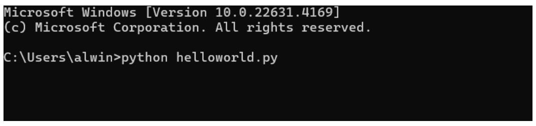
Competitive Programming using Python

Contents:

Python Installation:

https://www.python.org/

You can run python in the terminal using:



```
Microsoft Windows [Version 10.0.22631.4169]
(c) Microsoft Corporation. All rights reserved.

C:\Users\alwin>python helloworld.py
```

Basic Syntax and Arithematic Operators

- **Variables and Assignment**:
  - Variables in Python are dynamically typed, meaning you don't need to declare their type.
  - Assign values to variables using the `=` operator.

```
python
Copy code
x = 10   # x is assigned an integer value
name = "Alice"  # name is assigned a string
```

- **Comments**:
  - Single-line comments start with `#`.
  - Multi-line comments can be written using triple quotes (`'''` or `"""`).

```
python
Copy code
# This is a single-line comment

"""
This is a
multi-line comment
"""
```

- **Indentation**:
  - Python uses indentation (usually 4 spaces) to define code blocks, unlike other languages that use `{}` or keywords.

```
python
Copy code
if x > 5:
    print("x is greater than 5")
```

- **Functions**:
  - Define functions using the `def` keyword.

```
python
Copy code
def greet(name):
    print("Hello", name)

greet("Alice")
```

Basic Syntax in Python

1. **Variables and Assignment**:
   - Variables in Python are dynamically typed, meaning you don't need to declare their type.
   - Assign values to variables using the `=` operator.

```
python
Copy code
x = 10   # x is assigned an integer value
name = "Alice"  # name is assigned a string
```

2. **Comments**:
   - Single-line comments start with `#`.
   - Multi-line comments can be written using triple quotes (`'''` or `"""`).

```
python
Copy code
# This is a single-line comment

"""
This is a
multi-line comment
"""
```

3. **Indentation**:
   - Python uses indentation (usually 4 spaces) to define code blocks, unlike other languages that use `{}` or keywords.

```
python
Copy code
if x > 5:
    print("x is greater than 5")
```

4. **Functions**:
   - Define functions using the `def` keyword.

```
python
Copy code
def greet(name):
    print("Hello", name)

greet("Alice")
```

5. **Case Sensitivity**:
   - Python is case-sensitive, meaning `Var` and `var` are different variables.

Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Description	Example	Result
`+`	Addition	`x = 5 + 3`	`x = 8`
`-`	Subtraction	`x = 5 - 3`	`x = 2`
`*`	Multiplication	`x = 5 * 3`	`x = 15`
`/`	Division	`x = 5 / 2`	`x = 2.5`
`//`	Floor Division	`x = 5 // 2`	`x = 2`
`%`	Modulus (Remainder)	`x = 5 % 2`	`x = 1`
`**`	Exponentiation (Power)	`x = 5 ** 2`	`x = 25`

Bitwise Operators:

Bitwise operators work on bits and perform bit-by-bit operations.

Operator	Name	Description	Example	Result
`&`	AND	Sets each bit to 1 if both bits are 1	`x = 5 & 3`	`x = 1`
`	`	OR	Sets each bit to 1 if one of the bits is 1	`x = 5

`^`	XOR	Sets each bit to 1 if only one of the bits is 1	`x = 5 ^ 3`	`x = 6`
`~`	NOT	Inverts all the bits	`x = ~5`	`x = -6`
`<<`	Left Shift	Shifts bits to the left and fills with zeros	`x = 5 << 1`	`x = 10`
`>>`	Right Shift	Shifts bits to the right	`x = 5 >> 1`	`x = 2`

Comparison (Relational) Operators

These operators compare two values and return a boolean result (`True` or `False`).

Operator	Description	Example	Result
`==`	Equal to	`5 == 3`	`False`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`5 < 3`	`False`
`>=`	Greater than or equal to	`5 >= 5`	`True`
`<=`	Less than or equal to	`5 <= 3`	`False`

Assignment Operators

These operators assign values to variables.

Operator	Description	Example	Result
`=`	Assigns the right-hand value to the left	`x = 5`	`x = 5`
`+=`	Adds right-hand value to the left-hand variable	`x += 3`	`x = x + 3`
`-=`	Subtracts right-hand value from the left-hand variable	`x -= 3`	`x = x - 3`
`*=`	Multiplies left-hand variable by the right-hand value	`x *= 3`	`x = x * 3`
`/=`	Divides left-hand variable by the right-hand value	`x /= 3`	`x = x / 3`
`//=`	Performs floor division	`x //= 3`	`x = x // 3`
`%=`	Performs modulus operation	`x %= 3`	`x = x % 3`
`**=`	Performs exponentiation	`x **= 3`	`x = x ** 3`
`&=`	Performs bitwise AND	`x &= 3`	`x = x & 3`
`	=`	Performs bitwise OR	`x
`^=`	Performs bitwise XOR	`x ^= 3`	`x = x ^ 3`
`<<=`	Performs left shift	`x <<= 3`	`x = x << 3`
`>>=`	Performs right shift	`x >>= 3`	`x = x >> 3`

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description	Example	Result
`and`	Returns `True` if both statements are true	`True and False`	`False`
`or`	Returns `True` if one of the statements is true	`True or False`	`True`
`not`	Reverses the result of a condition	`not True`	`False`

Identity Operators

These operators check if two objects are identical (same memory location).

Operator	Description	Example	Result
`is`	Returns `True` if two variables point to the same object	`x is y`	`True/False`
`is not`	Returns `True` if two variables do not point to the same object	`x is not y`	`True/False`

Membership Operators

These operators test whether a value is a member of a sequence (like strings, lists, or sets).

Operator	Description	Example	Result
`in`	Returns `True` if the value is present in the sequence	`'a' in 'apple'`	`True`
`not in`	Returns `True` if the value is not present in the sequence	`'b' not in 'apple'`	`True`

Python Input and Output

Basic Datatypes in Python:

Here is a table with the basic Python data types:

Data Type	Description	Example
`int`	Whole numbers, positive or negative	`x = 10`, `y = -5`
`float`	Numbers with a decimal point	`x = 10.5`, `y = -3.14`
`str`	Sequence of characters (strings)	`x = "Hello"`, `y = 'World'`
`bool`	Boolean values (`True` or `False`)	`x = True`, `y = False`
`list`	Ordered collection of items (mutable)	`x = [1, 2, 3, "apple", 4.5]`
`tuple`	Ordered collection of items (immutable)	`x = (1, 2, "apple", 4.5)`

`dict`	Collection of key-value pairs	`x = {"name": "John", "age": 25}`
`set`	Unordered collection of unique items	`x = {1, 2, 3, 4}`
`NoneType`	Represents the absence of a value	`x = None`

User Input:

`input()` : It is used for **taking the user input**. (It we be of **type string by default**, So if we try to input an integer we should convert the input into integer.)

eg:

`input() ⇒ String`

`int(input()) ⇒ integer`

`float(input()) ⇒ float`

we can give comments for inputting inside the input() method:

`int(input("Enter the number");`

Basic Output:

The `print()` function is used to display output on the screen.

Example:

```
python
Copy code
name = "Alice"
age = 25
print("Name:", name, "Age:", age)
```

Formatting Output:

You can use various methods to format output in Python:

- **Using commas** (default space separator):

```
python
Copy code
print("Name:", name, "Age:", age)
```

- **Using `+` to concatenate strings**:

```
python
Copy code
print("Name: " + name + ", Age: " + str(age))
```

- **Using f-strings (Python 3.6+)**:

```
python
Copy code
print(f"Name: {name}, Age: {age}")
```

- **Using `format()`**:

```
python
Copy code
print("Name: {}, Age: {}".format(name, age))
```

Example Combining Input and Output:

```
python
Copy code
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print(f"Hello {name}, you are {age} years old.")
```

This demonstrates how to use input and output efficiently in Python.

Control Statements:

. `if` Statement

The `if` statement is used to execute a block of code if a specified condition is `True`.

```
python
Copy code
x = 10
if x > 5:
    print("x is greater than 5")
```

2. `else` Statement

The `else` statement is used to execute a block of code if the `if` condition is `False`.

```
python
Copy code
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

3. `elif` Statement

The `elif` statement allows you to check multiple conditions. It is used after an `if` statement, and if the `if` condition is `False`, Python checks the `elif` condition.

```
python
Copy code
x = 5
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

4. `for` Loop

The `for` loop is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string).

```
python
Copy code
for i in range(3):
    print(i)  # Output: 0 1 2
```

5. `while` Loop

The `while` loop repeats a block of code as long as the condition is `True`.

```
python
Copy code
x = 0
while x < 3:
    print(x)
    x += 1  # Output: 0 1 2
```

6. `break` Statement

The `break` statement is used to exit the nearest enclosing loop prematurely.

```
python
Copy code
for i in range(5):
    if i == 3:
        break
    print(i)  # Output: 0 1 2
```

7. `continue` Statement

The `continue` statement is used to skip the rest of the current iteration and move to the next iteration of the loop.

```
python
Copy code
for i in range(5):
    if i == 3:
        continue
    print(i)  # Output: 0 1 2 4
```

8. `pass` Statement

The `pass` statement does nothing and acts as a placeholder where code is syntactically required but no action is needed.

```
python
Copy code
x = 10
if x > 5:
    pass  # Do nothing
else:
    print("x is less than or equal to 5")
```

9. `try-except` Statement

The `try-except` block is used to handle exceptions (errors) that occur during execution.

```
python
Copy code
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

10. `finally` Statement

The `finally` block executes code after the `try-except` block, regardless of whether an exception occurred or not.

```
python
Copy code
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Error")
finally:
    print("This will always execute")
```

11. `raise` Statement

The `raise` statement is used to manually raise an exception.

```
python
Copy code
raise ValueError("This is an error message")
```

12. `return` Statement

The `return` statement is used to exit a function and optionally return a value.

