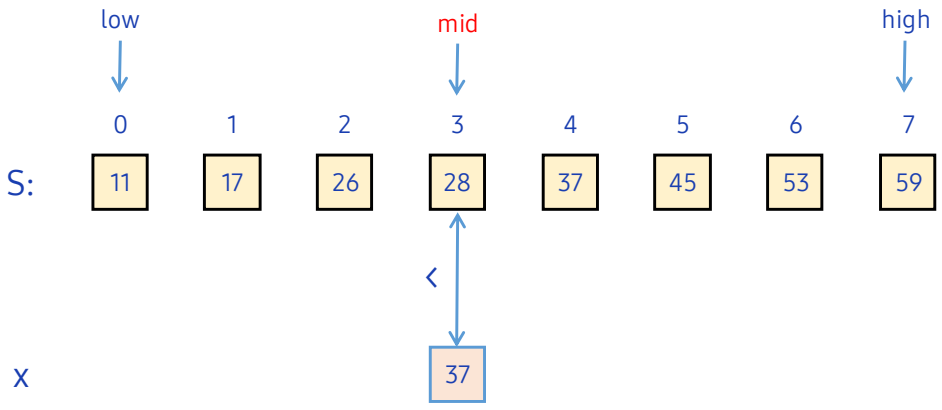
- The binary search algorithm searches x value in the given list S.
- The initial search space is between the low and high (low as the 1st element index and high as the last element index) in S.



1. Binary Search

1.1. An algorithm for the binary search

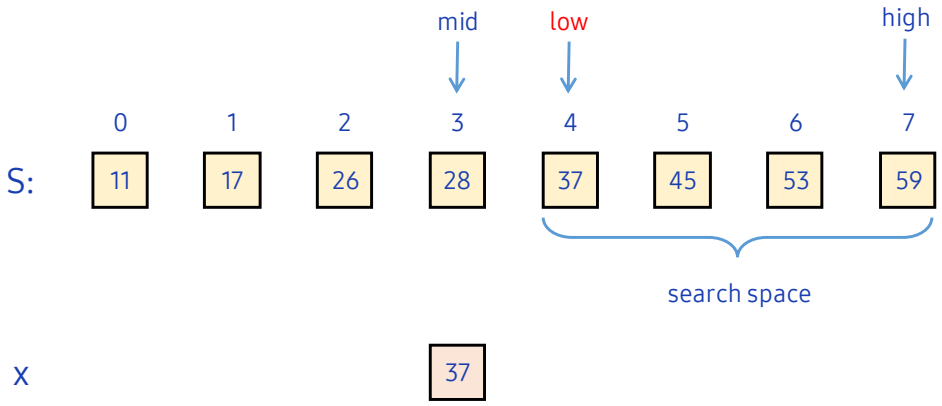
- In the binary search algorithm, the input list is arranged in ascending order.
- Compare the x value with mid value that is in the middle of low and high.



1. Binary Search

1.1. An An algorithm for the binary search

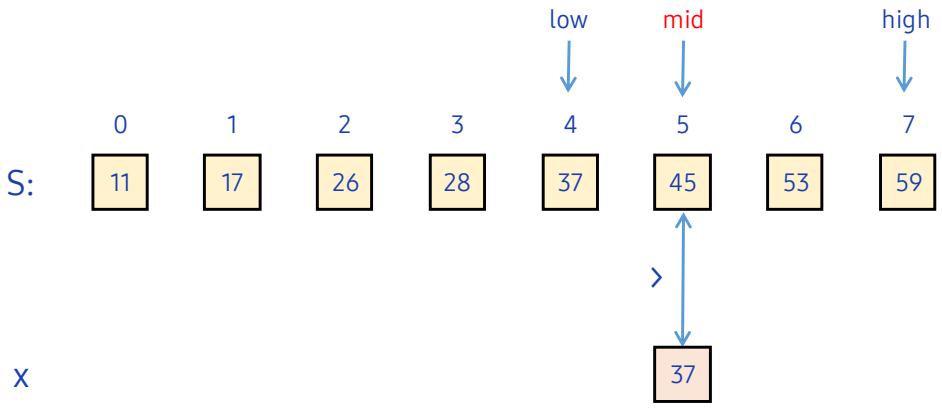
- If S[mid] is smaller than x, the x to find would be in between mid+1 and high.
- Thus, low should be changed to mid+1, which halves the search space from [0, 7] to [4, 7].



1. Binary Search

1.1. An algorithm for the binary search

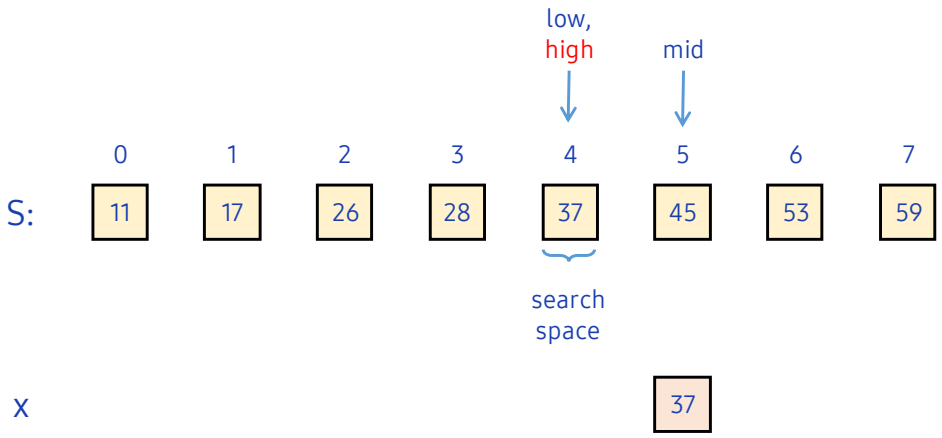
- Calculate the 'mid' which is in between the low and high to compare S[mid] and x.
- S[mid] is greater than x, so high should be changed to mid-1.



