1. How to reverse a list in Python?

You can reverse a list in Python using multiple methods:

- **Using `reverse()` method**: This method reverses the list in place.

```
python

my_list = [1, 2, 3, 4, 5]
my_list.reverse()
print(my_list)  # Output: [5, 4, 3, 2, 1]
```

- **Using slicing**: You can create a new reversed list using slicing.

```
python

my_list = [1, 2, 3, 4, 5]
reversed_list = my_list[::-1]
print(reversed_list)  # Output: [5, 4, 3, 2, 1]
```

- **Using `reversed()` function**: This function returns an iterator that can be converted to a list.

```
python

my_list = [1, 2, 3, 4, 5]
reversed_list =
list(reversed(my_list))
print(reversed_list)  # Output: [5, 4, 3, 2, 1]
```

2. What are the data types available in Python, and why are they called immutable and mutable?

- **Data Types**:
  - **Numeric Types**: `int`, `float`, `complex`
  - **Sequence Types**: `str`, `list`, `tuple`
  - **Mapping Type**: `dict`
  - **Set Types**: `set`, `frozenset`
  - **Boolean Type**: `bool`
  - **Binary Types**: `bytes`, `bytearray`, `memoryview`
  - **None Type**: `NoneType`
- **Immutable**: Data types like `int`, `float`, `str`, `tuple`, and `frozenset` are immutable, meaning their values cannot be changed after they are created. If you try to modify them, a new object is created.
- **Mutable**: Data types like `list`, `dict`, `set`, and `bytearray` are mutable, meaning their values can be changed in place without creating a new object.

3. What could be the key of a `dict` and why?

A key in a Python dictionary must be immutable and hashable. This is because the dictionary uses a hash table to manage its entries. The most common immutable types used as keys are `int`, `float`, `str`, and `tuple` (if the tuple only contains immutable elements).

4. What is a static method and class method? What are the differences?

- **Static Method**:
  - Defined using the `@staticmethod` decorator.
  - Does not receive an implicit first argument.
  - Can be called on the class or an instance, but it doesn't have access to the instance (`self`) or class (`cls`).

```
python

class MyClass:
    @staticmethod
    def static_method():
        print("I am a static
method")
```

- **Class Method**:
  - Defined using the `@classmethod` decorator.
  - Takes `cls` as its first argument, which is the class itself.
  - Can modify class state that applies across all instances.

```
python

class MyClass:
    @classmethod
    def class_method(cls):
        print("I am a class
method")
```

5. What is a metaclass?

A metaclass in Python is a class of a class that defines how a class behaves. Just as a class defines the behavior of instances, a metaclass defines the behavior of classes themselves. Metaclasses allow customization of class creation.

6. What is monkey patching?

Monkey patching refers to modifying or extending a module or class at runtime. It is generally used to dynamically modify the behavior of a class or module.

Example:

```
python

class MyClass:
    def original_method(self):
        print("Original method")

def patched_method(self):
    print("Patched method")

MyClass.original_method = patched_method

obj = MyClass()
obj.original_method()  # Output: Patched
method
```

7. Memory management in Python

- **Garbage Collection (GC)**: Python uses automatic memory management that includes a garbage collector to free up memory when objects are no longer in use. Python primarily uses reference counting and a cyclic garbage collector for detecting and collecting circular references.
- **Stack Memory**: Used for storing local variables and function calls. This memory is managed automatically and is typically faster.
- **Heap Memory**: Used for dynamic memory allocation, where variables and objects are stored. Heap memory is managed by the Python memory manager and garbage collector.

8. Explain `filter()`, `map()`, `reduce()`

- **`filter()`**: Filters elements of an iterable based on a function that returns either `True` or `False`.

  ```
  python

  nums = [1, 2, 3, 4, 5]
  even_nums = list(filter(lambda x: x
  % 2 == 0, nums))
  print(even_nums)  # Output: [2, 4]
  ```

- **`map()`**: Applies a function to all items in an iterable and returns a map object (iterator).

  ```
  python

  nums = [1, 2, 3, 4, 5]
  squared_nums = list(map(lambda x: x
  ** 2, nums))
  print(squared_nums)  # Output: [1,
  4, 9, 16, 25]
  ```

- **`reduce()`**: Reduces an iterable to a single value by applying a function cumulatively to the items (requires `functools` module).

  ```
  python

  from functools import reduce
  nums = [1, 2, 3, 4, 5]
  sum_nums = reduce(lambda x, y: x +
  y, nums)
  print(sum_nums)  # Output: 15
  ```

9. What is MRO (Method Resolution Order)?

MRO is the order in which Python looks for a method in a hierarchy of classes. It's determined by the C3 linearization algorithm, which is used in Python's new-style classes (derived from `object`). You can view MRO using the `__mro__` attribute or the `mro()` method.

```
python

class A:
    pass

class B(A):
    pass

class C(B):
    pass

print(C.__mro__)
```

10. Suppose there are three classes named `A`, `B`, and `C`, where `B` inherits `A` and `C` inherits `B`. What will print first if we create an object of `C`?

If you create an object of `C`, the constructor of `C` is called first. If `C` doesn't have a constructor, Python will look for a constructor in its superclass, `B`, and so on. This follows the MRO. Thus, the constructor in `A` will be executed first if `B` and `C` don't have their own constructors.

```
python

class A:
    def __init__(self):
        print("A")

class B(A):
    def __init__(self):
        super().__init__()
        print("B")

class C(B):
    def __init__(self):
        super().__init__()
        print("C")

obj = C()  # Output: A B C
```

1. Creating a List of Tuples from Two Lists

You can combine two lists into a list of tuples using the `zip()` function:

```
python

list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
output = list(zip(list1, list2))
print(output)  # Output: [('a', 1), ('b', 2), ('c', 3)]
```

2. List Comprehension with `if` and `else`

You can use `if-else` in a list comprehension like this:

```
python

numbers = [1, 2, 3, 4, 5]
output = ["even" if x % 2 == 0 else "odd" for x in numbers]
print(output)  # Output: ['odd', 'even', 'odd', 'even', 'odd']
```

3. Shallow Copy vs Deep Copy

- **Shallow Copy**: Creates a new object, but inserts references into it to the objects found in the original.

```
python

import copy
original = [1, 2, [3, 4]]
shallow_copy = copy.copy(original)
shallow_copy[2][0] = 99
print(original)  # Output: [1, 2, [99, 4]]
```

- **Deep Copy**: Creates a new object and recursively copies all objects found in the original.

```
python

deep_copy = copy.deepcopy(original)
deep_copy[2][0] = 100
print(original)  # Output: [1, 2, [99, 4]]
```

4. Scope in Python: LEGB Rule

- **L**ocal: Names defined within a function.
- **E**nclosing: Names in the local scope of any enclosing functions.
- **G**lobal: Names defined at the module level.
- **B**uilt-in: Names in Python's built-in namespace.

5. Importance of GIL (Global Interpreter Lock)

The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously in a multi-threaded environment. It ensures thread safety but can be a performance bottleneck in CPU-bound multi-threaded programs.

6. When to Use Multiprocessing and Multithreading

- **Multiprocessing**: Useful for CPU-bound tasks that require parallel execution across multiple cores.

- **Multithreading**: Better suited for I/O-bound tasks where the program spends a lot of time waiting for external resources.