```
python
Copy code
def add(a, b):
    return a + b

result = add(3, 4)
print(result)  # Output: 7
```

13. `with` Statement

The `with` statement is used to wrap the execution of a block of code within methods defined by a context manager. It simplifies resource management, such as file handling.

```
python
Copy code
with open("file.txt", "r") as file:
    content = file.read()
    print(content)
```

Built-in Function for Competitive Programming

Category	Function	Description	Example
Input/Output	`input()`	Reads a line of input from the user.	`user_input = input()`
	`print()`	Outputs data to the console.	`print("Hello, World!")`
Type Conversion	`int()`	Converts a string/number to an integer.	`num = int("42")`
	`float()`	Converts a string/number to a floating-point number.	`num = float("3.14")`
	`str()`	Converts an object to a string.	`s = str(42)`
	`list()`	Converts an iterable to a list.	`lst = list("abc")`
Math Functions	`abs()`	Returns the absolute value of a number.	`result = abs(-5)`
	`max()`	Returns the largest item in an iterable.	`result = max([1, 3, 2])`
	`min()`	Returns the smallest item in an iterable.	`result = min([1, 3, 2])`
	`sum()`	Sums elements in an iterable.	`total = sum([1, 2, 3])`
	`round()`	Rounds a floating-point number to the given precision.	`result = round(3.14159, 2)`
Iterators/Looping	`range()`	Generates a sequence of numbers.	`for i in range(3): print(i)`
	`enumerate()`	Adds a counter to an iterable, returns an enumerate object.	`for idx, val in enumerate(['a', 'b', 'c']): print(idx, val)`
	`zip()`	Combines multiple iterables, returns a zip object with paired elements.	`for a, b in zip([1, 2], ['x', 'y']): print(a, b)`
Sorting/Ordering	`sorted()`	Returns a sorted list from an iterable.	`result = sorted([3, 1, 2])`
	`reversed()`	Returns an iterator that accesses elements in reverse order.	`result = list(reversed([1, 2, 3]))`
Set and List Operations	`len()`	Returns the length of an object (list, string, etc.).	`length = len([1, 2, 3])`
	`set()`	Converts an iterable to a set (removes duplicates).	`unique_elements = set([1, 2, 2, 3])`
String Functions	`len()`	Returns the length of a string.	`length = len("hello")`
	`split()`	Splits a string into a list.	`result = "a b c".split()`
	`join()`	Joins a list of strings into a single string with a delimiter.	`result = " ".join(['a', 'b', 'c'])`
Type Checking	`isinstance()`	Checks if an object is of a specific type.	`result = isinstance(5, int)`
	`type()`	Returns the type of an object.	`result = type(5)`
Advanced Functions	`map()`	Applies a function to all items in an iterable.	`result = list(map(int, ['1', '2', '3']))`
	`filter()`	Filters elements from an iterable based on a function returning `True/False`.	`result = list(filter(lambda x: x > 1, [1, 2, 3]))`
	`all()`	Returns `True` if all elements in an iterable are `True`.	`result = all([True, True, False])`
	`any()`	Returns `True` if any element in an iterable is `True`.	`result = any([False, True, False])`
Miscellaneous	`eval()`	Evaluates a string as a Python expression.	`result = eval("3 + 5")`
	`exec()`	Executes Python code dynamically.	`exec("x = 5; print(x)")`

Basics of Array and Array Manipulations:

Array Basics

In Python, arrays are typically implemented as lists. Arrays are indexed collections that store elements of the same or different data types. Here's a summary of basic operations:

Creating an Array

- **Static Array (fixed values)**:

```
python
Copy code
arr = [1, 2, 3, 4, 5]
```

- **Dynamic Array (size specified, filled with default values)**:

```
python
Copy code
arr = [0] * 5  # [0, 0, 0, 0, 0]
```

- **2D Array (matrix)**:

```
python
Copy code
matrix = [[0] * 3 for _ in range(3)]  # 3x3 matrix with all elements 0
```

Array Inputs

1. Single Line Input of Space-Separated Elements

This is the most common format where multiple elements are entered in a single line, separated by spaces.

Example: Taking Input as Integers

- **Input Format:** `1 2 3 4 5`

```
python
Copy code
arr = list(map(int, input().split()))
```

Explanation:

- `input()` reads the entire line as a string.
- `split()` breaks the string into individual components based on spaces.
- `map(int, ...)` converts each component to an integer.
- `list()` converts the result to a list.

Example: Taking Input as Strings

- **Input Format:** `apple banana cherry`

```
python
Copy code
arr = input().split()
```

Explanation:

- `split()` breaks the input string into a list of strings based on spaces.

2. Multiple Lines Input for an Array

In this case, each element of the array is entered on a new line.

Example: Enter `n` Elements on Separate Lines

```
python
Copy code
n = int(input())  # Take number of elements
arr = [int(input()) for _ in range(n)]
```

Explanation:

- First, you take the number of elements (`n`).
- Then, you use a list comprehension with a `for` loop to input each element on a new line.

3. Input for 2D Array (Matrix)

You can input a 2D array either in a single line or across multiple lines, depending on the format.

Single Line Input (Space-Separated)

- **Input Format (3×3 Matrix):** `1 2 3 4 5 6 7 8 9`

```
python
Copy code
matrix = []
rows, cols = 3, 3  # specify the number of rows and columns
elements = list(map(int, input().split()))  # take all elements in one line
for i in range(rows):
    matrix.append(elements[i*cols:(i+1)*cols])
```

Explanation:

- `elements` stores all the input values.
- The loop slices the list to create rows and appends them to `matrix`.

Multiple Lines Input (Each Row on a New Line)

- **Input Format:** `1 2 34 5 67 8 9`

```
python
Copy code
matrix = []
for _ in range(3):  # Number of rows (change 3 to the required row count)
    row = list(map(int, input().split()))
    matrix.append(row)
```

Explanation:

- You input each row separately and append it to the `matrix` list.

4. Using List Comprehension for Input

List comprehension can be used to shorten array input code for competitive programming.

Single Line Input:

```
python
Copy code
arr = [int(x) for x in input().split()]
```

2D Array Input (Multiple Lines):

```
python
Copy code
matrix = [[int(x) for x in input().split()] for _ in range(3)]  # 3 rows
```

5. Example: Taking Input for 1D and 2D Arrays

Input for a 1D Array (Space-separated numbers)

- **Input:** `5 10 15 20`

```
python
Copy code
arr = list(map(int, input().split()))
print(arr)  # Output: [5, 10, 15, 20]
```

Input for a 2D Array (Matrix, Row-wise)

- **Input:** `1 2 34 5 67 8 9`

```
python
Copy code
matrix = []
for _ in range(3):  # For a 3x3 matrix
    matrix.append(list(map(int, input().split())))
print(matrix)  # Output: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

6. Special Cases

Input for Array with Fixed Length

Sometimes, you may know in advance that the array has a fixed length. You can combine this with other methods:

```
python
Copy code
n = 5  # Fixed number of elements
arr = [int(input()) for _ in range(n)]  # Take 5 elements, one per line
```

Input with Multiple Arrays

If the problem requires input for multiple arrays, you can simply repeat the above patterns:

```
python
Copy code
# Input two arrays of size n
n = int(input())  # Number of elements in each array
arr1 = list(map(int, input().split()))  # First array
arr2 = list(map(int, input().split()))  # Second array
```

Accessing Elements

- Access elements using an index (0-based):

  ```
  python
  Copy code
  print(arr[2])  # Output: 3
  ```

Modifying Elements

- Modify elements by index:

```
python
Copy code
arr[1] = 10  # arr becomes [1, 10, 3, 4, 5]
```

Common Array Operations

1. Traversing an Array

You can traverse (iterate through) an array using a loop:

- **Using `for` loop**:

```
python
Copy code
for element in arr:
    print(element)
```

- **Using `for` loop with index**:

```
python
Copy code
for i in range(len(arr)):
    print(arr[i])
```

2. Adding Elements

- **At the End**:

```
python
Copy code
arr.append(6)  # arr becomes [1, 2, 3, 4, 5, 6]
```

- **At a Specific Position**:

```
python
Copy code
arr.insert(2, 9)  # arr becomes [1, 2, 9, 3, 4, 5]
```

3. Removing Elements

- **By Value**:

```
python
Copy code
arr.remove(4)  # arr becomes [1, 2, 3, 5] (removes first occurrence of 4)
```

- **By Index**:

```
python
Copy code
arr.pop(2)  # arr becomes [1, 2, 4, 5] (removes element at index 2)
```

- **Remove Last Element**:

```
python
Copy code
arr.pop()  # arr becomes [1, 2, 3, 4] (removes the last element)
```

4. Finding Elements

- **Find index of an element**:

```
python
Copy code
index = arr.index(3)  # Output: 2 (index of the first occurrence of 3)
```

- **Count occurrences of an element**:

```
python
Copy code
count = arr.count(3)  # Output: 1
```

5. Sorting an Array

- **Ascending Order**:

```
python
Copy code
arr.sort()  # arr becomes [1, 2, 3, 4, 5]
```

- **Descending Order**:

```
python
Copy code
arr.sort(reverse=True)  # arr becomes [5, 4, 3, 2, 1]
```

- **Sorted Without Modifying Original**:

```
python
Copy code
sorted_arr = sorted(arr)  # returns sorted version, arr remains unchanged
```

6. Reversing an Array

- **In-place reversal**:

```
python
Copy code
arr.reverse()  # arr becomes [5, 4, 3, 2, 1]
```

7. Slicing an Array

- **Extracting a portion of an array**:

```
python
Copy code
subarray = arr[1:4]  # arr[1] to arr[3], subarray becomes [2, 3, 4]
```

- **Slicing with step**:

```
python
Copy code
subarray = arr[::2]  # Every second element, subarray becomes [1, 3, 5]
```

3. 2D Arrays (Matrices)

For competitive programming, working with 2D arrays is common for grid-related problems.