1. Binary Search

1.1. An algorithm for the binary search

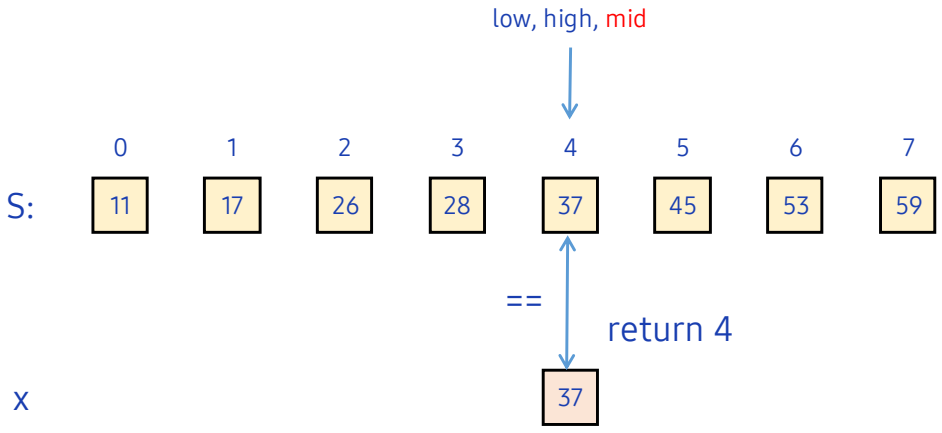
- The search space is reduced from [4, 7] to [4, 4].



1. Binary Search

1.1. An algorithm for the binary search

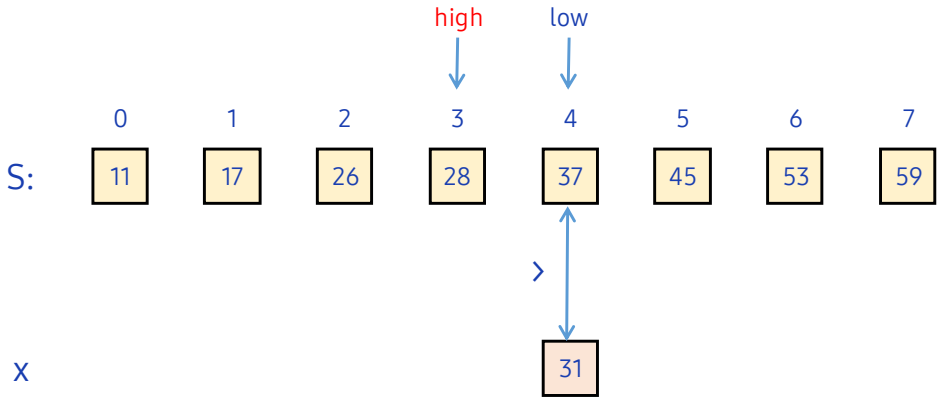
- Calculate the 'mid' which is in between the low and high to compare S[mid] and x.
- Since the S[mid] and x values are the same, return the mid value at current location.



1. Binary Search

1.1. An algorithm for the binary search

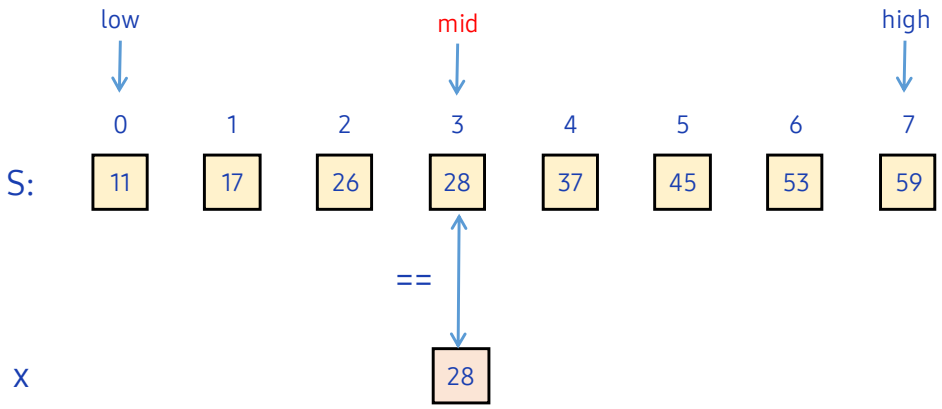
- If the value to find does not exist, for instance 31, because x is smaller than S[mid], high would change to mid-1.
- The low < high condition does not satisfy, so break the while statement and return -1.