7. Polymorphism in Python

Polymorphism allows methods in different classes to have the same name but behave differently.

```
python

class Cat:
    def sound(self):
        return "Meow"

class Dog:
    def sound(self):
        return "Woof"

animals = [Cat(), Dog()]
for animal in animals:
    print(animal.sound())
```

8. Context Manager

A context manager in Python manages resource allocation and deallocation. It is often used with the `with` statement to handle files, connections, etc.

```
python

with open('file.txt', 'w') as file:
    file.write('Hello, World!')
```

9. How to Order a Dictionary

To maintain the order of a dictionary, you can use `OrderedDict` from the `collections` module, though in Python 3.7+, dictionaries maintain insertion order by default.

```
python

from collections import OrderedDict

ordered_dict = OrderedDict([('a', 1),
('b', 2), ('c', 3)])
print(ordered_dict)
```

10. Is it Possible to Overload Operators in Python?

Yes, you can overload operators in Python by defining special methods in your class, like `__add__`, `__sub__`, etc.

```
python

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x,
self.y + other.y)

point1 = Point(1, 2)
point2 = Point(3, 4)
result = point1 + point2  # Uses __add__
```

11. `dir()` and `id()` in Python

- **`dir()`**: Returns a list of valid attributes for an object.
- **`id()`**: Returns the unique identifier of an object.

```
python

a = 10
print(dir(a))  # List of attributes
print(id(a))   # Memory address
```

12. What are Generators?

Generators are iterators that yield values one at a time, allowing for efficient memory usage.

```
python

def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

gen = count_up_to(5)
for num in gen:
    print(num)
```

13. What are Decorators?

Decorators are functions that modify the behavior of another function.

```
python

def my_decorator(func):
    def wrapper():
        print("Something before the
function")
        func()
```

```
        print("Something after the
function")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

14. Pickling and Unpickling

- **Pickling**: Serializing a Python object into a byte stream.
- **Unpickling**: Deserializing the byte stream back into a Python object.

```
python

import pickle

data = {'a': 1, 'b': 2}
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)

with open('data.pkl', 'rb') as file:
    data_loaded = pickle.load(file)
```

15. Converting List to a Sentence

You can join list elements into a string:

```
python

words = ['python', 'is', 'fun']
sentence = " ".join(words)
print(sentence)  # Output: "python is
fun"
```

16. Difference Between List and Tuple

- **List**: Mutable, can change after creation, slower.
- **Tuple**: Immutable, cannot change after creation, faster.

17. Print the Last Word of a Sentence

You can do this in a single line:

```
python

sentence = "Python is fun"
print(sentence.split()[-1])  # Output:
"fun"
```

18. Difference Between Python 2.x and 3.x

One major difference is the `print` statement in Python 2.x vs. the `print()` function in Python 3.x.

```
python

# Python 2.x
print "Hello, World!"

# Python 3.x
print("Hello, World!")
```

19. What is Namespace?

A namespace is a container where names are mapped to objects. Python uses namespaces to avoid name conflicts.

20. What are Dunder Methods?

Dunder methods (double underscore methods) are special methods in Python, such as `__init__`, `__str__`, `__add__`, that allow customization of class behavior.

21. Printing a Descriptive Message in a Class

You can override the `__str__` or `__repr__` method in a class to return a descriptive message:

```
python

class A:
    def __str__(self):
        return "I am A class"

obj_a = A()
print(obj_a)  # Output: "I am A class"
```

22. SOLID Principles

The SOLID principles are a set of design principles in object-oriented programming aimed at making software designs more understandable, flexible, and maintainable. Here's a resource for deeper exploration

1. Python Program to Find the Second Largest Number in a List

```
python
```

```
def second_largest(numbers):
    if len(numbers) < 2:
        return None
    first = second = float('-inf')
    for num in numbers:
        if num > first:
            first, second = num, first
        elif first > num > second:
            second = num
    return second

numbers = [10, 20, 4, 45, 99]
print("Second largest number is:",
second_largest(numbers))  # Output: 45
```

2. Python Program to Count the Frequency of Each Element in a List

```
python

from collections import Counter

def count_frequency(lst):
    return dict(Counter(lst))

lst = [1, 2, 2, 3, 4, 4, 4, 5]
frequency = count_frequency(lst)
print("Frequency of elements:",
frequency)
# Output: {1: 1, 2: 2, 3: 1, 4: 3, 5: 1}
```

3. Python Program to Find the Number of Palindromes in an Input String

```
python

def count_palindromes(s):
    words = s.split()
    return sum(1 for word in words if
word == word[::-1])

s = "madam arora teaches malayalam"
print("Number of palindromes:",
count_palindromes(s))  # Output: 3
```

4. Python Program to Check if the Given Strings Are Anagrams or Not

```
python

def are_anagrams(str1, str2):
    return sorted(str1) == sorted(str2)

str1 = "listen"
str2 = "silent"
print("Anagrams:" if are_anagrams(str1,
str2) else "Not Anagrams")  # Output:
Anagrams
```

5. Python Program to Print the Given Pattern

```
python

def print_pattern(n):
    for i in range(1, n + 1):
        print('*' * i)

print_pattern(6)
```

Output:

```
markdown

*
**
***
****
*****
******
```

6. Python Program to Print the Given Alphabet Pattern

```
python

def print_alphabet_pattern(rows):
    current_char = 65  # ASCII value of
'A'
    for i in range(1, rows + 1):
        for j in range(i):
            print(chr(current_char),
end="")
            current_char += 1
        print()

print_alphabet_pattern(5)
```

Output:

```
css

A
BC
DEF
GHIJ
KLMNO
```

7. Python Program to Find the Sum of Fibonacci Numbers

```
python

def fibonacci_sum(n):
    if n < 1:
        return 0
```

```
    a, b = 0, 1
    sum_fib = a + b
    for _ in range(2, n+1):
        a, b = b, a + b
        sum_fib += b
    return sum_fib

n = 4
print("Sum of first", n, "Fibonacci
numbers is:", fibonacci_sum(n))  #
Output: 7
```

8. Python Program to Find the Length of the Longest Substring Without Repeating Characters

```
python

def length_of_longest_substring(s):
    char_index = {}
    max_len = start = 0

    for i, char in enumerate(s):
        if char in char_index and start
<= char_index[char]:
            start = char_index[char] + 1
        else:
            max_len = max(max_len, i -
start + 1)
        char_index[char] = i

    return max_len

# Test Cases
s1 = "abcabcbb"
print("Length of the longest substring
without repeating characters:",
length_of_longest_substring(s1))  #
Output: 3

s2 = "bbbb"
print("Length of the longest substring
without repeating characters:",
length_of_longest_substring(s2))  #
Output: 1

s3 = "pwwkew"
print("Length of the longest substring
without repeating characters:",
length_of_longest_substring(s3))  #
Output: 3
```

To solve the problem of reversing a signed 32-bit integer while ensuring it doesn't exceed the bounds of a 32-bit signed integer, you can use the following approach:

Python Program to Reverse an Integer

```
python

def reverse_integer(x):
    # Define the bounds of a 32-bit
signed integer
    MIN_INT, MAX_INT = -2**31, 2**31 - 1

    # Determine if the number is negative
    sign = -1 if x < 0 else 1
    x *= sign

    # Reverse the digits of the integer
    reversed_x = 0
    while x:
        digit = x % 10
        x //= 10
        reversed_x = reversed_x * 10 +
digit

    # Apply the sign to the reversed
number
    reversed_x *= sign

    # Check for overflow
    if reversed_x < MIN_INT or reversed_x
> MAX_INT:
        return 0

    return reversed_x

# Test Cases
print(reverse_integer(123))   # Output:
321
print(reverse_integer(-123))  # Output: -
321
print(reverse_integer(120))   # Output:
21
print(reverse_integer(0))     # Output: 0
```

Explanation:

1. **Handling Sign**: First, the program checks if the integer `x` is negative and stores the sign. The absolute value of `x` is then considered for the reversal process.
2. **Reversing Digits**: The program iteratively extracts the last digit of `x` using the modulus operation and builds the reversed number. This is done by multiplying the current `reversed_x` by 10 and adding the last digit.
3. **Checking for Overflow**: After reversing the digits, the program checks whether the reversed number falls within the valid 32-bit signed integer range (`[-2^31, 2^31 - 1]`). If it doesn't, the function returns `0`.
4. **Returning the Result**: Finally, the reversed number is returned with the correct sign applied.

Problem Statement: Find Number of Palindromes in Input String

Input Examples:

1. **i/p1:** "Working in JPMC" ==> **o/p1:** "Palindrome not found"
2. **i/p2:** "MADAM reached RADAR office" ==> **o/p2:** "Two Palindromic strings found: MADAM, RADAR"

1. Approach to Solve the Problem

1. **Split the String**: Break the input string into individual words.
2. **Check Each Word**: For each word, check if it reads the same forwards and backwards.
3. **Store Palindromes**: Keep a list or counter for words that are palindromes.
4. **Return Result**: Based on the count, return the appropriate message.

2. Solution Using `for` Loop