Creating a 2D Array

- **Example: 3×3 matrix initialized with 0s**:

```
python
Copy code
matrix = [[0] * 3 for _ in range(3)]  # Creates a 3x3 matrix of 0s
```

Accessing/Modifying 2D Array Elements

- **Accessing an element**:

```
python
Copy code
print(matrix[1][2])  # Access element at row 1, column 2
```

- **Modifying an element**:

```
python
Copy code
matrix[2][2] = 5  # Changes the value at row 2, column 2 to 5
```

Traversing a 2D Array

- **Using nested loops**:

```
python
Copy code
for row in matrix:
    for element in row:
        print(element)
```

Row/Column Operations

- **Row slicing**:

```
python
Copy code
row = matrix[1]  # Gets the second row
```

- **Column extraction**:

```
python
Copy code
column = [matrix[i][2] for i in range(len(matrix))]  # Extracts the 3rd column
```

Algorithms Regarding Array Manipulation

Kadane's Algorithm (Maximum Subarray Sum)

Problem: Find the largest sum of a contiguous subarray within a given one-dimensional array.

Algorithm:

- Iterate through the array, maintaining a running sum. If the sum becomes negative, reset it to zero.
- Track the maximum sum encountered.

Code:

```
python
Copy code
def kadane(arr):
    max_so_far = arr[0]
    max_ending_here = arr[0]

    for i in range(1, len(arr)):
        max_ending_here = max(arr[i], max_ending_here + arr[i])
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far

# Example
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(kadane(arr))  # Output: 6 (Subarray: [4, -1, 2, 1])
```

Explanation:

- At each step, decide whether to add the current element to the previous sum or start a new subarray. The maximum subarray sum is stored and returned.

---

2. Two Pointer Technique

Problem: Find if there exists two numbers in a sorted array that sum up to a target value.

Algorithm:

- Start with two pointers: one at the beginning and the other at the end of the array.
- If the sum of the two elements is greater than the target, move the right pointer left. If it's smaller, move the left pointer right.

Code:

```
python
Copy code
def two_pointer(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
        current_sum = arr[left] + arr[right]
        if current_sum == target:
            return (left, right)  # Return indices of the elements
        elif current_sum < target:
            left += 1
        else:
            right -= 1
    return -1

# Example
arr = [1, 2, 3, 4, 6]
target = 9
print(two_pointer(arr, target))  # Output: (2, 4) (Elements: 3 + 6)
```

Explanation:

- By leveraging the sorted nature of the array, this technique efficiently finds pairs whose sum matches the target in O(n) time.

3. Sliding Window Technique

Problem: Find the maximum sum of `k` consecutive elements in an array.

Algorithm:

- Maintain a "window" of size `k` and slide it across the array.
- Calculate the sum of the first window and for each new position, adjust the sum by subtracting the element going out of the window and adding the element coming into the window.

Code:

```
python
Copy code
def sliding_window(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i - k]
        max_sum = max(max_sum, window_sum)

    return max_sum

# Example
arr = [2, 1, 5, 1, 3, 2]
k = 3
print(sliding_window(arr, k))  # Output: 9 (Subarray: [5, 1, 3])
```

Explanation:

- We calculate the sum for each sliding window in constant time by adjusting the previous sum.

4. Prefix Sum Array

Problem: Efficiently calculate the sum of any subarray in constant time.

Algorithm:

- Construct a prefix sum array where each element at index `i` represents the sum of all elements from the start of the array to `i`. Using this array, the sum of any subarray can be calculated as:

  ```
  python
  Copy code
  sum(i, j) = prefix[j+1] - prefix[i]
  ```

Code:

```
python
Copy code
def prefix_sum(arr):
    n = len(arr)
```

```
    prefix = [0] * (n + 1)

    for i in range(1, n + 1):
        prefix[i] = prefix[i - 1] + arr[i - 1]

    return prefix

# Example
arr = [1, 2, 3, 4, 5]
prefix = prefix_sum(arr)

# Sum of subarray from index 1 to 3 (2 + 3 + 4)
print(prefix[4] - prefix[1])  # Output: 9
```

Explanation:

- The prefix sum array allows subarray sum queries to be answered in constant time, making it efficient for problems where multiple sum queries are needed.

5. Binary Search on a Sorted Array

Problem: Find a specific element in a sorted array in O(log n) time.

Algorithm:

- Use binary search by repeatedly dividing the search range in half until the target is found or the range is empty.

Code:

```
python
Copy code
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid  # Element found at index mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # Element not found

# Example
arr = [1, 3, 5, 7, 9]
target = 5
print(binary_search(arr, target))  # Output: 2
```

Explanation:

- Binary search works on sorted arrays and efficiently finds the target element by halving the search space at each step.

6. Dutch National Flag Problem (3-Way Partition)

Problem: Given an array with three distinct values (for example, 0, 1, and 2), sort the array in linear time.

Algorithm:

- The array is divided into three sections: one for each value. The key is to maintain the boundaries of these sections and swap elements to ensure the correct order.

Code:

```
python
Copy code
def dutch_national_flag(arr):
    low, mid, high = 0, 0, len(arr) - 1

    while mid <= high:
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1

# Example
arr = [2, 0, 1, 2, 1, 0]
dutch_national_flag(arr)
print(arr)  # Output: [0, 0, 1, 1, 2, 2]
```

Explanation:

- This algorithm sorts an array of three distinct values in O(n) time by maintaining three pointers to manage the positions of each value.

7. Merge Intervals

Problem: Given a list of intervals, merge all overlapping intervals.

Algorithm:

- Sort the intervals by their starting points and then iterate through the list, merging overlapping intervals.

Code:

```
python
Copy code
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []

    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])

    return merged

# Example
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
print(merge_intervals(intervals))  # Output: [[1, 6], [8, 10], [15, 18]]
```

Explanation:

- The intervals are merged by maintaining the last merged interval and updating its end if the current interval overlaps.

Other Important Problems for Reference

Finding the Kth Largest/Smallest Element

Problem: Given an array, find the Kth largest or smallest element.

Algorithm:

- Use a max-heap or min-heap (or sort the array) to find the Kth largest/smallest element efficiently.

Code:

```
python
Copy code
import heapq

def kth_largest(arr, k):
    return heapq.nlargest(k, arr)[-1]  # Returns the kth largest element

def kth_smallest(arr, k):
    return heapq.nsmallest(k, arr)[-1]  # Returns the kth smallest element

# Example
arr = [3, 2, 1, 5, 6, 4]
k = 2
print(kth_largest(arr, k))  # Output: 5 (2nd largest)
print(kth_smallest(arr, k))  # Output: 2 (2nd smallest)
```

2. Rotate Array

Problem: Rotate an array to the right by `k` steps.

Algorithm:

- Reverse the whole array, then reverse the first `k` elements and the rest of the array.

Code:

```
python
Copy code
def rotate(arr, k):
    n = len(arr)
    k %= n  # In case k is larger than n
    arr.reverse()
    arr[:k] = reversed(arr[:k])
    arr[k:] = reversed(arr[k:])

# Example
arr = [1, 2, 3, 4, 5, 6, 7]
rotate(arr, 3)
print(arr)  # Output: [5, 6, 7, 1, 2, 3, 4]
```

3. Remove Duplicates from Sorted Array

Problem: Given a sorted array, remove duplicates in-place and return the new length.

Algorithm:

- Use two pointers to track unique elements.

Code:

```
python
Copy code
def remove_duplicates(arr):
    if not arr:
        return 0

    unique_index = 1

    for i in range(1, len(arr)):
        if arr[i] != arr[i - 1]:
            arr[unique_index] = arr[i]
            unique_index += 1

    return unique_index

# Example
arr = [1, 1, 2, 2, 3, 4]
length = remove_duplicates(arr)
print(arr[:length])  # Output: [1, 2, 3, 4]
```

4. Sorting an Array of 0s, 1s, and 2s (Dutch National Flag Problem)

Problem: Sort an array containing only 0s, 1s, and 2s.

Code:

```
python
Copy code
def sort_012(arr):
    low, mid, high = 0, 0, len(arr) - 1

    while mid <= high:
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1

# Example
arr = [0, 1, 2, 0, 1, 2]
sort_012(arr)
```

```
print(arr)  # Output: [0, 0, 1, 1, 2, 2]
```

5. Finding the Intersection of Two Arrays

Problem: Find common elements between two arrays.

Algorithm:

- Use a set to track elements of the first array, then iterate through the second array to find intersections.

Code:

```
python
Copy code
def intersection(arr1, arr2):
    set1 = set(arr1)
    return [x for x in arr2 if x in set1]

# Example
arr1 = [1, 2, 2, 1]
arr2 = [2, 2]
print(intersection(arr1, arr2))  # Output: [2, 2]
```

6. Prefix Product Array

Problem: Create an array where each element is the product of all other elements in the original array except itself.

Algorithm:

- Use two passes to calculate the prefix and suffix products.

Code:

```
python
Copy code
def product_array(arr):
    n = len(arr)
    prefix = [1] * n
    suffix = [1] * n
    result = [0] * n

    for i in range(1, n):
        prefix[i] = prefix[i - 1] * arr[i - 1]

    for i in range(n - 2, -1, -1):
        suffix[i] = suffix[i + 1] * arr[i + 1]

    for i in range(n):
        result[i] = prefix[i] * suffix[i]

    return result

# Example
arr = [1, 2, 3, 4]
print(product_array(arr))  # Output: [24, 12, 8, 6]
```

7. Finding Maximum and Minimum in an Array

Problem: Find the maximum and minimum elements in an array.

Algorithm:

- Traverse the array once to find both maximum and minimum values.

Code:

```
python
Copy code
def find_max_min(arr):
    max_val = min_val = arr[0]

    for num in arr:
        if num > max_val:
            max_val = num
        if num < min_val:
            min_val = num
    return max_val, min_val

# Example
arr = [3, 5, 1, 8, 2]
max_val, min_val = find_max_min(arr)
print(f"Max: {max_val}, Min: {min_val}")  # Output: Max: 8, Min: 1
```

8. Counting Inversions

Problem: Count the number of inversions in an array. An inversion is a pair of indices (i, j) such that i < j and arr[i] > arr[j].