1. Binary Search

1.2. An algorithm for the binary search

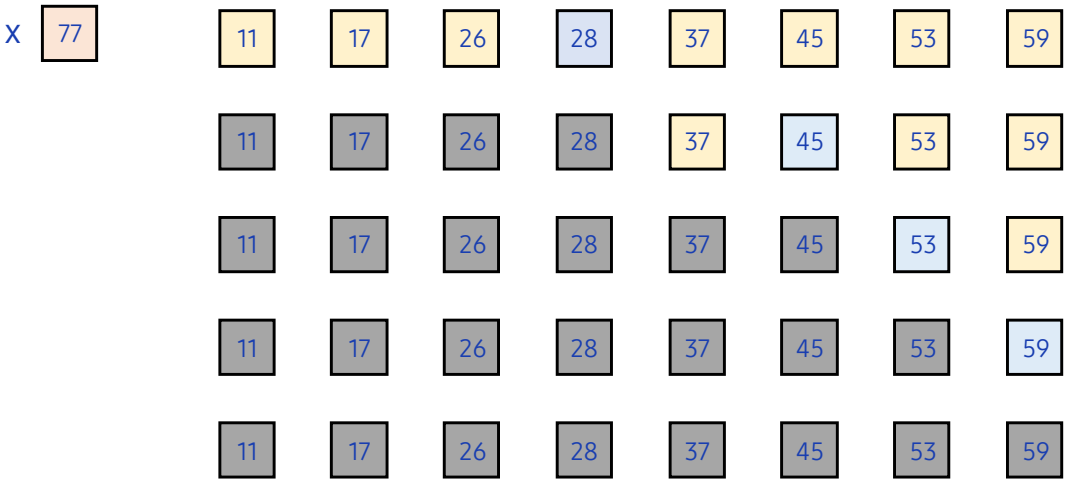
- What would be the best-case in binary search? When you're lucky to search the mid value in the S, only one comparison can be done to find the element.



1. Binary Search

1.2. An algorithm for the binary search

- What would be the worst-case in binary search? Similar to sequential search, highest number of comparison must be done to search for an element that does not exist in S.
- In the following example, a total of 4 comparison operations should be done to search for 77.



1. Binary Search

1.3. Implementation and coding

- The binary search algorithm can be implemented as follows. Receive 'nums' and x as input to set the entire nums as search space.

```
1  def bin_search(nums, x):
2      low, high = 0, len(nums) - 1
3      while low <= high:
4          mid = (low + high) // 2
5          if nums[mid] == x:
6              return mid
7          elif nums[mid] > x:
8              high = mid - 1
9          else:
10             low = mid + 1
11     return -1
```

Line 2

- The 'low' and 'high' are initialized as the first and last element of the nums, respectively.
- In the start part of the binary search, entire numbs is the search space. Thus, the search space is [0, len(nums)-1].

1. Binary Search

1.3. Implementation and coding

Binary search compares the mid value that is in between low and high. Since the data is ordered, a single comparison can halve the search space.

```
1  def bin_search(nums, x):
2      low, high = 0, len(nums) - 1
3      while low <= high:
4          mid = (low + high) // 2
5          if nums[mid] == x:
6              return mid
7          elif nums[mid] > x:
8              high = mid - 1
9          else:
10             low = mid + 1
11     return -1
```

Line 3-10

- The mid value is calculated as (low + high) // 2. Here, the // operator returns the quotient of division.
- If the nums[mid] value is the same as x, the search is successful so return the mid.
- If the nums[mid] value is larger than x, change 'high' to mid-1, and if it is smaller than x, change 'low' to mid+1.
- If the low or high value changes to mid + 1 or mid – 1, then the search space halves.

1. Binary Search

1.3. Implementation and coding

Return -1 if x is not found in the binary search process.

```
1  def bin_search(nums, x):
2      low, high = 0, len(nums) - 1
3      while low <= high:
4          mid = (low + high) // 2
5          if nums[mid] == x:
6              return mid
7          elif nums[mid] > x:
8              high = mid - 1
9          else:
10             low = mid + 1
11     return -1
```

Line 11

- If the low value is larger than the high value, then break the while statement.
- If so, return -1 because x value does not exist in S.

One More Step

- What is the time complexity of binary search? First, the best-case can finish searching by only one time, so it is O(1). Then, how would you analyze the worst-case time complexity of binary search?
- Assume that S has N elements. The size of the first search space is N. The second space to search is half of the original one, so it is N/2. The next search space is halved again, which is N/4. This explanation can be generalized as follows.

$$N, \frac{N}{2}, \frac{N}{2^2}, \cdots, \frac{N}{2^k}$$

- Binary search is finished if there is only one search space, which is $N/2^k=1$.

$$N = 2^k$$
$$\log_2 N = k$$

- The max. number of binary search operation is $\log_2 N$, so the worst-case time complexity of binary search is $O(\log_2 N)$.

1. Binary Search

1.4. Performance evaluation

- Add the print() function to bin_search() to check time complexity of binary search.

```
1  def bin_search(nums, x):
2      low, high = 0, len(nums) - 1
3      while low <= high:
4          mid = (low + high) // 2
5          print(low, high, mid)
6          if nums[mid] == x:
7              return mid
8          elif nums[mid] > x:
9              high = mid - 1
10         else:
11             low = mid + 1
12     return -1
```

Line 5

- Print the low, high, and mid in the loop statement to count the number of binary search repetition.

1. Binary Search

1.4. Performance evaluation

In the list with 8 elements, the search is finished after one-time operation in the best-case, while four times of operations were done in the worst-case.

```
1 S = [11, 17, 26, 28, 37, 45, 53, 59]
2 x = int(input("Input the number to search: "))
3 pos = bin_search(S, x)
4 print(f"In S, {x} is at position {pos}.")
```

```
Input the number to search: 28
0 7 3
In S, 28 is at position 3.
```

```
1 S = [11, 17, 26, 28, 37, 45, 53, 59]
2 x = int(input("Input the number to search: "))
3 pos = bin_search(S, x)
4 print(f"In S, {x} is at position {pos}.")
```

```
Input the number to search: 77
0 7 3
4 7 5
6 7 6
7 7 7
In S, 77 is at position -1.
```


Let's code

1. Egg Dropping Problem

1.1. Problem definition

- Write an algorithm to find the highest floor that does not break the egg when dropping it if the building height is given.
- The algorithm implementation conditions are as follows.