```
python

def find_palindromes_for_loop(s):
    words = s.split()
    palindromes = []

    for word in words:
        if word.lower() == word[::-
1].lower():
            palindromes.append(word)

    if palindromes:
        print(f"{len(palindromes)}
Palindromic strings found: {',
'.join(palindromes)}")
    else:
        print("Palindrome not found")

# Test
find_palindromes_for_loop("Working in
JPMC")  # Output: Palindrome not found
find_palindromes_for_loop("MADAM reached
RADAR office")  # Output: Two Palindromic
strings found: MADAM, RADAR
```

3. Solution Using `while` Loop

```
python

def find_palindromes_while_loop(s):
    words = s.split()
    palindromes = []
    i = 0

    while i < len(words):
        word = words[i]
        if word.lower() == word[::-
1].lower():
            palindromes.append(word)
        i += 1

    if palindromes:
        print(f"{len(palindromes)}
Palindromic strings found: {',
'.join(palindromes)}")
    else:
        print("Palindrome not found")

# Test
find_palindromes_while_loop("Working in
JPMC")  # Output: Palindrome not found
find_palindromes_while_loop("MADAM
reached RADAR office")  # Output: Two
Palindromic strings found: MADAM, RADAR
```

4. Optimized Solution

In both the `for` and `while` loop implementations, the code is quite efficient already, but an optimization could involve handling punctuation and ignoring case by default:

```
python

def find_palindromes_optimized(s):
    words = s.split()
    palindromes = []

    for word in words:
        cleaned_word =
''.join(filter(str.isalnum,
word)).lower()  # Remove punctuation and
make lowercase
        if cleaned_word and cleaned_word
== cleaned_word[::-1]:
            palindromes.append(word)

    if palindromes:
        print(f"{len(palindromes)}
Palindromic strings found: {',
'.join(palindromes)}")
    else:
        print("Palindrome not found")

# Test
find_palindromes_optimized("MADAM reached
RADAR office!")  # Output: Two
Palindromic strings found: MADAM, RADAR
```

5. Object-Oriented Approach

```
python
```

```
class PalindromeFinder:
    def __init__(self, text):
        self.text = text

    def find_palindromes(self):
        words = self.text.split()
        palindromes = []

        for word in words:
            cleaned_word =
''.join(filter(str.isalnum,
word)).lower()
            if cleaned_word and
cleaned_word == cleaned_word[::-1]:
                palindromes.append(word)

        return palindromes

    def display_result(self):
        palindromes =
self.find_palindromes()
        if palindromes:
            print(f"{len(palindromes)}
Palindromic strings found: {',
'.join(palindromes)}")
        else:
            print("Palindrome not found")

# Test
finder = PalindromeFinder("MADAM reached
RADAR office")
finder.display_result()  # Output: Two
Palindromic strings found: MADAM, RADAR
```

6. Recursive Approach

```
python

def is_palindrome(word):
    if len(word) <= 1:
        return True
    if word[0] != word[-1]:
        return False
    return is_palindrome(word[1:-1])

def find_palindromes_recursive(words,
index=0, palindromes=None):
    if palindromes is None:
        palindromes = []

    if index >= len(words):
        return palindromes

    word = words[index]
    cleaned_word =
''.join(filter(str.isalnum,
word)).lower()

    if cleaned_word and
is_palindrome(cleaned_word):
        palindromes.append(word)
    return
find_palindromes_recursive(words, index +
1, palindromes)

def display_palindromes_recursive(s):
    words = s.split()
    palindromes =
find_palindromes_recursive(words)

    if palindromes:
        print(f"{len(palindromes)}
Palindromic strings found: {',
'.join(palindromes)}")
    else:
        print("Palindrome not found")

# Test
display_palindromes_recursive("MADAM
reached RADAR office")  # Output: Two
Palindromic strings found: MADAM, RADAR
```

7. Solution Without Using Inbuilt Python Functions

```
python

def find_palindromes_without_inbuilt(s):
    words = []
    start = 0

    # Split the string manually
    for i in range(len(s)):
        if s[i] == ' ':
            words.append(s[start:i])
            start = i + 1
    words.append(s[start:])

    palindromes = []

    # Check for palindrome without using
inbuilt functions
    for word in words:
        is_palindrome = True
        cleaned_word = ''.join([c.lower()
for c in word if c.isalnum()])
        for i in range(len(cleaned_word)
// 2):
            if cleaned_word[i] !=
cleaned_word[len(cleaned_word) - 1 - i]:
                is_palindrome = False
                break
        if is_palindrome and
cleaned_word:
            palindromes.append(word)

    if palindromes:
        print(f"{len(palindromes)}
Palindromic strings found: {',
'.join(palindromes)}")
    else:
        print("Palindrome not found")
```

```
# Test
find_palindromes_without_inbuilt("MADAM
reached RADAR office")  # Output: Two
Palindromic strings found: MADAM, RADAR
```

Find first non-repeating character of given String

Last Updated : 05 Jul, 2024

Given a string **S** of lowercase English letters, the task is to find the index of the first non-repeating character. If there is no such character, return -1.

Examples:

Input: *S = "geeksforgeeks"*
Output: *5*
Explanation: *'f' is the first character in the string which does not repeat.*

Input: *"aabbccc"*
Output: *-1*
Explanation: *All the characters in the given string are repeating*

Approaches to find first non-repeating character of given String:

Table of Content

[Naive Approach] Using Two Nested Loops – O(n^2) time and O(1) auxiliary space

The very basic idea is to use two nested loops, the outer loop for picking an element and the inner loop for checking the repeating character for picked element. If no repeating character found for any characters then return the index of currently picked element, otherwise return -1;

Code Implementation:

Recommended Problem

Non Repeating Character

Topics: Hash Strings +1 more

Companies: Flipkart Amazon +11 more

Solve Problem

Easy 40.43% 2.2L

C++ Java Python C# JavaScript

```python
def firstUniqChar(s):
    n = len(s)

    # Step 1: Iterate over each character in the string
    for i in range(n):
        found = True

        # Step 2: Check if the character repeats in the rest of the string
        for j in range(n):
            if i != j and s[i] == s[j]:
                found = False
                break

        # Step 3: If character does not repeat, return its index
        if found:
            return i

    # Step 4: If no such character is found, return -1
    return -1


# Driver code
s = "geeksforgeeks"
print(firstUniqChar(s))
```

Output

```
5
```

Time Complexity: $O(n^2)$
Auxiliary Space: O(1)

[Efficient Approach-1] Using Frequency Counting (Two Traversal) – O(n) time and O(1) auxiliary space

The efficient approach is to either use a hash map or ***an array of size 26*** *to store the* ***frequencies*** *of each character. After choosing either of data structure for frequency counting, traversing the string*

twice: once to populate the frequency into chosen data structure and other to find the first character with a frequency of one.

Code Implementation:

C++ Java Python C# JavaScript

```python
def firstUniqChar(s):
    # Step 1: Initialize frequency array for 26 lowercase letters
    freq = [0] * 26