Algorithm:

- Use a modified merge sort to count inversions while sorting the array.

Code:

```
python
Copy code
def merge_and_count(arr, temp_arr, left, mid, right):
    i = left    # Starting index for left subarray
    j = mid + 1 # Starting index for right subarray
    k = left    # Starting index to be sorted
    inv_count = 0

    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            i += 1
        else:
            temp_arr[k] = arr[j]
            inv_count += (mid - i + 1)  # Count inversions
            j += 1
        k += 1

    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1

    while j <= right:
        temp_arr[k] = arr[j]
        j += 1
        k += 1

    for i in range(left, right + 1):
        arr[i] = temp_arr[i]

    return inv_count

def merge_sort_and_count(arr, temp_arr, left, right):
    inv_count = 0
    if left < right:
        mid = (left + right) // 2
        inv_count += merge_sort_and_count(arr, temp_arr, left, mid)
        inv_count += merge_sort_and_count(arr, temp_arr, mid + 1, right)
        inv_count += merge_and_count(arr, temp_arr, left, mid, right)

    return inv_count

def count_inversions(arr):
    temp_arr = [0] * len(arr)
    return merge_sort_and_count(arr, temp_arr, 0, len(arr) - 1)

# Example
arr = [1, 20, 6, 4, 5]
print(count_inversions(arr))  # Output: 5
```

9. Finding All Pairs with a Given Sum

Problem: Given an array and a sum, find all pairs in the array that add up to the given sum.

Algorithm:

- Use a hash set to track the elements needed to form pairs.

Code:

```
python
Copy code
def find_pairs_with_sum(arr, target_sum):
    pairs = []
    seen = set()

    for number in arr:
        complement = target_sum - number
        if complement in seen:
            pairs.append((complement, number))
        seen.add(number)

    return pairs

# Example
arr = [1, 2, 3, 4, 5, 6]
target_sum = 7
print(find_pairs_with_sum(arr, target_sum))  # Output: [(3, 4), (2, 5), (1, 6)]
```

10. Finding Longest Increasing Subsequence (LIS)

Problem: Given an array, find the length of the longest increasing subsequence.

Algorithm:

- Use dynamic programming or binary search to optimize the approach.

Code:

```
python
Copy code
def longest_increasing_subsequence(arr):
    if not arr:
        return 0

    dp = []

    for num in arr:
        pos = binary_search(dp, num)
        if pos == len(dp):
            dp.append(num)
        else:
            dp[pos] = num

    return len(dp)

def binary_search(dp, num):
    left, right = 0, len(dp)
    while left < right:
        mid = (left + right) // 2
        if dp[mid] < num:
            left = mid + 1
        else:
            right = mid
    return left

# Example
arr = [10, 9, 2, 5, 3, 7, 101, 18]
print(longest_increasing_subsequence(arr))  # Output: 4 (LIS: [2, 3, 7, 101])
```

Strings and String Manipulation

Basics of Strings

- **Definition**: A string is a sequence of characters enclosed in single quotes (`'`) or double quotes (`"`).
- **Immutability**: Strings in Python are immutable, meaning once created, their contents cannot be changed. Operations on strings return new strings.

String Creation

```
python
Copy code
string1 = 'Hello, World!'
```

```
string2 = "Python Programming"
```

Common String Manipulation Functions

Function	Description	Example
`len(s)`	Returns the length of the string.	`len("Hello")` → 5
`s.lower()`	Converts all characters to lowercase.	`"HELLO".lower()` → `"hello"`
`s.upper()`	Converts all characters to uppercase.	`"hello".upper()` → `"HELLO"`
`s.strip()`	Removes whitespace from the beginning and end of the string.	`" hello ".strip()` → `"hello"`
`s.split(delimiter)`	Splits the string into a list based on the delimiter.	`"a,b,c".split(",")` → `["a", "b", "c"]`
`s.join(iterable)`	Joins the elements of an iterable into a string with the string as a separator.	`",".join(["a", "b", "c"])` → `"a,b,c"`
`s.replace(old, new)`	Replaces occurrences of a substring with another substring.	`"hello world".replace("world", "Python")` → `"hello Python"`
`s.find(sub)`	Returns the lowest index of the substring if found, otherwise -1.	`"hello".find("e")` → 1
`s.index(sub)`	Similar to `find()`, but raises a ValueError if the substring is not found.	`"hello".index("e")` → 1
`s.count(sub)`	Returns the number of occurrences of a substring in the string.	`"hello".count("l")` → 2
`s.startswith(prefix)`	Returns `True` if the string starts with the specified prefix.	`"hello".startswith("he")` → `True`
`s.endswith(suffix)`	Returns `True` if the string ends with the specified suffix.	`"hello".endswith("lo")` → `True`
`s.isalpha()`	Returns `True` if all characters in the string are alphabetic.	`"hello".isalpha()` → `True`
`s.isdigit()`	Returns `True` if all characters in the string are digits.	`"123".isdigit()` → `True`
`s.isalnum()`	Returns `True` if all characters in the string are alphanumeric.	`"abc123".isalnum()` → `True`
`s.capitalize()`	Capitalizes the first character of the string.	`"hello".capitalize()` → `"Hello"`
`s.title()`	Capitalizes the first character of each word in the string.	`"hello world".title()` → `"Hello World"`

String Formatting

1. F-strings (Python 3.6+)

```
python
Copy code
name = "Alice"
age = 25
greeting = f"Hello, my name is {name} and I am {age} years old."
```

2. `str.format()` Method

```
python
Copy code
greeting = "Hello, my name is {} and I am {} years old.".format(name, age)
```

3. Percent Formatting

```
python
Copy code
greeting = "Hello, my name is %s and I am %d years old." % (name, age)
```

Examples of String Manipulation

```
python
Copy code
# Example 1: String Length
my_string = "Hello, World!"
print(len(my_string))  # Output: 13

# Example 2: Lowercase and Uppercase
print(my_string.lower())  # Output: hello, world!
print(my_string.upper())  # Output: HELLO, WORLD!

# Example 3: Stripping Whitespace
whitespace_string = "   Hello, World!   "
print(whitespace_string.strip())  # Output: "Hello, World!"

# Example 4: Splitting and Joining
fruits = "apple,banana,cherry"
fruit_list = fruits.split(",")
print(fruit_list)  # Output: ['apple', 'banana', 'cherry']
joined_fruits = " | ".join(fruit_list)
print(joined_fruits)  # Output: "apple | banana | cherry"
```

```
# Example 5: Replacing Substrings
new_string = my_string.replace("World", "Python")
print(new_string)  # Output: "Hello, Python!"

# Example 6: Finding Substring
index = my_string.find("World")
print(index)  # Output: 7

# Example 7: String Formatting
name = "Bob"
age = 30
formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)  # Output: "My name is Bob and I am 30 years old."
```

Important String Problems for Reference

1. Palindrome Check

Problem: Determine if a string is a palindrome (reads the same forwards and backwards).

Algorithm:

- Reverse the string and compare it with the original.

Code:

```
python
Copy code
def is_palindrome(s):
    return s == s[::-1]

# Example
s = "racecar"
print(is_palindrome(s))  # Output: True
```

2. Anagram Check

Problem: Check if two strings are anagrams (contain the same characters in different orders).

Algorithm:

- Sort both strings and compare, or use a frequency count of characters.

Code:

```
python
Copy code
def are_anagrams(s1, s2):
    return sorted(s1) == sorted(s2)

# Example
s1 = "listen"
s2 = "silent"
print(are_anagrams(s1, s2))  # Output: True
```

3. String Compression

Problem: Implement basic string compression using the counts of repeated characters.

Algorithm:

- Traverse the string and count consecutive characters.

Code:

```
python
Copy code
def compress_string(s):
    compressed = []
    count = 1

    for i in range(1, len(s)):
        if s[i] == s[i - 1]:
            count += 1
        else:
            compressed.append(s[i - 1] + str(count))
            count = 1
    compressed.append(s[-1] + str(count))  # Add the last group
    return ''.join(compressed)

# Example
s = "aabcccccaaa"
print(compress_string(s))  # Output: "a2b1c5a3"
```

---

4. Longest Common Substring

Problem: Find the longest common substring between two strings.

Algorithm:

- Use dynamic programming to build a table of lengths of common substrings.

Code:

```
python
Copy code
def longest_common_substring(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    longest_length = 0
    end_index = 0

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                if dp[i][j] > longest_length:
                    longest_length = dp[i][j]
                    end_index = i
            else:
                dp[i][j] = 0

    return s1[end_index - longest_length:end_index]

# Example
s1 = "abcdef"
s2 = "zcdemf"
print(longest_common_substring(s1, s2))  # Output: "cd"
```

---

5. Longest Palindromic Substring

Problem: Find the longest palindromic substring in a given string.

Algorithm:

- Expand around potential centers (each character and each pair of characters).

Code:

```
python
Copy code
def longest_palindromic_substring(s):
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]

    longest = ""
    for i in range(len(s)):
        # Odd length palindromes
        odd_palindrome = expand_around_center(i, i)
        if len(odd_palindrome) > len(longest):
            longest = odd_palindrome
        # Even length palindromes
        even_palindrome = expand_around_center(i, i + 1)
        if len(even_palindrome) > len(longest):
            longest = even_palindrome

    return longest

# Example
s = "babad"
print(longest_palindromic_substring(s))  # Output: "bab" or "aba"
```

6. KMP Algorithm (Knuth-Morris-Pratt)

Problem: Efficiently search for occurrences of a pattern within a text.

Algorithm:

- Preprocess the pattern to create a longest prefix-suffix (LPS) array, then use it to skip unnecessary comparisons.