1. The 'breaking,' which is the lowest floor where the egg breaks, is randomly selected between 1 and height.
2. The egg dropping experiment is expressed as do_experiment(floor).
   - Return True if the egg breaks when dropping from the floor.
   - If not, return False.
3. The do_experiment() function can be called without limit because there are infinite number of eggs.

1. Egg Dropping Problem

1.2. Implementation and coding

- Receive user input regarding the building 'height' and the 'breaking' which is the lowest floor of breaking an egg.

```
1 height = int(input("Input the number of floors: "))
2 breaking = int(input("Input the first breaking floor: "))
3 floor = find_highest_safe_floor2(height, breaking)
4 print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Input the first breaking floor: 93
Your egg will safe till the 92-th floor.
```

Line 1-2

- Convert the height and breaking values entered by the user as input() function into int() function and save.

1. Egg Dropping Problem

1.2. Implementation and coding

Call the function that returns the highest floor that does not break an egg.

```
1 height = int(input("Input the number of floors: "))
2 breaking = int(input("Input the first breaking floor: "))
3 floor = find_highest_safe_floor2(height, breaking)
4 print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Input the first breaking floor: 93
Your egg will safe till the 92-th floor.
```

Line 3-4

- The find_highest_safe_floor2() function finds the highest floor where an egg does not break through binary search.
- Use f-string to print the floor variable, which should be -1 than the breaking value.

1. Egg Dropping Problem

1.2. Implementation and coding

The do_experiment() function is the same for the egg dropping experiment.

```
1  def do_experiment(floor, breaking):
2      return floor >= breaking
3
4  def find_highest_safe_floor2(height, breaking):
5      low, high = 1, height
6      while low < high:
7          mid = (low + high) // 2
8          if do_experiment(mid, breaking):
9              high = mid
10         else:
11             low = mid + 1
12     return low - 1
```

Line 1-2

- The 'breaking' is the lowest floor where an egg breaks.
- If the floor value is the same or larger than breaking, return True, and if not, return False.

1. Egg Dropping Problem

1.2. Implementation and coding

The find_highest_safe_floor2() function finds the highest floor where an egg does not break through binary search.

```
1  def do_experiment(floor, breaking):
2      return floor >= breaking
3
4  def find_highest_safe_floor2(height, breaking):
5      low, high = 1, height
6      while low < high:
7          mid = (low + high) // 2
8          if do_experiment(mid, breaking):
9              high = mid
10         else:
11             low = mid + 1
12     return low - 1
```

Line 4-5

- Initialize the initial search space for binary search into low and high.
- Initialize 'low' into 1st floor and 'high' into the 'height' which is the highest floor of the building.

1. Egg Dropping Problem

1.2. Implementation and coding

Find the 1st floor where the egg breaks by reducing the search space into half through binary search.

```
1  def do_experiment(floor, breaking):
2      return floor >= breaking
3
4  def find_highest_safe_floor2(height, breaking):
5      low, high = 1, height
6      while low < high:
7          mid = (low + high) // 2
8          if do_experiment(mid, breaking):
9              high = mid
10         else:
11             low = mid + 1
12     return low - 1
```

Line 6-11

- Set mid as the in-between floor of low and high, and then perform the experiment.
- If the egg does not break, then change high to mid to reduce the search space into lower floors.
- If the egg breaks, then change low to mid+1 to reduce the search space into upper floors.
- Finish searching if the low value becomes the same or larger than the high value.

1. Egg Dropping Problem

1.2. Implementation and coding

After searching is done, return the highest floor where the egg breaks.

```
1  def do_experiment(floor, breaking):
2      return floor >= breaking
3
4  def find_highest_safe_floor2(height, breaking):
5      low, high = 1, height
6      while low < high:
7          mid = (low + high) // 2
8          if do_experiment(mid, breaking):
9              high = mid
10         else:
11             low = mid + 1
12     return low - 1
```

Line 12

- From the binary search, the result is the lowest floor where the egg does not break.
- So, the highest floor where the egg breaks returns the low-1 value after searching is done.

1. Egg Dropping Problem

1.3. Performance evaluation

Add the print() function to check how many binary search operations were done.

```
1  def do_experiment(floor, breaking):
2      return floor >= breaking
3
4  def find_highest_safe_floor2(height, breaking):
5      low, high = 1, height
6      while low < high:
7          mid = (low + high) // 2
8          print(low, high, mid)
9          if do_experiment(mid, breaking):
10             high = mid
11         else:
12             low = mid + 1
13     return low - 1
```

Line 8

- Print the low, high, and mid in the loop statement to count the number of binary search repetition.