    # Step 2: Populate the frequency array
    for c in s:
        freq[ord(c) - ord('a')] += 1

    # Step 3: Find the first character with frequency 1
    for i in range(len(s)):
        if freq[ord(s[i]) - ord('a')] == 1:
            return i

    return -1


# Driver code
if __name__ == "__main__":
    s = "geeksforgeeks"
    print(firstUniqChar(s))
```

Output

```
5
```

Time Complexity: O(n), where n is the length of given string
Auxiliary Space: O(1)

[Efficient Approach-2] By Hashing for Single Traversal – O(n) time and O(1) auxiliary space

*The above approach can be optimized using a single traversal of the string. The idea is to maintain a frequency array of size **26** initialized to **-1**, indicating no characters have been seen yet.*

Now, we iterate through the string:

- if a character is seen for the first time, its index is stored in the array.
- If the character is found again then its array value is set to **-2** to represent this character is now repeating.
- After the string traversal, traverse the frequency array and check if value in the array is not equal to **-1** or **-2** (means, this character is not repeating)
    - then we'll minimize the array value with stored index of non-repeating character.
    - If no such index is found, the function returns **-1.**

Code Implementation:

C++ Java Python C# JavaScript

```python
def first_uniq_char(s):

    # List to store the first occurrence index of each character
    freq = [-1] * 26

    # Iterate through the string
    for i in range(len(s)):
        cur = s[i]
        if freq[ord(cur) - ord('a')] == -1:
            freq[ord(cur) - ord('a')] = i
        else:
            freq[ord(cur) - ord('a')] = -2

    # Initialize idx to maximum integer value
    idx = float('inf')

    # Find the smallest index which is not marked as -1 or -2
    for i in range(26):
        if freq[i] >= 0:
            idx = min(idx, freq[i])

    # If no non-repeating character is found, return -1
    return -1 if idx == float('inf') else idx


# Example usage
s = "geeksforgeeks"
print(first_uniq_char(s))
```

Output

```
-1
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Related Problem:

- K’th Non-repeating Character

Egg Dropping Puzzle | DP-11

Last Updated : 01 Jul, 2024

The following is a description of the instance of this famous puzzle involving N = 2 eggs and a building with K = 36 floors.
Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks when dropped, then it would break if dropped from a higher floor.
- If an egg survives a fall then it would survive a shorter fall.
- It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor does not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second-floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg droppings that are guaranteed to work in all cases?
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that the total number of trials is minimized.

Note: In this post, we will discuss a solution to a general problem with 'N' eggs and 'K' floors

Egg Dropping Puzzle using Recursion:

To solve the problem follow the below idea:

The solution is to try dropping an egg from every floor(from 1 to K) and recursively calculate the minimum number of droppings needed in the worst case. The floor which gives the minimum value in the worst case is going to be part of the solution.
In the following solutions, we return the minimum number of trials in the worst case; these solutions can be easily modified to print the floor numbers of every trial also.

What is worst case scenario?

Worst case scenario gives the user the surety of the threshold floor. For example- If we have '1' egg and 'K' floors, we will start dropping the egg from the first floor till the egg breaks suppose on the 'Kth' floor so the number of tries to give us surety is 'K'.

1. Optimal Substructure:

When we drop an egg from floor x, there can be two cases (1) The egg breaks (2) The egg doesn't break.

1. If the egg breaks after dropping from 'xth' floor, then we only need to check for floors lower than 'x' with remaining eggs as some floors should exist lower than 'x' in which the egg would not break, so the problem reduces to x-1 floors and n-1 eggs.
2. If the egg doesn't break after dropping from the 'xth' floor, then we only need to check for floors higher than 'x'; so the problem reduces to 'k-x' floors and n eggs.

Since we need to minimize the number of trials in the *worst* case, we take a maximum of two cases. We consider the max of the above two cases for every floor and choose the floor which yields the minimum number of trials.

Below is the illustration of the above approach:

K ==> Number of floors
N ==> Number of Eggs

eggDrop(N, K) ==> Minimum number of trials needed to find the critical floor in worst case.
eggDrop(N, K) = 1 + min{max(eggDrop(N – 1, x – 1), eggDrop(N, K – x)), where x is in {1, 2, …, K}}

Concept of worst case:

Let there be '2' eggs and '2' floors then-:
If we try throwing from '1st' floor:
Number of tries in worst case= 1+max(0, 1)

0=>If the egg breaks from first floor then it is threshold floor (best case possibility).
1=>If the egg does not break from first floor we will now have '2' eggs and 1 floor to test which will give answer as '1'.(worst case possibility)

We take the worst case possibility in account, so 1+max(0, 1)=2
If we try throwing from '2nd' floor:
Number of tries in worst case= 1+max(1, 0)
1=>If the egg breaks from second floor then we will have 1 egg and 1 floor to find threshold floor. (Worst Case)
0=>If egg does not break from second floor then it is threshold floor.(Best Case)
We take worst case possibility for surety, so 1+max(1, 0)=2.

The final answer is min(1st, 2nd, 3rd....., kth floor)
So answer here is '2'.

Below is the implementation of the above approach:

Recommended Problem

Egg Dropping Puzzle

Topics: Dynamic Programming Algorithms

Companies: VMWare Amazon +15 more

Solve Problem

Medium 39.64% 1.4L

C++ C Java Python 3 C# JavaScript

```python
import sys

# Function to get minimum number of trials
# needed in worst case with n eggs and k floors


def eggDrop(n, k):

    # If there are no floors, then no trials
    # needed. OR if there is one floor, one
    # trial needed.
    if (k == 1 or k == 0):
        return k

    # We need k trials for one egg
    # and k floors
    if (n == 1):
        return k

    min = sys.maxsize

    # Consider all droppings from 1st
    # floor to kth floor and return
    # the minimum of these values plus 1.
    for x in range(1, k + 1):

        res = max(eggDrop(n - 1, x - 1),
                  eggDrop(n, k - x))
        if (res < min):
            min = res

    return min + 1


# Driver Code
if __name__ == "__main__":

    n = 2
    k = 10
    print("Minimum number of trials in worst case with",
          n, "eggs and", k, "floors is", eggDrop(n, k))

# This code is contributed by ita_c
```

Output

```
Minimum number of trials in worst case with 2 eggs and 10 floors is 4
```

Note: The above function computes the same subproblems again and again. See the following partial recursion tree, E(2, 2) is being evaluated twice. There will be many repeated subproblems when you draw the complete recursion tree even for small values of N and K.

Partial recursion tree for 2 eggs and 4 floors.

Time Complexity: As there is a case of overlapping sub-problems the time complexity is exponential.
Auxiliary Space: O(1). As there was no use of any data structure for storing values.

Egg Dropping Puzzle using Dynamic Programming:

To solve the problem follow the below idea:

Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So Egg Dropping Puzzle has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array eggFloor[][] in a bottom-up manner.

In this approach, we work on the same idea as described above neglecting the case of calculating the answers to sub-problems again and again. The approach will be to make a table that will store the results of sub-problems so that to solve a sub-problem, would only require a look-up from the table which will take constant time, which earlier took exponential time.

Formally for filling DP[i][j] state where 'i' is the number of eggs and 'j' is the number of floors:
We have to traverse for each floor 'x' from '1' to 'j' and find a minimum of:

(1 + max(DP[i-1][j-1], DP[i][j-x])).

Below is the illustration of the above approach:

i => Number of eggs
j => Number of floors

Look up find maximum
Lets fill the table for the following case:
Floors = '4'
Eggs = '2'

1 2 3 4
1 2 3 4 => 1
1 2 2 3 => 2

For 'egg-1' each case is the base case so the number of attempts is equal to floor number.
For 'egg-2' it will take '1' attempt for 1st floor which is base case.