Code:

```
python
Copy code
def KMP_search(pattern, text):
    lps = compute_lps(pattern)
    i = j = 0
    occurrences = []

    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == len(pattern):
            occurrences.append(i - j)
            j = lps[j - 1]
        elif i < len(text) and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return occurrences

def compute_lps(pattern):
    lps = [0] * len(pattern)
    length = 0
    i = 1

    while i < len(pattern):
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1

    return lps

# Example
text = "ababcababcabc"
pattern = "abc"
print(KMP_search(pattern, text))  # Output: [2, 7, 12]
```

7. String Rotation

Problem: Check if one string is a rotation of another string.

Algorithm:

- Concatenate the original string with itself and check if the other string is a substring.

Code:

```
python
Copy code
def is_rotation(s1, s2):
    if len(s1) != len(s2):
        return False
    return s2 in (s1 + s1)

# Example
s1 = "waterbottle"
s2 = "erbottlewat"
print(is_rotation(s1, s2))  # Output: True
```

8. Substring Search (Rabin-Karp Algorithm)

Problem: Search for a substring in a string using hashing.

Algorithm:

- Compute the hash of the pattern and the hash of the substrings of the same length in the text.

Code:

```
python
Copy code
def rabin_karp(pattern, text):
    m, n = len(pattern), len(text)
    d = 256  # Number of characters in the input alphabet
    q = 101  # A prime number
    p = 0  # Hash value for pattern
    t = 0  # Hash value for text
    h = 1

    for i in range(m - 1):
        h = (h * d) % q

    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    for i in range(n - m + 1):
        if p == t:
            if text[i:i + m] == pattern:
                print(f"Pattern found at index {i}")

        if i < n - m:
            t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) % q
            if t < 0:
                t += q

# Example
text = "abcxabcdabcdabcy"
pattern = "abcdabcy"
rabin_karp(pattern, text)  # Output: Pattern found at index 15
```

9. Count Distinct Substrings

Problem: Count the number of distinct substrings in a string.

Algorithm:

- Use a Suffix Array or a Trie to store and count unique substrings.

Code:

```
python
Copy code
def count_distinct_substrings(s):
    n = len(s)
    suffixes = sorted(s[i:] for i in range(n))
    distinct_count = n  # All suffixes are distinct

    for i in range(1, n):
        # Compare with previous suffix
        common_length = 0
        while common_length < len(suffixes[i]) and common_length < len(suffixes[i - 1]) and suffixes[i][common_length] ==
suffixes[i - 1][common_length]:
            common_length += 1
        distinct_count += len(suffixes[i]) - common_length

    return distinct_count

# Example
s = "ababa"
print(count_distinct_substrings(s))  # Output: 9
```

10. Longest Repeating Substring

Problem: Find the longest substring that occurs more than once in the string.

Algorithm:

- Use binary search combined with a rolling hash technique to find the longest repeating substring.

Code:

```
python
Copy code
def longest_repeating_substring(s):
    def search(length):
        seen = set()
        for i in range(len(s) - length + 1):
            substring = s[i:i + length]
            if substring in seen:
                return True
            seen.add(substring)
        return False

    left, right = 1, len(s)
    longest = ""

    while left <= right:
        mid = (left + right) // 2
        if search(mid):
            longest = mid
            left = mid + 1
        else:
            right = mid - 1

    return longest

# Example
s = "banana"
print(longest_repeating_substring(s))  # Output: "an"
```

Dictionary and Dictionary Manipulation

Basics of Dictionaries

- **Definition**: A dictionary is an unordered collection of items that are stored as key-value pairs. Each key must be unique and immutable (strings, numbers, or tuples).
- **Syntax**: A dictionary is created using curly braces `{}` or the `dict()` constructor.

Creating a Dictionary

```
python
Copy code
# Using curly braces
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Using the dict() constructor
my_dict = dict(name="Alice", age=25, city="New York")
```

Common Dictionary Manipulation Methods

Method	Description	Example
`len(d)`	Returns the number of key-value pairs in the dictionary.	`len(my_dict)` → 3
`d[key]`	Accesses the value associated with the specified key.	`my_dict["name"]` → `"Alice"`
`d[key] = value`	Adds a new key-value pair or updates the value if the key exists.	`my_dict["age"] = 26`
`del d[key]`	Deletes the key-value pair associated with the specified key.	`del my_dict["city"]`
`d.get(key)`	Returns the value for the specified key, or `None` if the key does not exist.	`my_dict.get("age")` → `25`
`d.keys()`	Returns a view object displaying a list of all keys in the dictionary.	`my_dict.keys()` → `dict_keys(['name', 'age'])`
`d.values()`	Returns a view object displaying a list of all values in the dictionary.	`my_dict.values()` → `dict_values(['Alice', 26])`
`d.items()`	Returns a view object displaying a list of key-value pairs as tuples.	`my_dict.items()` → `dict_items([('name', 'Alice'), ('age', 26)])`
`d.pop(key)`	Removes the specified key and returns its value. Raises a `KeyError` if the key is not found.	`my_dict.pop("age")` → `26`
`d.popitem()`	Removes and returns the last inserted key-value pair as a tuple.	`my_dict.popitem()` → `('city', 'New York')`
`d.clear()`	Removes all items from the dictionary.	`my_dict.clear()`
`d.update(other)`	Updates the dictionary with elements from another dictionary or an iterable of key-value pairs.	`my_dict.update({"age": 30})`

`d.copy()`	Returns a shallow copy of the dictionary.	`new_dict = my_dict.copy()`

Example of Dictionary Manipulation

```
python
Copy code
# Creating a dictionary
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing a value
print(person["name"])  # Output: Alice

# Updating a value
person["age"] = 26

# Adding a new key-value pair
person["job"] = "Engineer"

# Deleting a key-value pair
del person["city"]

# Using get() method
print(person.get("age"))  # Output: 26
print(person.get("city", "Not found"))  # Output: Not found

# Iterating over keys, values, and items
for key in person.keys():
    print(key)  # Output: name, age, job

for value in person.values():
    print(value)  # Output: Alice, 26, Engineer

for item in person.items():
    print(item)  # Output: ('name', 'Alice'), ('age', 26), ('job', 'Engineer')

# Copying a dictionary
new_person = person.copy()
print(new_person)  # Output: {'name': 'Alice', 'age': 26, 'job': 'Engineer'}
```

Use Cases in Competitive Programming

- **Counting Frequencies**: Dictionaries are commonly used to count the occurrences of elements in a list or string.

  ```
  python
  Copy code
  def count_frequencies(arr):
      frequency = {}
      for item in arr:
          frequency[item] = frequency.get(item, 0) + 1
      return frequency

  # Example
  arr = [1, 2, 2, 3, 3, 3]
  print(count_frequencies(arr))  # Output: {1: 1, 2: 2, 3: 3}
  ```

- **Grouping Items**: You can group items based on a certain key.

  ```
  python
  Copy code
  def group_by_first_letter(words):
      grouped = {}
      for word in words:
          first_letter = word[0]
          if first_letter not in grouped:
              grouped[first_letter] = []
          grouped[first_letter].append(word)
      return grouped

  # Example
  words = ["apple", "banana", "avocado", "blueberry", "cherry"]
  print(group_by_first_letter(words))  # Output: {'a': ['apple', 'avocado'], 'b': ['banana', 'blueberry'], 'c': ['cherr
  ```

```
y']}
```

- **Storing State or Configuration**: Dictionaries are useful for storing state or configuration settings, allowing easy access and modification.

Important Problems For Reference

Library/Module	Class/Function	Description	Example Usage
`collections`	`Counter`	A subclass of `dict` for counting hashable objects. It returns a dictionary with elements as keys and their counts as values.	`Counter(['a', 'b', 'a', 'c'])` → `Counter({'a': 2, 'b': 1, 'c': 1})`
		Used in competitive programming to count frequencies of elements efficiently.	`from collections import Counter`
`heapq`	`nlargest()`	Returns the n largest elements from the dataset defined by the iterable.	`heapq.nlargest(2, [1, 3, 5, 2, 4])` → `[5, 4]`
		Helpful for problems requiring the top k elements or priorities.	`import heapq`
`dict`	`setdefault()`	Returns the value of a key if it is in the dictionary; if not, it inserts the key with a specified default value.	`my_dict.setdefault(key, default)`
		Used to simplify dictionary operations and avoid checking key existence beforehand.	`my_dict = {}; my_dict.setdefault('key', 0)`

Usage Examples in Code

- **Counter Example**:

```
python
Copy code
from collections import Counter
count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'banana'])
print(count)  # Output: Counter({'banana': 3, 'apple': 2, 'orange': 1})
```

- **nlargest Example**:

```
python
Copy code
import heapq
nums = [1, 3, 5, 2, 4]
largest_two = heapq.nlargest(2, nums)
print(largest_two)  # Output: [5, 4]
```

- **setdefault Example**:

```
python
Copy code
my_dict = {}
my_dict.setdefault('a', 0)
my_dict['a'] += 1
print(my_dict)  # Output: {'a': 1}
```

1. Counting Frequencies

Problem: Count the frequency of elements in a list.

Algorithm:

- Use a dictionary to store the count of each element.

Code:

```
python
Copy code
def count_frequencies(arr):
    frequency = {}
    for item in arr:
        frequency[item] = frequency.get(item, 0) + 1
    return frequency

# Example
arr = [1, 2, 2, 3, 3, 3]
print(count_frequencies(arr))  # Output: {1: 1, 2: 2, 3: 3}
```

---

2. Group Anagrams

Problem: Group a list of strings into anagrams.

Algorithm:

- Use a dictionary where the key is the sorted tuple of characters in the string.

Code:

```
python
Copy code
def group_anagrams(words):
    anagrams = {}
    for word in words:
        key = tuple(sorted(word))
        anagrams.setdefault(key, []).append(word)
    return list(anagrams.values())

# Example
words = ["eat", "tea", "tan", "ate", "nat", "bat"]
print(group_anagrams(words))  # Output: [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
```

3. Two Sum Problem

Problem: Given an array of integers, find two numbers that add up to a specific target.