1. Egg Dropping Problem

1.3. Performance evaluation

If the 50th floor is the first floor where the egg breaks, then binary search is operated for 6 times.

```
height = int(input("Input the number of floors: "))
breaking = int(input("Input the first breaking floor: "))
floor = find_highest_safe_floor2(height, breaking)
print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Input the first breaking floor: 50
1 100 50
1 50 25
26 50 38
39 50 44
45 50 47
48 50 49
Your egg will safe till the 49-th floor.
```


1. Egg Dropping Problem

1.3. Performance evaluation

If the 51st floor is the first floor where the egg breaks, then binary search is operated for 7 times.

```
height = int(input("Input the number of floors: "))
breaking = int(input("Input the first breaking floor: "))
floor = find_highest_safe_floor2(height, breaking)
print(f"Your egg will safe till the {floor}-th floor.")
```

```
Input the number of floors: 100
Input the first breaking floor: 51
1 100 50
51 100 75
51 75 63
51 63 57
51 57 54
51 54 52
51 52 51
Your egg will safe till the 50-th floor.
```

Pop quiz

Pop quiz

UNIT 25

Quiz. #1

The following is the code for 'guessing number' game. If maximum=100 and number = 51, how many counts would be printed?

```
from random import randint

maximum = int(input("Enter the number of maximum: "))
number = int(input("Enter your guessing number: "))
count = 0
low, high = 1, maximum
while low < high:
    mid = (low + high) // 2
    count += 1
    if mid == number:
        print(f"Your number is {number}.")
        break
    elif mid > number:
        high = mid - 1
    else:
        low = mid + 1
print(f"Total {count} times are searched.")
```

Quiz. #2

If maximum=100 and number = 25 in the 'guessing number' game, then how many counts would be printed?

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair programming UNIT 25

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. If an ordered list 'nums' and a random integer x are given, implement the function that returns the location of index where x will be inserted. However, the S should be an ordered list even after inserting x.

```
1 nums = [10, 20, 40, 50, 60, 80]
2 x = int(input("Input a number to insert: "))
3 pos = search_insert_position(nums, x)
4 print(f"{x} should be inserted at position {pos}.")
5 nums.insert(pos, x)
6 print(nums)
```

```
Input a number to insert: 30
30 should be inserted at position 2.
[10, 20, 30, 40, 50, 60, 80]
```


Unit 26.

Hash Table

Unit learning objective (1/3)

Learning objectives

- ✓ Understand hashing problems and be able to solve search problems by using a hash table
- ✓ Be able to use dictionary data structures in Python language
- ✓ Be able to implement a hash table in Python language and apply it for problem solving

Unit learning objective (2/3)

Learning overview

- ✓ Learn how to save the key-value pair and do searching by using dictionary data structures
- ✓ Learn how to implement a hash table into Python class
- ✓ Learn how to search data from constant time by using a hash table

Concepts You Will Need to Know From Previous Units

- ✓ How to save and search for data by using key values through dictionary
- ✓ How to define variables and methods through the class and make approach
- ✓ How to express the time complexity of the algorithm by using Big-O notation

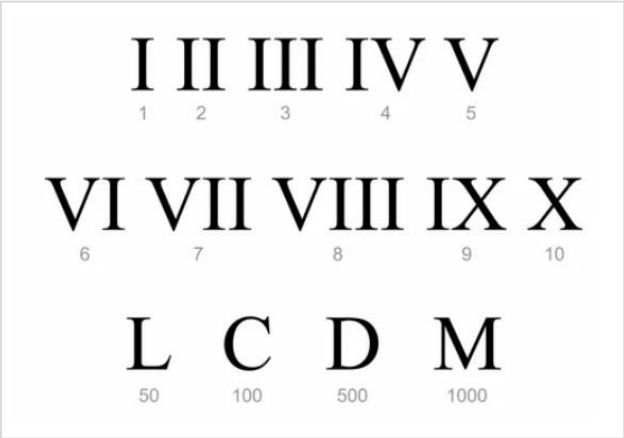
Keywords

- Hash Table
- Hashing
- Hash Function
- Dictionary
- Constant Time

Mission

1. Real world problem

1.1. Roman numbers



- In the past, roman numbers were used in ancient Rome.
- Create a roman numerals convertor that converts Roman numerals into Arabic numbers.

2. Mission

2.1. Roman numerals converter

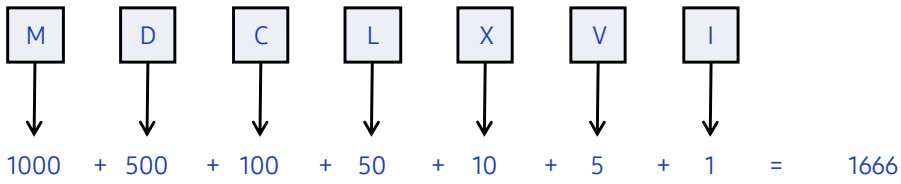
Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

- Roman numerals express numbers with 7 different symbols.
- The table on the left shows the value that each symbol represents.
- Roman numerals are easily calculated if you add the symbols in order.
- For example, III is adding 1 for three times, and XVI is the addition of 10, 5, and 1.
- However, if a symbol that goes behind is a larger number, subtract the value of the symbol in front.
- For example, IX is addition of 10 and subtracting 1, while IV is addition of 5 and subtracting 1.

2. Mission

2.1. Roman numerals converter

- How would you do to convert the Roman numerals MDCLXVI?
- As shown below, convert each symbol into Arabic numeral and then add them.



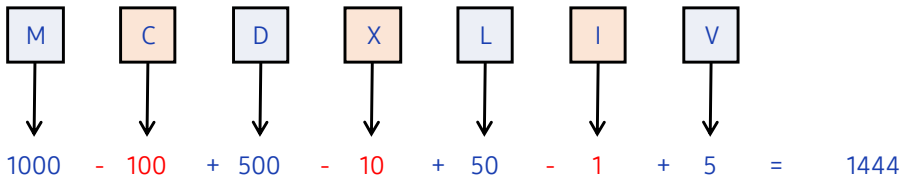

```
1 roman = input("Input a Roman number: ")
2 number = roman_to_int(roman)
3 print(number)
```

```
Input a Roman number: MDCLXVI
1666
```


2. Mission

2.1. Roman numerals converter

- How would you do to convert the Roman numerals MCDXLIV?
- In this case, because C is lesser than D, you need to subtract C. Likewise, subtract X and I from XL and IV.