For floor-2 =>
Taking 1st floor 1 + max(0, DP[1][1])
Taking 2nd floor 1 + max(DP[1][1], 0)
DP[2][2] = min(1 + max(0, DP[1][1]), 1 + max(DP[1][1], 0))

For floor-3 =>
Taking 1st floor 1 + max(0, DP[2][2])
Taking 2nd floor 1 + max(DP[1][1], DP[2][1])
Taking 3rd floor 1 + max(0, DP[2][2])
DP[2][3]= min('all three floors') = 2

For floor-4 =>
Taking 1st floor 1 + max(0, DP[2][3])
Taking 2nd floor 1 + max(DP[1][1], DP[2][2])
Taking 3rd floor 1 + max(DP[1][2], DP[2][1])
Taking 4th floor 1 + max(0, DP[2][3])
DP[2][4]= min('all four floors') = 3

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript PHP

```python
# Python3 program for the above approach
INT_MAX = 32767

# Function to get minimum number of trials needed in worst
# case with n eggs and k floors


def eggDrop(n, k):
    # A 2D table where entry eggFloor[i][j] will represent minimum
    # number of trials needed for i eggs and j floors.
    eggFloor = [[0 for x in range(k + 1)] for x in range(n + 1)]

    # We need one trial for one floor and0 trials for 0 floors
    for i in range(1, n + 1):
        eggFloor[i][1] = 1
        eggFloor[i][0] = 0

    # We always need j trials for one egg and j floors.
    for j in range(1, k + 1):
        eggFloor[1][j] = j

    # Fill rest of the entries in table using optimal substructure
    # property
    for i in range(2, n + 1):
        for j in range(2, k + 1):
            eggFloor[i][j] = INT_MAX
            for x in range(1, j + 1):
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x])
                if res < eggFloor[i][j]:
                    eggFloor[i][j] = res

    # eggFloor[n][k] holds the result
```

```
        return eggFloor[n][k]


# Driver code
if __name__ == "__main__":
    n = 2
    k = 36
    print("Minimum number of trials in worst case with " + str(n) + " eggs and "
          + str(k) + " floors is " + str(eggDrop(n, k)))

# This code is contributed by Bhavya Jain
```

Output

```
Minimum number of trials in worst case with 2 eggs and 36 floors is 8
```

Time Complexity: O(N * K^2). As we use a nested for loop 'k^2' times for each egg
Auxiliary Space: O(N * K). A 2-D array of size 'n*k' is used for storing elements.

Egg Dropping Puzzle using Memoization:

To solve the problem follow the below idea:

We can use a 2D dp table in the first recursive approach to store the results of overlapping subproblems which will help to reduce the time complexity from exponential to quadratic

dp table state –> dp[i][j], where where 'i' is the number of eggs and 'j' is the number of floors:

Follow the below steps to solve the problem:

- Declare a 2-D array memo of size N+1 * K+1 and call the function solveEggDrop(N, K)
- If memo[N][K] is already computed then return memo[n][k]
- If K == 1 or K == 0 then return K (Base Case)
- If N == 1 return K (Base case)
- Create an integer min equal to the integer Maximum and an integer res
- Run a for loop from x equal to 1 till x is less than or equal to K
    - Set res equal to maximum of solveEggDrop(N-1, x-1) or solveEggDrop(N, k-x)
    - If res is less than integer min then set min equal to res
- Set memo[N][K] equal to min + 1
- Return min + 1

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript

```
# Python3 program for the above approach

import sys

MAX = 1000

memo = [[-1 for i in range(MAX)] for j in range(MAX)]


def solveEggDrop(n, k):

    if (memo[n][k] != -1):
        return memo[n][k]

    if (k == 1 or k == 0):
        return k

    if (n == 1):
        return k

    min = sys.maxsize
    res = 0

    for x in range(1, k+1):
        res = max(solveEggDrop(n - 1, x - 1), solveEggDrop(n, k - x))
        if (res < min):
            min = res

    memo[n][k] = min + 1
    return min + 1


# Driver code
if __name__ == '__main__':
    n = 2
    k = 36
    print("Minimum number of trials in worst case with " , n, " eggs and ", k, " floors is
", solveEggDrop(n, k))

# This code is contributed by gauravrajput1
```

Output

```
Minimum number of trials in worst case with 2 eggs and 36 floors is 8
```

Time Complexity: O(n*k*k) where n is number of eggs and k is number of floors.
Auxiliary Space: O(n*k) as 2D memoisation table has been used.

Approach: To solve the problem follow the below idea:

The approach with O(N * K^2) has been discussed before, where dp[N][K] = 1 + max(dp[N – 1][i – 1], dp[N][K – i]) for i in 1…K. You checked all the possibilities in that approach.

> *Consider the problem in a different way:*
> *dp[m][x] means that, given x eggs and m moves,*
> *what is the maximum number of floors that can be checked*
>
> *The DP equation is: dp[m][x] = 1 + dp[m – 1][x – 1] + dp[m – 1][x],*
> *which means we take 1 move to a floor.*
> *If egg breaks, then we can check dp[m – 1][x – 1] floors.*
> *If egg doesn't break, then we can check dp[m – 1][x] floors.*

Follow the below steps to solve the problem:

- Declare a 2-D array of size K+1 * N+1 and an integer m equal to zero
- While dp[m][n] < k
    - increase 'm' by 1 and run a for loop from x equal to one till x is less than or equal to n
    - Set dp[m][x] equal to 1 + dp[m-1][x-1] + dp[m-1][x]
- Return m

Below is the implementation of the above approach:

C++ Java Python C# JavaScript

```python
def minTrials(n, k):
    # Initialize 2D array of size (k+1) * (n+1).
    dp = [[0 for x in range(n + 1)] for y in range(k + 1)]
    m = 0 # Number of moves
    while dp[m][n] < k:
        m += 1
        for x in range(1, n + 1):
            dp[m][x] = 1 + dp[m - 1][x - 1] + dp[m - 1][x]
    return m

# Driver code
n, k = 2, 36
print("Minimum number of trials in worst case with", n, "eggs and", k, "floors is",
minTrials(n, k))
```

Output

```
Minimum number of trials in worst case with 2 eggs and 36 floors is 8
```

Time Complexity: O(N * K)
Auxiliary Space: O(N * K)

Egg Dropping Puzzle using space-optimized DP:

> *The fourth approach can be optimized to 1-D DP as for calculating the current row of the DP table, we require only the previous row results and not beyond that.*

Follow the below steps to solve the problem:

- Create an array dp of size N+1 and an integer m
- Run a for loop from m equal to zero till dp[n] < k
    - Run a nested for loop from x equal to n till x is greater than zero
    - Set dp[x] equal to 1 + dp[x-1]
- Return m

Below is the implementation of the above approach:

C++ Java Python C# JavaScript

```python
# Python implementation for the above approach.

def minTrials(n, k):
    # Initialize array of size (n+1) and m as moves.
    dp = [0 for i in range(n+1)]
    m = 0
    while dp[n] < k:
        m += 1
        for x in range(n, 0, -1):
            dp[x] += 1 + dp[x - 1]
    return m


# Driver code
n, k = 2, 36
print("Minimum number of trials in worst case with",
      n, "eggs and", k, "floors is", minTrials(n, k))
```

```
    # This code is contributed by Amit Mangal.
```

Output

```
Minimum number of trials in worst case with 2 eggs and 36 floors is 8
```

Time Complexity: O(N * K)
Auxiliary Space: O(N)

Check if two arrays are equal or not

Last Updated : 05 Jul, 2024

Given two arrays, **arr1** and **arr2** of equal length **N**, the task is to determine if the given arrays are equal or not. Two arrays are considered equal if:

- Both arrays contain the same set of elements.
- The arrangements (or permutations) of elements may be different.
- If there are repeated elements, the counts of each element must be the same in both arrays.