Algorithm:

- Use a dictionary to store the difference needed to reach the target.

Code:

```
python
Copy code
def two_sum(nums, target):
    num_dict = {}
    for index, num in enumerate(nums):
        difference = target - num
        if difference in num_dict:
            return [num_dict[difference], index]
        num_dict[num] = index
    return []

# Example
nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))  # Output: [0, 1]
```

4. Longest Substring Without Repeating Characters

Problem: Find the length of the longest substring without repeating characters.

Algorithm:

- Use a sliding window and a dictionary to track characters and their indices.

Code:

```
python
Copy code
def longest_substring(s):
    char_index = {}
    left = max_length = 0

    for right in range(len(s)):
        if s[right] in char_index:
            left = max(left, char_index[s[right]] + 1)
        char_index[s[right]] = right
        max_length = max(max_length, right - left + 1)

    return max_length

# Example
s = "abcabcbb"
print(longest_substring(s))  # Output: 3 (substring "abc")
```

5. Top K Frequent Elements

Problem: Given a non-empty array of integers, return the k most frequent elements.

Algorithm:

- Count frequencies using a dictionary, then use sorting or a heap to find the top k.

Code:

```
python
Copy code
from collections import Counter
import heapq

def top_k_frequent(nums, k):
    count = Counter(nums)
    return heapq.nlargest(k, count.keys(), key=count.get)

# Example
nums = [1, 1, 1, 2, 2, 3]
k = 2
print(top_k_frequent(nums, k))  # Output: [1, 2]
```

---

6. Constructing a Dictionary from a List of Pairs

Problem: Given a list of key-value pairs, construct a dictionary.

Algorithm:

- Use a dictionary comprehension to construct the dictionary.

Code:

```
python
Copy code
def construct_dict(pairs):
    return {key: value for key, value in pairs}

# Example
pairs = [("name", "Alice"), ("age", 25), ("city", "New York")]
print(construct_dict(pairs))  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

---

7. Find All Pairs with a Given Sum

Problem: Find all pairs in an array that sum up to a specific target.

Algorithm:

- Use a dictionary to store seen elements and their counts.

Code:

```
python
Copy code
def find_pairs(nums, target):
    seen = {}
    pairs = []

    for num in nums:
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen[num] = seen.get(num, 0) + 1

    return pairs

# Example
nums = [1, 2, 3, 4, 3, 5]
target = 6
print(find_pairs(nums, target))  # Output: [(3, 3), (2, 4)]
```

---

8. Check for Subarray Sum

Problem: Given an array, determine if a subarray sums to a given value.

Algorithm:

- Use a dictionary to store cumulative sums and check for the target sum.

Code:

```
python
Copy code
def check_subarray_sum(nums, k):
    cumulative_sum = 0
    sum_dict = {0: -1}  # Base case for sum of 0 at index -1
    for index, num in enumerate(nums):
        cumulative_sum += num
        if k != 0:
            cumulative_sum %= k  # Modulo to avoid large sums
        if cumulative_sum in sum_dict:
            if index - sum_dict[cumulative_sum] > 1:  # Check for at least 2 elements
                return True
        else:
            sum_dict[cumulative_sum] = index

    return False

# Example
nums = [23, 2, 4, 6, 7]
k = 6
print(check_subarray_sum(nums, k))  # Output: True (subarray [2, 4] sums to 6)
```

9. Finding Missing Numbers

Problem: Find the missing number in an array of numbers from 1 to n.

Algorithm:

- Use a dictionary to track the presence of each number.

Code:

```
python
Copy code
def find_missing(nums, n):
    num_dict = {num: True for num in nums}
    for i in range(1, n + 1):
        if i not in num_dict:
            return i
    return None

# Example
nums = [3, 7, 1, 2, 8]
n = 8
print(find_missing(nums, n))  # Output: 4
```

10. Finding Duplicate Elements

Problem: Find all duplicates in an array.

Algorithm:

- Use a dictionary to count occurrences and return those with a count greater than one.

Code:

```
python
Copy code
def find_duplicates(nums):
    count = {}
    duplicates = []

    for num in nums:
        count[num] = count.get(num, 0) + 1

    for num, freq in count.items():
        if freq > 1:
            duplicates.append(num)

    return duplicates

# Example
nums = [4, 3, 2, 7, 8, 3, 2, 1]
print(find_duplicates(nums))  # Output: [3, 2]
```

Basics of Tuples

- **Definition**: A tuple is an ordered collection of items, similar to a list, but it is immutable (cannot be modified after creation).
- **Syntax**: Tuples are created using parentheses `()` or the `tuple()` constructor.

Creating a Tuple

```
python
Copy code
# Using parentheses
my_tuple = (1, 2, 3)

# Using the tuple() constructor
my_tuple = tuple([1, 2, 3])
```

Common Tuple Manipulation Methods

Method	Description	Example
`len(t)`	Returns the number of elements in the tuple.	`len(my_tuple)` → 3
`t[index]`	Accesses the element at the specified index.	`my_tuple[0]` → `1`
`t.count(value)`	Returns the number of occurrences of a value.	`my_tuple.count(2)` → `1`
`t.index(value)`	Returns the index of the first occurrence of a value.	`my_tuple.index(2)` → `1`
`t + t2`	Concatenates two tuples.	`(1, 2) + (3, 4)` → `(1, 2, 3, 4)`
`t * n`	Repeats the tuple n times.	`(1, 2) * 3` → `(1, 2, 1, 2, 1, 2)`

Example of Tuple Manipulation

```
python
Copy code
# Creating a tuple
my_tuple = (1, 2, 3, 4)

# Accessing an element
print(my_tuple[1])  # Output: 2

# Counting occurrences
print(my_tuple.count(2))  # Output: 1

# Finding an index
print(my_tuple.index(3))  # Output: 2

# Concatenating tuples
new_tuple = my_tuple + (5, 6)
print(new_tuple)  # Output: (1, 2, 3, 4, 5, 6)

# Repeating a tuple
repeated_tuple = my_tuple * 2
print(repeated_tuple)  # Output: (1, 2, 3, 4, 1, 2, 3, 4)
```

Basics of Sets

- **Definition**: A set is an unordered collection of unique items. Sets are mutable, but they do not allow duplicate elements.
- **Syntax**: Sets are created using curly braces `{}` or the `set()` constructor.

Creating a Set

```
python
Copy code
# Using curly braces
my_set = {1, 2, 3}

# Using the set() constructor
my_set = set([1, 2, 3])
```

Common Set Manipulation Methods

Method	Description	Example
`len(s)`	Returns the number of elements in the set.	`len(my_set)` → 3
`s.add(value)`	Adds a new element to the set.	`my_set.add(4)`
`s.remove(value)`	Removes a specified element from the set. Raises `KeyError` if the element is not found.	`my_set.remove(2)`
`s.discard(value)`	Removes a specified element from the set if it exists. Does not raise an error if the element is not found.	`my_set.discard(5)`
`s.pop()`	Removes and returns an arbitrary element from the set. Raises `KeyError` if the set is empty.	`my_set.pop()`
`s.clear()`	Removes all elements from the set.	`my_set.clear()`
`s.union(s2)`	Returns a new set containing elements from both sets.	`my_set.union({3, 4, 5})`
`s.intersection(s2)`	Returns a new set containing elements common to both sets.	`my_set.intersection({2, 3, 4})`
`s.difference(s2)`	Returns a new set containing elements in the first set but not in the second.	`my_set.difference({2, 3})`

Example of Set Manipulation

```
python
Copy code
# Creating a set
my_set = {1, 2, 3, 4}

# Adding an element
my_set.add(5)
print(my_set)  # Output: {1, 2, 3, 4, 5}

# Removing an element
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4, 5}

# Discarding an element (no error if it doesn't exist)
my_set.discard(6)  # No error raised
print(my_set)  # Output: {1, 2, 4, 5}

# Finding the length of the set
print(len(my_set))  # Output: 4

# Set union
another_set = {4, 5, 6}
union_set = my_set.union(another_set)
print(union_set)  # Output: {1, 2, 4, 5, 6}

# Set intersection
intersection_set = my_set.intersection(another_set)
print(intersection_set)  # Output: {4, 5}

# Set difference
difference_set = my_set.difference(another_set)
print(difference_set)  # Output: {1, 2}
```

Methods for Tuples

Algorithm/Method	Description	Example Code
Finding Maximum and Minimum	Use `max()` and `min()` to find maximum and minimum values.	`max_value = max(my_tuple)`
Tuple to List Conversion	Convert a tuple to a list using `list()`.	`my_list = list(my_tuple)`
Nested Tuples	Create tuples of tuples for multidimensional data.	`nested_tuple = ((1, 2), (3, 4))`
Tuple Unpacking	Assign elements of a tuple to variables.	`a, b, c = my_tuple`
Sorting a Tuple	Use `sorted()` to sort elements in a tuple.	`sorted_tuple = tuple(sorted(my_tuple))`

Methods for Sets

Algorithm/Method	Description	Example Code
Set Intersection	Find common elements between two sets using `intersection()`.	`common_elements = set1.intersection(set2)`
Set Union	Combine unique elements from two sets using `union()` or `	`.
Finding Differences	Use `difference()` to find elements in one set not in another.	`diff_set = set1.difference(set2)`
Checking Subset and Superset	Use `issubset()` and `issuperset()` to check set relationships.	`is_subset = set1.issubset(set2)`
Removing Duplicates	Convert a list to a set to remove duplicates, then back to a list.	`unique_elements = list(set(my_list))`
Set Comprehensions	Create sets using concise set comprehensions.	`squares = {x**2 for x in range(5)}`
Finding Symmetric Difference	Use `symmetric_difference()` for elements in either set, but not both.	`sym_diff = set1.symmetric_difference(set2)`

Important Problems For Reference

Problems Involving Tuples

1. **Sum of Tuple Elements**
   - **Description**: Calculate the sum of all elements in a tuple.
   - **Code**:

   ```
   python
   Copy code
   def sum_of_tuple(t):
       return sum(t)