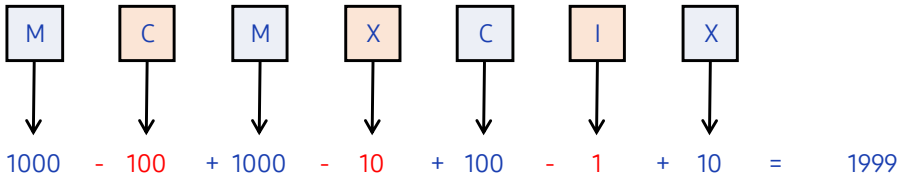

```
1 roman = input("Input a Roman number: ")
2 number = roman_to_int(roman)
3 print(number)
```

```
Input a Roman number: MCDXLIV
1444
```

2. Mission

2.1. Roman numerals converter

Also, for MCMSCIX, because the values of C, X, I are smaller than the values of M, C, X that follow behind, they need to be subtracted.




```
roman = input("Input a Roman number: ")
number = roman_to_int(roman)
print(number)
```

```
Input a Roman number: MCMXCIX
1999
```


3. Solution

3.1. How the Roman numerals converter works

As shown below, the Roman numerals converter receives Roman numeral input and prints in into Arabic numeral.

```
table = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
roman = input("Input a Roman number: ")
number = roman_to_int(roman)
print(number)
```

```
Input a Roman number: MCDXLIV
1444
```

3. Solution

3.2. Roman numerals converter final code

```python
def roman_to_int(str):
    result = 0
    for i in range(len(str) - 1):
        if table[str[i]] < table[str[i + 1]]:
            result -= table[str[i]]
        else:
            result += table[str[i]]
    return result + table[str[-1]]
```


Key concept

1. Hash Table

1.1. Definition of hash table

- Hash table, or hash map refers to an abstract data type that stores and extracts data by mapping keys to values through hashing.
- Hashing refers to designating the locations to save data or find saved data by data values. Hash function determines the key by using data value.

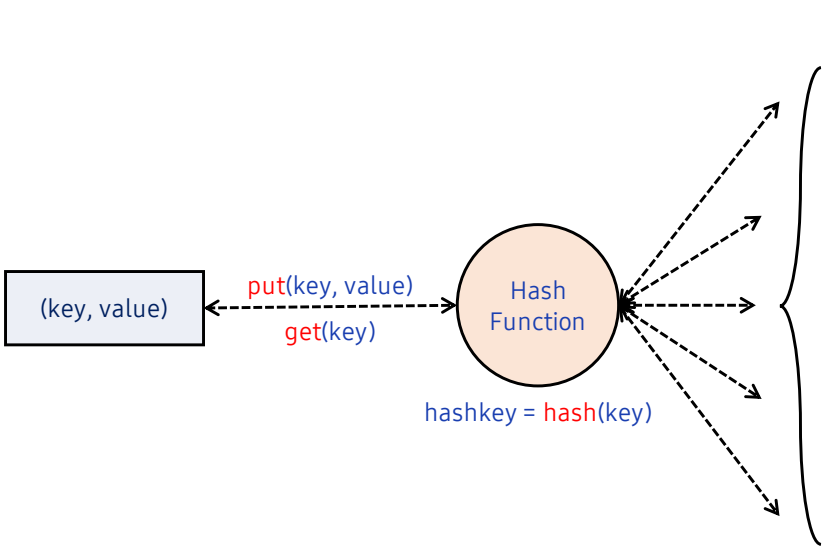
Focus Hash table should provide three significant interfaces as follows.

- put(key, value): Stores the value to the bucket designated by the key hash value
- get(key): Extracts the value stored in the bucket designated by the key hash value
- hash(key): Converts data key value into a hash value in specific range

1. Hash Table

1.1. Definition of hash table

- The following shows structure of hash table.



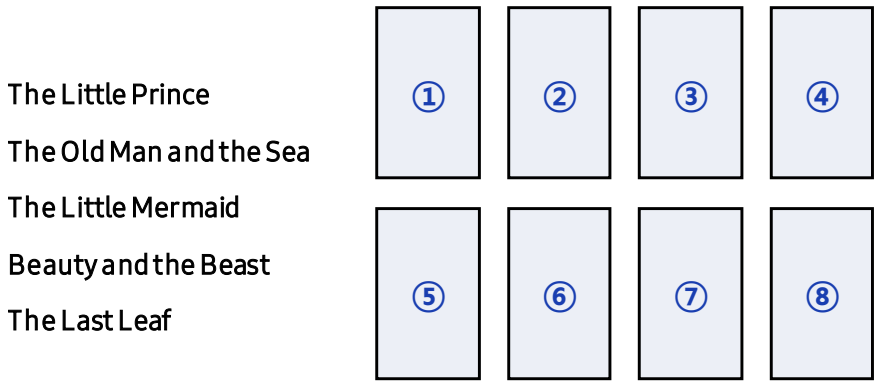
Hashkey	Value
0	
1	
2	
3	(key, value)
4	
5	
6	
7	

Hash Table

1. Hash Table

1.2. An example of the hash table

For example, imagine that you put 5 different books in the bookshelf.



Bookshelf with 8 slots

1. Hash Table

1.2. An example of the hash table

Assign random key values to each book.