Examples:

Input: *arr1[] = {1, 2, 5, 4, 0}, arr2[] = {2, 4, 5, 0, 1}*
Output: *Yes*

Input: *arr1[] = {1, 2, 5, 4, 0, 2, 1}, arr2[] = {2, 4, 5, 0, 1, 1, 2}*
Output: *Yes*

Input: *arr1[] = {1, 7, 1}, arr2[] = {7, 7, 1}*
Output: *No*

Approaches to check if two arrays are equal or not

Table of Content

[Naive Approach] Using Sorting – O(N*log(N)) time and O(1) auxiliary Space

*The basic idea is to **sort** the both given arrays and idea behind **sorting** is that once both arrays are sorted, they will be identical if and only if they contain the same elements in the same quantities. After sorting, we compare the arrays element by element. If any pair of corresponding elements in the sorted arrays differ, we return **false** (i.e indicating the arrays are not equal). If all elements match, we return **true**.*

Code Implementation:

Recommended Problem

Check Equal Arrays

Topics: Arrays Hash +3 more

Companies: Goldman Sachs

Solve Problem

Basic 42.18% 3.5L

C++ Java Python C# JavaScript

```python
# Python3 program to find given
# two array are equal or not

# Returns true if arr1[0..n-1] and
# arr2[0..m-1] contain same elements.


def areEqual(arr1, arr2):
    N = len(arr1)
    M = len(arr2)

    # If lengths of array are not
    # equal means array are not equal
    if (N != M):
        return False

    # Sort both arrays
    arr1.sort()
    arr2.sort()

    # Linearly compare elements
    for i in range(0, N):
        if (arr1[i] != arr2[i]):
            return False

        # If all elements were same.
        return True


# Driver Code
if __name__ == "__main__":
    arr1 = [3, 5, 2, 5, 2]
    arr2 = [2, 3, 5, 5, 2]

    if (areEqual(arr1, arr2)):
        print("Yes")
    else:
        print("No")
```

Output

```
Yes
```

Time Complexity: O(N*log(N)), due to sorting
Auxiliary Space: O(1)

[Expected Approach] Using Hashing – O(N) time and O(N) auxiliary space

The hashing approach is more efficient approach for this problem. The use of a hash map (or dictionary) is to count the occurrences of each element in one array and then verifying these counts against the second array.

Code Implementation:

C++ Java Python C# JavaScript

```python
# Python3 program for the above approach

# Returns true if arr1[0..N-1] and
# arr2[0..M-1] contain same elements.

def is_arr_equal(arr1, arr2):
    # Check if the length of arrays are
    # equal or not: A Easy Logic Check
    if len(arr1) != len(arr2):
        return False

    # Create a dict named count to
    # store counts of each element
    count = {}
    # Store the elements of arr1
    # and their counts in the dictionary
    for i in arr1:
        if i in count:
                # Element already in dict, simply increment its count
            count[i] += 1
        else:
                # Element found for first time, initialize it with value 1.
            count[i] = 1

    # Traverse through arr2 and compare
    # the elements and its count with
    # the elements of arr1
    for i in arr2:
        # Return false if the element
        # is not in count or if any element
        # appears more no. of times than in arr1
        if i not in count or count[i] == 0:
            return False
        else:
                # If element is found, decrement
                # its value in the dictionary
            count[i] -= 1
    # Return true if both arr1 and
    # arr2 are equal
    return True


# Driver Code
if __name__ == "__main__":
    arr1 = [3, 5, 2, 5, 2]
    arr2 = [2, 3, 5, 5, 2]

    if is_arr_equal(arr1, arr2):
        print("Yes")
    else:
        print("No")
```

Output

```
Yes
```

Time Complexity: O(N), where N is the length of given array
Auxiliary Space: O(N)

Note: *Some of you might come up with a more better solution using XOR, but it wont work. Read below why*

[Not Possible Approaches] Using XOR

The idea can be come up that XOR can be used to check if both the arrays are equal, following the property that XOR of same numbers is always 0. This approach will be as follows:

- Find XOR of all elements of first Array as XOR(Arr1)
- Then XOR each element of second Array with this.
- If the result is 0, then the arrays are equal, otherwise not.

The point to be noted here is that, this approach might work for some test cases but not for all.

Examples:

Input: arr1[] = {1, 2, 5, 4, 0}, arr2[] = {2, 4, 5, 0, 1}
Approach:

- XOR(arr1) = 1 ^ 2 ^ 5 ^ 4 ^ 0 = 2
- XOR(arr2[i]) with XOR(arr1) = (((((2 ^ 2) ^ 4) ^ 5) ^ 0) ^ 1) = 0
- Since the final answer is 0, the given arrays are equal.

Conclusion: *The approach works for this testcase.*

Input: arr1[] = {11, 12}, arr2[] = {2, 5}
Approach:

- XOR(arr1) = 11 ^ 12 = 7
- XOR(arr2[i]) with XOR(arr1) = ((7 ^ 2) ^ 5) = 0
- Since the final answer is 0, the given arrays are equal.

Conclusion: *The approach does not works for this testcase.*

This is why XOR approach wont work for this problem.

Sort an array in wave form

Last Updated : 12 Jun, 2024

Given an unsorted array of integers, sort the array into a wave array. An array **arr[0..n-1]** is sorted in wave form if:

arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= …..

Examples:

> ***Input:*** *arr[] = {10, 5, 6, 3, 2, 20, 100, 80}*
> ***Output:*** *arr[] = {10, 5, 6, 2, 20, 3, 100, 80}*
> ***Explanation:***
> *here you can see {10, 5, 6, 2, 20, 3, 100, 80} first element is larger than the second and the same thing is repeated again and again. large element – small element-large element -small element and so on .it can be small element-larger element – small element-large element -small element too. all you need to maintain is the up-down fashion which represents a wave. there can be multiple answers.*
>
> ***Input:*** *arr[] = {20, 10, 8, 6, 4, 2}*
> ***Output:*** *arr[] = {20, 8, 10, 4, 6, 2}*

What is a wave array?

well, you have seen waves right? how do they look? if you will form a graph of them it would be some in some up-down fashion. that is what you have to do here, you are supposed to arrange numbers in such a way that if we will form a graph it will be in an up-down fashion rather than a straight line.

Wave Array using sorting

> *A idea is to use sorting. First sort the input array, then swap all adjacent elements.*

Follow the steps mentioned below to implement the idea:

- Sort the array.
- Traverse the array from index **0** to **N-1,** and increase the value of the index by **2**.
- While traversing the array swap **arr[i]** with **arr[i+1].**
- Print the final array.

Below is the implementation of the above approach:

Recommended Problem

Wave Array

Topics: Arrays Sorting +2 more

Companies: Paytm Flipkart +5 more

Solve Problem

Easy 63.69% 2.5L

C++ C Java Python C# JavaScript

```python
# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

    #sort the array
    arr.sort()

    # Swap adjacent elements
    for i in range(0,n-1,2):
        arr[i], arr[i+1] = arr[i+1], arr[i]

# Driver program
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print (arr[i],end=" ")

# This code is contributed by __Devesh Agrawal__
```

Output

```
2 1 10 5 49 23 90
```

Time Complexity: O(N*log(N))
Auxiliary Space: O(1)

Wave Array Optimized Approach

> *The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don't need to worry about oddly positioned elements.*

Follow the steps mentioned below to implement the idea:

- Traverse all even positioned elements of the input array, and do the following.
    - If the current element is smaller than the previous odd element, swap the previous and current.
    - If the current element is smaller than the next odd element, swap next and current.

Below is the implementation of the above approach:

C++14 Java Python C# JavaScript

```python
# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]


def sortInWave(arr, n):

    # Traverse all even elements
    for i in range(0, n - 1, 2):

        # If current even element is smaller than previous
        if (i > 0 and arr[i] < arr[i-1]):
            arr[i], arr[i-1] = arr[i-1], arr[i]

        # If current even element is smaller than next
        if (i < n-1 and arr[i] < arr[i+1]):
            arr[i], arr[i+1] = arr[i+1], arr[i]


# Driver program
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0, len(arr)):
    print(arr[i], end=" ")

# This code is contributed by __Devesh Agrawal__
```

Output

```
90 10 49 1 5 2 23
```

Time Complexity: O(N)
Auxiliary Space: O(1)

Ugly Numbers

Last Updated : 29 Dec, 2023

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is included.
Given a number n, the task is to find n'th Ugly number.