   # Example usage
   my_tuple = (1, 2, 3, 4, 5)
   print(sum_of_tuple(my_tuple))  # Output: 15
   ```

2. **Count Occurrences of Elements**

- **Description**: Count occurrences of each element in a tuple.
- **Code**:

  ```
  python
  Copy code
  from collections import Counter

  def count_occurrences(t):
      return dict(Counter(t))

  # Example usage
  my_tuple = (1, 2, 2, 3, 4, 4, 4)
  print(count_occurrences(my_tuple))  # Output: {1: 1, 2: 2, 3: 1, 4: 3}
  ```

3. **Find the Largest and Smallest Elements**
   - **Description**: Find the largest and smallest elements in a tuple.
   - **Code**:

     ```
     python
     Copy code
     def find_largest_smallest(t):
         return max(t), min(t)

     # Example usage
     my_tuple = (5, 1, 9, 3)
     print(find_largest_smallest(my_tuple))  # Output: (9, 1)
     ```

4. **Tuple Indexing**
   - **Description**: Return the index of an integer in a tuple; return -1 if not found.
   - **Code**:

     ```
     python
     Copy code
     def find_index(t, value):
         try:
             return t.index(value)
         except ValueError:
             return -1

     # Example usage
     my_tuple = (10, 20, 30)
     print(find_index(my_tuple, 20))  # Output: 1
     print(find_index(my_tuple, 40))  # Output: -1
     ```

5. **Flatten a Nested Tuple**
   - **Description**: Flatten a nested tuple into a single tuple.
   - **Code**:

     ```
     python
     Copy code
     def flatten_tuple(nested):
         flat = []
         for item in nested:
             if isinstance(item, tuple):
                 flat.extend(flatten_tuple(item))
             else:
                 flat.append(item)
         return tuple(flat)

     # Example usage
     nested_tuple = ((1, 2), (3, 4), (5, 6))
     print(flatten_tuple(nested_tuple))  # Output: (1, 2, 3, 4, 5, 6)
     ```

Problems Involving Sets

1. **Unique Elements Count**
   - **Description**: Count of unique elements in a list.
   - **Code**:

```
python
Copy code
def unique_count(lst):
    return len(set(lst))

# Example usage
my_list = [1, 2, 2, 3, 4, 4, 5]
print(unique_count(my_list))  # Output: 5
```

2. **Set Intersection**
   - **Description**: Find common elements between two lists.
   - **Code**:

```
python
Copy code
def set_intersection(list1, list2):
    return list(set(list1) & set(list2))

# Example usage
list1 = [1, 2, 3, 4]
list2 = [3, 4, 5, 6]
print(set_intersection(list1, list2))  # Output: [3, 4]
```

3. **Set Difference**
   - **Description**: Elements in the first set but not in the second.
   - **Code**:

```
python
Copy code
def set_difference(set1, set2):
    return set1.difference(set2)

# Example usage
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5}
print(set_difference(set1, set2))  # Output: {1, 2}
```

4. **Checking Subset and Superset**
   - **Description**: Check if one set is a subset of another.
   - **Code**:

```
python
Copy code
def check_subset(set1, set2):
    return set1.issubset(set2), set2.issuperset(set1)

# Example usage
set1 = {1, 2}
set2 = {1, 2, 3}
print(check_subset(set1, set2))  # Output: (True, True)
```

5. **Finding Symmetric Difference**
   - **Description**: Elements that are in either set, but not both.
   - **Code**:

```
python
Copy code
def symmetric_difference(set1, set2):
    return set1.symmetric_difference(set2)

# Example usage
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(symmetric_difference(set1, set2))  # Output: {1, 2, 4, 5}
```

6. **Union of Multiple Sets**
   - **Description**: Return a single set containing all unique elements from multiple sets.

- **Code**:

```
python
Copy code
def union_of_sets(sets):
    result_set = set()
    for s in sets:
        result_set = result_set.union(s)
    return result_set

# Example usage
sets = [{1, 2}, {2, 3}, {3, 4}]
print(union_of_sets(sets))  # Output: {1, 2, 3, 4}
```

Libraries Useful for Competitive Programming

1. Math Library (`math`)

This library provides access to mathematical functions.

Method/Function	Description	Example Code
`math.sqrt(x)`	Returns the square root of `x`.	`math.sqrt(16)` # Output: 4.0
`math.factorial(x)`	Returns the factorial of `x`.	`math.factorial(5)` # Output: 120
`math.gcd(a, b)`	Returns the greatest common divisor of `a` and `b`.	`math.gcd(12, 15)` # Output: 3
`math.comb(n, k)`	Returns the number of ways to choose `k` items from `n` items.	`math.comb(5, 2)` # Output: 10
`math.perm(n, k)`	Returns the number of ways to choose `k` items from `n` items with order.	`math.perm(5, 2)` # Output: 20

2. Collections Library (`collections`)

This library implements specialized container datatypes.

Class/Method	Description	Example Code
`Counter`	Counts hashable objects.	`Counter([1, 2, 2, 3])` # Output: Counter({2: 2, 1: 1, 3: 1})
`defaultdict`	A dictionary that provides a default value for non-existing keys.	`d = defaultdict(int)`; `d['a'] += 1` # d['a'] = 1
`deque`	A double-ended queue for fast appends and pops.	`d = deque([1, 2, 3]); d.append(4); d.popleft()` # Output: 2
`OrderedDict`	A dictionary that maintains insertion order.	`OrderedDict([(1, 'a'), (2, 'b')])`

3. Heap Queue Library (`heapq`)

This library implements a min-heap queue algorithm.

Method/Function	Description	Example Code
`heapq.heappush(heap, item)`	Pushes an item onto the heap.	`h = []`; `heapq.heappush(h, 5)`
`heapq.heappop(heap)`	Pops the smallest item from the heap.	`smallest = heapq.heappop(h)`
`heapq.nlargest(n, iterable)`	Returns the `n` largest elements from the dataset.	`heapq.nlargest(3, [1, 4, 2, 3])` # Output: [4, 3, 2]
`heapq.nsmallest(n, iterable)`	Returns the `n` smallest elements from the dataset.	`heapq.nsmallest(2, [1, 4, 2, 3])` # Output: [1, 2]

4. Itertools Library (`itertools`)

This library provides functions that create iterators for efficient looping.

Method/Function	Description	Example Code
`itertools.permutations(iterable, r=None)`	Returns all possible permutations of a given iterable.	`list(itertools.permutations([1, 2]))` # Output: [(1, 2), (2, 1)]
`itertools.combinations(iterable, r)`	Returns all possible combinations of a given length.	`list(itertools.combinations([1, 2, 3], 2))` # Output: [(1, 2), (1, 3), (2, 3)]
`itertools.product(*iterables)`	Returns the Cartesian product of input iterables.	`list(itertools.product([1, 2], ['a', 'b']))` # Output: [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
`itertools.chain(*iterables)`	Combines multiple iterables into a single iterable.	`list(itertools.chain([1, 2], ['a', 'b']))` # Output: [1, 2, 'a', 'b']

5. Functools Library (`functools`)

This library provides higher-order functions for functional programming.

Method/Function	Description	Example Code
`functools.reduce(function, iterable[, initializer])`	Applies a rolling computation to sequential pairs of values in `iterable`.	`from functools import reduce; reduce(lambda x, y: x + y, [1, 2, 3, 4])` # Output: 10
`functools.lru_cache(maxsize=None)`	Decorator to cache results of function calls.	`@lru_cache; def fib(n): return n if n < 2 else fib(n-1) + fib(n-2)`

6. Random Library (`random`)

This library implements pseudo-random number generators.

Method/Function	Description	Example Code

`random.randint(a, b)`	Returns a random integer N such that `a <= N <= b`.	`random.randint(1, 10)`
`random.choice(seq)`	Returns a random element from a non-empty sequence.	`random.choice(['a', 'b', 'c'])`
`random.sample(population, k)`	Returns a `k` length list of unique elements from the population.	`random.sample([1, 2, 3, 4], 2)`
`random.shuffle(x)`	Shuffles the elements of a list in place.	`random.shuffle(my_list)`

7. NumPy Library

NumPy is a powerful library for numerical computations.

Method/Function	Description	Example Code
`numpy.array()`	Creates an array from a list.	`np.array([1, 2, 3])`
`numpy.zeros()`	Creates an array filled with zeros.	`np.zeros((3, 3))` # Output: 3×3 matrix of zeros
`numpy.ones()`	Creates an array filled with ones.	`np.ones((2, 2))` # Output: 2×2 matrix of ones
`numpy.mean()`	Returns the mean of the array.	`np.mean(np.array([1, 2, 3]))` # Output: 2.0

8. Pandas Library

Pandas is used for data manipulation and analysis.

Method/Function	Description	Example Code
`pandas.DataFrame()`	Creates a DataFrame from a dictionary or list.	`pd.DataFrame({'A': [1, 2], 'B': [3, 4]})`
`df.head(n)`	Returns the first `n` rows of the DataFrame.	`df.head(3)` # Returns first 3 rows
`df.describe()`	Generates descriptive statistics.	`df.describe()`
`df.groupby(column)`	Groups the DataFrame using a column.	`df.groupby('A').sum()`

Basic Common Algorithms:

1. Sorting Algorithms

1.1 Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

```
python
Copy code
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(arr))  # Output: [11, 12, 22, 25, 34, 64, 90]
```

1.2 Quick Sort

Quick Sort is a divide-and-conquer algorithm. It picks an element as a pivot and partitions the array around the pivot.

```
python
Copy code
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print(quick_sort(arr))  # Output: [11, 12, 22, 25, 34, 64, 90]
```

2. Searching Algorithms

2.1 Linear Search

Linear Search is a simple search algorithm that checks each element in the list until it finds the target.