Here, use the sum of ASCII Codes of the book title as the key value of the book.

Title	T	h	e		L	a	s	t		L	e	a	f		
ASCII Codes	84	+ 104	+ 101	+ 32	+ 76	+ 97	+ 115	+ 116	+ 32	+ 76	+ 101	+ 97	+ 102	=	1,113

```
1 list(map(ord, "The Last Leaf"))
```

```
[84, 104, 101, 32, 76, 97, 115, 116, 32, 76, 101, 97, 102]
```

1. Hash Table

1.2. An example of the hash table

The key value for each book can be calculated with the following code.

```
1  books = [
2      "The Little Prince",
3      "The Old Man and the Sea",
4      "The Little Mermaid",
5      "Beauty and the Beast",
6      "The Last Leaf"
7  ]
8  for book in books:
9      key = sum(map(ord, book))
10     print(key, book)
```

```
1584 The Little Prince
1929 The Old Man and the Sea
1678 The Little Mermaid
1837 Beauty and the Beast
1133 The Last Leaf
```

Line 8-10

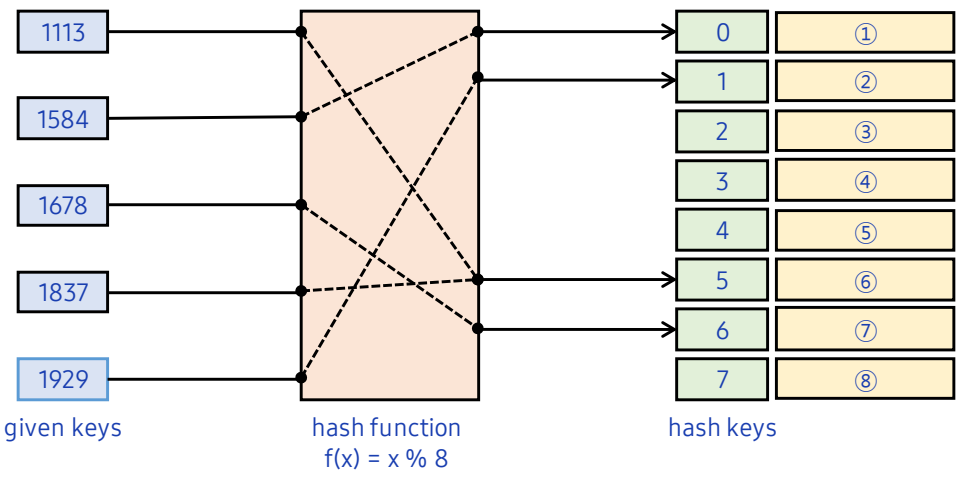
- The ord function returns ASCII Code values of a specific letter, so obtain the ASCII Value of each letter by using the map function.
- Designate sum of the ASCII Codes of all letters as the key value of the book.

1. Hash Table

1.3. Hashing and hash functions

Hashing refers to a process of converting a given key into another key, and hash function refers to a function that converts a random length key into another key that has a fixed length. The key converted by the hash function is called hash key.

There are 8 slots in the bookshelf, so define the hash function as f(x) = x % 8.



1. Hash Table

1.4. Define a hash table as a class type

Defining the hash table as a class is as follows.

```
1  class HashTable:
2
3      def __init__(self, size):
4          self.size = size
5          self.table = {}
6          for i in range(size):
7              self.table[i] = []
8
9      def hash(self, key):
10         return key % self.size
11
12     def get(self, key):
13         return self.table[self.hash(key)]
14
15     def put(self, key, value):
16         bucket = self.table[self.hash(key)]
17         if value not in bucket:
18             bucket.append(value)
```


1. Hash Table

1.4. Define a hash table as a class type

The hash table constructor initializes the hash table in the size of the given key.

```
1  class HashTable:
2
3      def __init__(self, size):
4          self.size = size
5          self.table = {}
6          for i in range(size):
7              self.table[i] = []
```

Line 3-7

- The 'size' as input parameter is the size of the hash table, and hash table is defined by Python dictionary.
- In the dictionary, initialize the bucket in number of size. At first, the bucket becomes an empty list.

1. Hash Table

1.4. Define a hash table as a class type

In the hash table, define the hash function as a function that performs modular arithmetic with the size of given key.

```
1  class HashTable:
2
3      def __init__(self, size):
4          self.size = size
5          self.table = {}
6          for i in range(size):
7              self.table[i] = []
8
9      def hash(self, key):
10         return key % self.size
11
```

Line 3-7

- The 'size' as input parameter is the size of the hash table, and hash table is defined by Python dictionary.
- In the dictionary, initialize the bucket in number of size. At first, the bucket becomes an empty list.

1. Hash Table

1.4. Define a hash table as a class type

The put() method receives key and value as input and stores the value in the hash key bucket.

```
11
12     def get(self, key):
13         return self.table[self.hash(key)]
14
15     def put(self, key, value):
16         bucket = self.table[self.hash(key)]
17         if value not in bucket:
18             bucket.append(value)
```

Line 15-18

- Find the bucket where the key value is self.hash(key) in the self.table dictionary.
- When duplication is not allowed, add the value in the bucket if the bucket does not have the value.

1. Hash Table

1.4. Define a hash table as a class type

The get() method receives key as input and returns the hash key bucket.

```
11
12      def get(self, key):
13          return self.table[self.hash(key)]
14
15      def put(self, key, value):
16          bucket = self.table[self.hash(key)]
17          if value not in bucket:
18              bucket.append(value)
```

Line 12-13

- Return the bucket where the key value is self.hash(key) in the self.table dictionary.