Examples:

```
Input  : n = 7
Output : 8

Input  : n = 10
Output : 12

Input  : n = 15
Output : 24

Input  : n = 150
Output : 5832
```

Method 1 (Simple)

Loop for all positive integers till the ugly number count is smaller than n, if an integer is ugly then increment ugly number count.
To check if a number is ugly, divide the number by greatest divisible powers of 2, 3 and 5, if the number becomes 1 then it is an ugly number otherwise not.

For example, let us see how to check for 300 is ugly or not. Greatest divisible power of 2 is 4, after dividing 300 by 4 we get 75. Greatest divisible power of 3 is 3, after dividing 75 by 3 we get 25. Greatest divisible power of 5 is 25, after dividing 25 by 25 we get 1. Since we get 1 finally, 300 is ugly number.

Below is the implementation of the above approach:

Recommended Problem

Ugly Number II

Solve Problem

Medium 54.32% 589

C++ C Java Python3 C# Javascript PHP

```python
# Python3 code to find nth ugly number

# This function divides a by greatest
# divisible power of b


def maxDivide(a, b):
    while a % b == 0:
        a = a / b
    return a

# Function to check if a number
# is ugly or not
def isUgly(no):
    no = maxDivide(no, 2)
    no = maxDivide(no, 3)
    no = maxDivide(no, 5)
    return 1 if no == 1 else 0

# Function to get the nth ugly number
def getNthUglyNo(n):
    i = 1

    # ugly number count
    count = 1

    # Check for all integers until
    # ugly count becomes n
    while n > count:
        i += 1
        if isUgly(i):
            count += 1
    return i


# Driver code
no = getNthUglyNo(150)
print("150th ugly no. is ", no)

# This code is contributed by "Sharad_Bhardwaj".
```

Output

```
150th ugly no. is 5832
```

This method is not time efficient as it checks for all integers until ugly number count becomes n, but space complexity of this method is O(1)

Method 2 (Use Dynamic Programming)

Here is a time efficient solution with O(n) extra space. The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, …

because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:

(1) 1×2, 2×2, 3×2, 4×2, 5×2, …

(2) 1×3, 2×3, 3×3, 4×3, 5×3, …

(3) 1×5, 2×5, 3×5, 4×5, 5×5, …

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, …) multiply 2, 3, 5. Then we use similar merge method as merge sort, to get every ugly number from the three subsequences. Every step we choose the smallest one, and move one step after.

```
1 Declare an array for ugly numbers:  ugly[n]
2 Initialize first ugly no:  ugly[0] = 1
3 Initialize three array index variables i2, i3, i5 to point to
   1st element of the ugly array:
        i2 = i3 = i5 =0;
4 Initialize 3 choices for the next ugly no:
         next_multiple_of_2 = ugly[i2]*2;
         next_multiple_of_3 = ugly[i3]*3
         next_multiple_of_5 = ugly[i5]*5;
5 Now go in a loop to fill all ugly numbers till 150:
For (i = 1; i < 150; i++ )
{
    /* These small steps are not optimized for good
      readability. Will optimize them in C program */
    next_ugly_no  = Min(next_multiple_of_2,
                        next_multiple_of_3,
                        next_multiple_of_5);

    ugly[i] =  next_ugly_no

    if (next_ugly_no  == next_multiple_of_2)
    {
        i2 = i2 + 1;
        next_multiple_of_2 = ugly[i2]*2;
    }
    if (next_ugly_no  == next_multiple_of_3)
    {
        i3 = i3 + 1;
        next_multiple_of_3 = ugly[i3]*3;
     }
     if (next_ugly_no  == next_multiple_of_5)
     {
        i5 = i5 + 1;
        next_multiple_of_5 = ugly[i5]*5;
     }

}/* end of for loop */
6.return next_ugly_no
```

Example:

Let us see how it works

```
initialize
   ugly[] =  | 1 |
   i2 =  i3 = i5 = 0;

First iteration
   ugly[1] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
           = Min(2, 3, 5)
           = 2
   ugly[] =  | 1 | 2 |
   i2 = 1,  i3 = i5 = 0  (i2 got incremented )

Second iteration
    ugly[2] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
            = Min(4, 3, 5)
            = 3
    ugly[] =  | 1 | 2 | 3 |
    i2 = 1,  i3 =  1, i5 = 0  (i3 got incremented )

Third iteration
    ugly[3] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
            = Min(4, 6, 5)
            = 4
    ugly[] =  | 1 | 2 | 3 |  4 |
    i2 = 2,  i3 =  1, i5 = 0  (i2 got incremented )
```

```
Fourth iteration
    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(6, 6, 5)
              = 5
    ugly[] =  | 1 | 2 | 3 |  4 | 5 |
    i2 = 2,  i3 =  1, i5 = 1  (i5 got incremented )

Fifth iteration
    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(6, 6, 10)
              = 6
    ugly[] =  | 1 | 2 | 3 |  4 | 5 | 6 |
    i2 = 3,  i3 =  2, i5 = 1  (i2 and i3 got incremented )

Will continue same way till I < 150
```

C++ C Java Python C# Javascript PHP

```python
# Python program to find n'th Ugly number

# Function to get the nth ugly number


def getNthUglyNo(n):

    ugly = [0] * n  # To store ugly numbers

    # 1 is the first ugly number
    ugly[0] = 1

    # i2, i3, i5 will indicate indices for
    # 2,3,5 respectively
    i2 = i3 = i5 = 0

    # Set initial multiple value
    next_multiple_of_2 = 2
    next_multiple_of_3 = 3
    next_multiple_of_5 = 5

    # Start loop to find value from
    # ugly[1] to ugly[n]
    for l in range(1, n):

        # Choose the min value of all
        # available multiples
        ugly[l] = min(next_multiple_of_2,
                      next_multiple_of_3,
                      next_multiple_of_5)

        # Increment the value of index accordingly
        if ugly[l] == next_multiple_of_2:
            i2 += 1
            next_multiple_of_2 = ugly[i2] * 2

        if ugly[l] == next_multiple_of_3:
            i3 += 1
            next_multiple_of_3 = ugly[i3] * 3

        if ugly[l] == next_multiple_of_5:
            i5 += 1
            next_multiple_of_5 = ugly[i5] * 5

    # Return ugly[n] value
    return ugly[-1]

# Driver Code
def main():

    n = 150

    print getNthUglyNo(n)


if __name__ == '__main__':
    main()

# This code is contributed by Neelam Yadav
```

Output

```
5832
```

Time Complexity: O(n)
Auxiliary Space: O(n)
Super Ugly Number (Number whose prime factors are in the given set)

Method 3 (Using SET Data Structure in C++, Javascript and TreeSet in JAVA)

In this method, SET data structure to store the minimum of the 3 generated ugly numbers which will the i^{th} Ugly Number stored at the first position of the set. SET Data Structure as a set stores all the element in ascending order

Below is the implementation of the above approach:

C++ Java Python3 C# Javascript

```python
# Python Implementation of the above approach
def nthUglyNumber(n):

    # Base cases...
    if (n == 1 or n == 2 or n == 3 or n == 4 or n == 5):
        return n
    s = [1]
    n-=1

    while (n):
        it = s[0]

        # Get the beginning element of the set
        x = it

        # Deleting the ith element
        s = s[1:]
        s = set(s)

        # Inserting all the other options
        s.add(x * 2)
        s.add(x * 3)
        s.add(x * 5)
        s = list(s)
        s.sort()
        n -= 1
    # The top of the set represents the nth ugly number
    return s[0]

# Driver Code
n = 150

# Function call
print( nthUglyNumber(n))

# This code is contributed by Shubham Singh
```

Output

```
5832
```

Time Complexity:- O(N log N)
Auxiliary Space:- O(N)

Method 4(Using Binary Search)

1. This method is suitable if you have a max value for n. The no will be of form x=pow(2,p)*pow(3,q)*pow(5,r).
2. Search from low=1 to high =21474836647. We are expecting nth ugly no to be in this range.
3. So we do a binary search. Now suppose we are at mid now we are going to find the total number of ugly numbers less than mid and put our conditions accordingly.