```
python
Copy code
```

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1

# Example Usage
arr = [5, 3, 8, 4, 2]
print(linear_search(arr, 4))  # Output: 3
```

2.2 Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted list. It repeatedly divides the search interval in half.

```
python
Copy code
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Example Usage
arr = [1, 2, 3, 4, 5, 6]
print(binary_search(arr, 4))  # Output: 3
```

3. Dynamic Programming

3.1 Fibonacci Sequence

A classic example of dynamic programming is calculating the Fibonacci sequence. We can use memoization to optimize the recursive solution.

```
python
Copy code
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]

# Example Usage
print(fibonacci(10))  # Output: 55
```

4. Graph Algorithms

4.1 Depth-First Search (DFS)

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

```
python
Copy code
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example Usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
dfs(graph, 'A')  # Output: A B D E F C
```

4.2 Breadth-First Search (BFS)

BFS is a graph traversal algorithm that explores all the neighbor nodes at the present depth prior to moving on to nodes at the next depth level.

```
python
Copy code
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example Usage
bfs(graph, 'A')  # Output: A B C D E F
```

Advanced Algorithms

Greedy Algorithms

1.1 Coin Change Problem

This problem involves finding the minimum number of coins needed to make a given amount using the denominations available.

```
python
Copy code
def coin_change(coins, amount):
    coins.sort(reverse=True)  # Sort coins in descending order
    count = 0
    for coin in coins:
        while amount >= coin:
            amount -= coin
            count += 1
    return count if amount == 0 else -1

# Example Usage
coins = [1, 5, 10, 25]
amount = 37
print(coin_change(coins, amount))  # Output: 4 (1x25 + 1x10 + 2x1)
```

2. Backtracking Algorithms

2.1 N-Queens Problem

This classic problem involves placing N queens on an N×N chessboard so that no two queens threaten each other.

```
python
Copy code
def print_solution(board):
    for row in board:
        print(" ".join(row))
    print()

def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 'Q':
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False

    for i, j in zip(range(row, len(board)), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False

    return True

def solve_n_queens_util(board, col):
    if col >= len(board):
        print_solution(board)
        return True

    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 'Q'
            solve_n_queens_util(board, col + 1)
            board[i][col] = '.'  # Backtrack
    return False

def solve_n_queens(n):
    board = [['.' for _ in range(n)] for _ in range(n)]
    solve_n_queens_util(board, 0)

# Example Usage
solve_n_queens(4)  # Prints all solutions for 4 queens
```

3. Searching Algorithms

3.1 Ternary Search

Ternary Search is a divide-and-conquer algorithm that can be used to find the maximum or minimum of a unimodal function.

```
python
Copy code
def ternary_search(left, right, func):
    if right - left < 1e-5:  # Precision threshold
        return func(left)

    mid1 = left + (right - left) / 3
    mid2 = right - (right - left) / 3

    if func(mid1) < func(mid2):
        return ternary_search(mid1, right, func)
    else:
        return ternary_search(left, mid2, func)

# Example Usage
def function(x):
    return -(x ** 2) + 4 * x + 1  # Example function to maximize

max_value = ternary_search(0, 5, function)
print(max_value)  # Output: Maximum value of the function
```

4. Dynamic Programming

4.1 0/1 Knapsack Problem

This problem is about maximizing the value of items that can be carried in a knapsack with a weight limit.

```
python
Copy code
def knapsack(weights, values, capacity):
    n = len(values)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(n + 1):
        for w in range(capacity + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]

# Example Usage
weights = [1, 2, 3]
```

```
values = [10, 15, 40]
capacity = 6
print(knapsack(weights, values, capacity))  # Output: 55
```

5. Graph Algorithms

5.1 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph.

```
python
Copy code
import heapq

def dijkstra(graph, start):
    queue = []
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    heapq.heappush(queue, (0, start))

    while queue:
        current_distance, current_vertex = heapq.heappop(queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances

# Example Usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
print(dijkstra(graph, 'A'))  # Output: Shortest distances from A
```

6. String Algorithms

6.1 KMP (Knuth-Morris-Pratt) Algorithm

The KMP algorithm is used for substring search within a string. It preprocesses the pattern to create a partial match table.

```
python
Copy code
def kmp_search(pattern, text):
    m, n = len(pattern), len(text)
    lps = [0] * m
    j = 0

    # Preprocess the pattern
    i = 1
    while i < m:
        if pattern[i] == pattern[j]:
            j += 1
            lps[i] = j
            i += 1
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                lps[i] = 0
                i += 1

    # Search for the pattern in the text
    i, j = 0, 0
    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == m:
            print(f"Pattern found at index {i - j}")
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

# Example Usage
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
kmp_search(pattern, text)  # Output: Pattern found at index 10
```

Bitwise Operation Problems

Counting Set Bits

Problem: Given an integer, count the number of set bits (1s) in its binary representation.

Example:
Input:
`7` (binary `111`)

Output: `3`

```
python
Copy code
def count_set_bits(n):
    count = 0
    while n:
        count += n & 1
        n >>= 1
    return count

# Example Usage
print(count_set_bits(7))  # Output: 3
```

2. Checking if a Number is Power of Two

Problem: Determine if a given integer is a power of two.

Example:
Input:
`8` (binary `1000`)

Output: `True`

Input: `10`

Output: `False`

```
python
Copy code
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0

# Example Usage
print(is_power_of_two(8))   # Output: True
print(is_power_of_two(10))  # Output: False
```

3. Swapping Two Numbers Using XOR

Problem: Swap two numbers without using a temporary variable.

Example:
Input:
`a = 5, b = 10`

Output: `a = 10, b = 5`

```
python
Copy code
def swap(a, b):
    a = a ^ b
    b = a ^ b
    a = a ^ b
    return a, b
```

```
# Example Usage
a, b = 5, 10
a, b = swap(a, b)
print(a, b)  # Output: 10 5
```

4. Finding the Only Non-Repeating Element

Problem: Given an array where every element appears twice except for one, find that single element.

Example:
Input:
`[2, 2, 1, 4, 4]`

Output: `1`

```
python
Copy code
def find_non_repeating(arr):
    result = 0
    for num in arr:
        result ^= num
    return result

# Example Usage
print(find_non_repeating([2, 2, 1, 4, 4]))  # Output: 1
```

5. Finding the Missing Number in an Array

Problem: Given an array containing `n` distinct numbers taken from `0` to `n`, find the missing number.

Example:
Input:
`[3, 0, 1]`

Output: `2`

```
python
Copy code
def missing_number(arr):
    n = len(arr)
    total = n * (n + 1) // 2
    sum_of_array = sum(arr)
    return total - sum_of_array

# Example Usage
print(missing_number([3, 0, 1]))  # Output: 2
```

6. Reversing Bits

Problem: Reverse the bits of a given 32-bit unsigned integer.

Example:
Input:
`43261596` (binary `00000010100101000001111010011100`)

Output: `964176192` (binary `00111001011110000010100101000000`)

```
python
Copy code
def reverse_bits(n):
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result

# Example Usage
print(reverse_bits(43261596))  # Output: 964176192
```

7. Detecting a Cycle in a Bitmask

Problem: Given a bitmask representing a graph, check if there is a cycle.

Example:
Input:
`0b101011`

Output: Depends on the graph structure represented by the bitmask.

```
python
Copy code
def has_cycle(bitmask):
    # This is a simple example of detecting cycle using a naive approach
    n = len(bin(bitmask)) - 2  # Exclude '0b'
    visited = [False] * n
    stack = []

    for i in range(n):
        if bitmask & (1 << i):
            if visited[i]:
                return True
            visited[i] = True
            stack.append(i)

    return False

# Example Usage
print(has_cycle(0b101011))  # Example output: Depends on the graph structure
```

8. Finding the Two Non-Repeating Elements

Problem: Given an array where every element appears three times except for two, find those two elements.

Example:
Input:
`[2, 2, 3, 2]`

Output: `3`

```
python
Copy code
def find_two_non_repeating(arr):
    xor_all = 0
    for num in arr:
        xor_all ^= num

    # Get the rightmost set bit
    rightmost_set_bit = xor_all & -xor_all

    first = 0
    second = 0

    for num in arr:
        if num & rightmost_set_bit:
            first ^= num
        else:
            second ^= num

    return first, second

# Example Usage
print(find_two_non_repeating([2, 2, 3, 2]))  # Output: (3)
```

Basics of Linked Lists in Python

A linked list is a linear data structure where each element (node) contains a value and a reference (or pointer) to the next node in the sequence. This allows for efficient insertion and deletion of elements compared to arrays, which require shifting elements.

Structure of a Linked List Node

In Python, a linked list node can be represented using a class. Each node has two attributes:

1. **Value**: The data stored in the node.
2. **Next**: A reference to the next node in the list.

Implementation of a Linked List

Here's how you can implement a basic singly linked list in Python:

```
python
Copy code
class Node:
    def __init__(self, value):
        self.value = value  # Initialize the value of the node
        self.next = None    # Initialize the next reference to None

class LinkedList:
    def __init__(self):
        self.head = None  # Initialize the head of the linked list

    def append(self, value):
        new_node = Node(value)  # Create a new node
        if not self.head:       # If the list is empty
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:   # Traverse to the end of the list
            last_node = last_node.next
        last_node.next = new_node  # Link the new node

    def display(self):
        current_node = self.head  # Start from the head
        while current_node:        # Traverse the linked list
            print(current_node.value, end=" -> ")
            current_node = current_node.next
        print("None")  # Indicate the end of the list

# Example Usage
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)

# Display the linked list
linked_list.display()
```

Explanation of the Code

1. **Node Class**:
   - The `Node` class represents each node in the linked list. It has a constructor that initializes the `value` and `next` attributes.
2. **LinkedList Class**:
   - The `LinkedList` class manages the linked list. It has:
     - A constructor that initializes the `head` to `None`.
     - An `append` method to add new nodes to the end of the list.
     - A `display` method to print the values in the linked list.
3. **Example Usage**:
   - We create an instance of `LinkedList`, append three values (1, 2, 3), and then display the linked list.

Output

When you run the example, the output will be:

```
rust
Copy code
1 -> 2 -> 3 -> None
```