1. Hash Table

1.5. Example of using the hash table

Initialize the hash table that was previously defined into a table with 8 buckets.

```
1  table = HashTable(8)
2  for key in table.table.keys():
3      print(key, table.table[key])
```

```
0 []
1 []
2 []
3 []
4 []
5 []
6 []
7 []
```

1. Hash Table

1.5. Example of using the hash table

Add "The Little Prince" in the created hash table, and it will be stored in bucket 0.

```
table = HashTable(8)
book = "The Little Prince"
key = sum(map(ord, book))
table.put(key, book)
for key in table.table.keys():
    print(key, table.table[key])
```

```
0 ['The Little Prince']
1 []
2 []
3 []
4 []
5 []
6 []
7 []
```


1. Hash Table

1.5. Example of using the hash table

Put other four books in the bookshelf as follows.

```
books = [
    "The Old Man and the Sea",
    "The Little Mermaid",
    "Beauty and the Beast",
    "The Last Leaf"
]
for book in books:
    key = sum(map(ord, book))
    table.put(key, book)
for key in table.table.keys():
    print(key, table.table[key])
```

```
0 ['The Little Prince']
1 ['The Old Man and the Sea']
2 []
3 []
4 []
5 ['Beauty and the Beast', 'The Last Leaf']
6 ['The Little Mermaid']
7 []
```

1. Hash Table

1.5. Example of using the hash table

To find the slot where the "The Last Leaf" is located, use the get() method.

```
1 title = "The Last Leaf"
2 key = sum(map(ord, title))
3 bucket = table.get(key)
4 print(key, bucket)
```

```
1133 ['Beauty and the Beast', 'The Last Leaf']
```

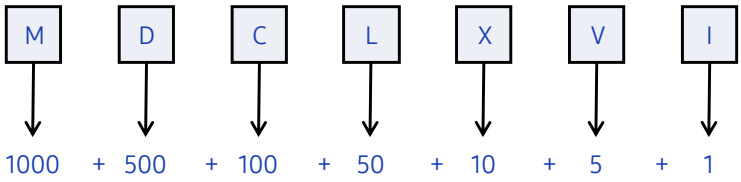

Let's code

1. Roman Numerals Converter

1.1. Define a hash table with a dictionary type

Take a look at the Roman numerals converter. Python dictionary can be considered as a hash table or hash map that converts Roman numerals into numbers by using the hash code of letters.

```
1 table = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
```



1. Roman Numerals Converter

1.2. Implementation and coding

Converting Roman numerals can be implemented as follows by using the hash table.

```
1 def roman_to_int(str):
2     result = 0
3     for i in range(len(str) - 1):
4         if table[str[i]] < table[str[i + 1]]:
5             result -= table[str[i]]
6         else:
7             result += table[str[i]]
8     return result + table[str[-1]]
```

Line 1-3

- The roman_to_int() function converts the str, which is input parameter, into Arabic numerals.
- The result value is initialized to 0 at first. Perform addition or subtraction to all letters.

1. Roman Numerals Converter

1.2. Implementation and coding

```
1  def roman_to_int(str):
2      result = 0
3      for i in range(len(str) - 1):
4          if table[str[i]] < table[str[i + 1]]:
5              result -= table[str[i]]
6          else:
7              result += table[str[i]]
8      return result + table[str[-1]]
```

Line 4-7

- If the letter value that is behind is greater than the current letter value, subtract it from the current value.
- If not, add the letter value to the current value.

1. Roman Numerals Converter

1.2. Implementation and coding

```
1  def roman_to_int(str):
2      result = 0
3      for i in range(len(str) - 1):
4          if table[str[i]] < table[str[i + 1]]:
5              result -= table[str[i]]
6          else:
7              result += table[str[i]]
8      return result + table[str[-1]]
```

Line 8

- When breaking the for-loop, all of the letters except of the last one are added or subtracted.
- So, return the sum of the values of str[-1] (the last letter) and result.

Pop quiz

Pop quiz

UNIT 26

Quiz. #1

Use the hash function of the hash table to calculate the key and hashkey of "Alice in Wonderland."

```
table = HashTable(8)
book = "Alice in WonderLand"
key = sum(map(ord, book))
print(key, table.hash(key))
```

Quiz. #2

If there are 10 slots in the bookshelf, calculate with the following code to find slots where each of the the following books will be put.

```
table = HashTable(10)
books = [
    "The Little Prince",
    "The Old Man and the Sea",
    "The Little Mermaid",
    "Beauty and the Beast",
    "The Last Leaf",
    "Alice in WonderLand"
]
for book in books:
    key = sum(map(ord, book))
    table.put(key, book)
for key in table.table.keys():
    print(key, table.table[key])
```


Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair programming UNIT 26

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Create a converter that converts Arabic numerals to Roman numerals by using the hash table provided below.

```
table = {1000:'M', 900:'CM', 500:'D', 400:'CD',
          100:'C',  90:'XC',  50:'L',  40:'XL',
           10:'X',   9:'IX',   5:'V',   4:'XI', 1:'I'}
```

```
num = int(input("Input a number: "))
print(int_to_roman(num))
```

```
Input a number: 1999
MCMXCIX
```

Examples of roman numbers:

369, 80, 29, 155, 14, 492, 348, 301, 469, 499

CCCLXIX, LXXX, XXIX, CLV, XIV, CDXCII, CCCXLVIII, CCCI, CDLXIX, CDXCIX,