Below is the rough CPP code:

C++ Java Python3 C# Javascript

```python
# Python Program to implement
# the above approach

def upper_bound(a, low, high, element) :
    while(low < high) :
      middle = low + (high - low)//2
      if(a[middle] > element) :
        high = middle
      else :
        low = middle + 1

    return low

# Print nth Ugly number
def nthUglyNumber(n) :

  pow = [1] * 40

  # stored powers of 2 from
  # pow(2,0) to pow(2,30)
  for i in range(1, 31):
    pow[i] = pow[i - 1] * 2

  # Initialized low and high
  l = 1
  r = 2147483647

  ans = -1

  # Applying Binary Search
  while (l <= r) :

    # Found mid
    mid = l + ((r - l) // 2)
```

```
    # cnt stores total numbers of ugly
    # number less than mid
    cnt = 0

    # Iterate from 1 to mid
    i = 1
    while(i <= mid) :

        # Possible powers of i less than mid is i
        j = 1
        while(j * i <= mid) :
            # possible powers of 3 and 5 such that
            # their product is less than mid

            # using the power array of 2 (pow) we are
            # trying to find the max power of 2 such
            # that i*J*power of 2 is less than mid

            cnt += upper_bound(pow, 0,  31, mid // (i * j))
            j *= 3

        i *= 5



    # If total numbers of ugly number
    # less than equal
    # to mid is less than n we update l
    if (cnt < n):
      l = mid + 1

    # If total numbers of ugly number
    # less than equal to
    # mid is greater than n we update
    # r and ans simultaneously.
    else :
      r = mid - 1
      ans = mid
  return ans


# Driver Code
n = 150

# Function Call
print(nthUglyNumber(n))

# This code is contributed by code_hunt.
```

Output

```
5832
```

Time Complexity: O(log N)
Auxiliary Space: O(1)

Reverse words in a string

Last Updated : 20 Aug, 2024

Given a string, the task is to reverse the order of the words in the given string.

Examples:

Input: *s = “i love programming very much”*
Output: *s = “much very programming love i”*

Input*: s = ” geeks for all”*
Output*: s = “all for geeks”*
We need to remove all the extra spaces in the output

Using Split, Reverse and Join:

We follow the three steps mentioned below.

1. Split the given string into an array of words. For example, ” geeks for all” is spitted as {“geeks”, “for”, “all”}
2. Reverse the array of words. We get {“all”, “for”, “geeks”}
3. Join the reversed array back into a string. We get “all for geeks”

Below is the implementation of the above approach:

C++ Java Python C# JavaScript PHP

```python
def reverse_words(string):

    # Split words in the given string
    # and reverse the order
    reversed_words = string.split()[::-1]

    # Join the reversed words into a
    # single string
    return " ".join(reversed_words)

# Input string
string = " geeks for all"
print(reverse_words(string))
```

Output

```
all for geeks
```

Using Stack

1. Push all words separated by a space into a stack.
2. Set the string as empty.
3. Pop all words one by one from stack and append back into the string.

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript

```python
# Function to reverse words
def reverse_words(s):

    # Create a stack and push all
    # words one by one
    stack = []
    word = ""

    # Traverse the string and split words
    for char in s:
        if char != ' ':
            word += char

        # If we see a space, push the
        # previously seen word into the stack
        elif word:
            stack.append(word)
            word = ""

    # Last word remaining, add it to stack
    if word:
        stack.append(word)

    # Now print from top to bottom of the stack
    while stack:
        print(stack.pop(), end=" ")
```

```
# Input string
string = "  geeks for all"
reverse_words(string)
```

Output

```
all for geeks
```

Time complexity: O(n)
Auxiliary Space: O(n)

In this method, we can also use library methods to split as we have show in the first approach.

Using Two Reverses:

This solution works in O(1) auxiliary space and has mainly two steps.

1. We first reverse individual words of the given string. For example if the given string is “geeks for all” would become “skeeg rof lla”
2. Then we reverse the whole string. The string “skeeg rof lla” would become “all for geeks”

C++ C Java Python C# JavaScript

```
# Function to reverse individual words
# in a string
def reverse_words(s):
    word_begin = -1

    # STEP 1. Reverse each word in the string
    for i in range(len(s)):

        # Beginning of a word
        if word_begin == -1 and s[i] != ' ':
            word_begin = i

        # End of a word
        if word_begin != -1 and (i + 1 == len(s) or s[i + 1] == ' '):
            reverse(s, word_begin, i)
            word_begin = -1

    # STEP 2. Reverse the entire string
    reverse(s, 0, len(s) - 1)

# Function to reverse a sequence from
# "begin" index to "end" index
def reverse(s, begin, end):
    while begin < end:
        s[begin], s[end] = s[end], s[begin]
        begin += 1
        end -= 1

# Driver Code
s = list("  geeks for all")
reverse_words(s)
print("".join(s))
```

Output

```
all for geeks
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Find Excel column name from a given column number

Last Updated : 17 Aug, 2022

MS Excel columns have a pattern like A, B, C, …, Z, AA, AB, AC, …., AZ, BA, BB, … ZZ, AAA, AAB ….. etc. In other words, column 1 is named “A”, column 2 as “B”, and column 27 as “AA”.
Given a column number, find its corresponding Excel column name. The following are more examples.

Input	Output
26	Z
51	AY
52	AZ
80	CB
676	YZ
702	ZZ
705	AAC

Thanks to Mrigank Dembla for suggesting the below solution in a comment.
Suppose we have a number n, let’s say 28. so corresponding to it we need to print the column name. We need to take the remainder with 26.

If the remainder with 26 comes out to be 0 (meaning 26, 52, and so on) then we put ‘Z’ in the output string and new n becomes n/26 -1 because here we are considering 26 to be ‘Z’ while in actuality it’s 25th with respect to ‘A’.

Similarly, if the remainder comes out to be non-zero. (like 1, 2, 3, and so on) then we need to just insert the char accordingly in the string and do n = n/26.

Finally, we reverse the string and print.

Example:
n = 700
The remainder (n%26) is 24. So we put ‘X’ in the output string and n becomes n/26 which is 26.
Remainder (26%26) is 0. So we put ‘Z’ in the output string and n becomes n/26 -1 which is 0.

Following is the implementation of the above approach.

Recommended Problem

Column name from a given column number

Topics: Strings Mathematical +2 more

Solve Problem

Medium 32.41% 78K

Companies: Paytm Amazon +4 more

C++ Java Python C# Javascript

```python
# Python program to find Excel column name from a
# given column number

MAX = 50

# Function to print Excel column name
# for a given column number
def printString(n):

    # To store result (Excel column name)
    string = ["\0"] * MAX

    # To store current index in str which is result
    i = 0

    while n > 0:
        # Find remainder
        rem = n % 26

        # if remainder is 0, then a
        # 'Z' must be there in output
        if rem == 0:
            string[i] = 'Z'
            i += 1
            n = (n / 26) - 1
        else:
            string[i] = chr((rem - 1) + ord('A'))
            i += 1
            n = n / 26
    string[i] = '\0'

    # Reverse the string and print result
    string = string[::-1]
    print "".join(string)

# Driver program to test the above Function
printString(26)
printString(51)
printString(52)
printString(80)
printString(676)
printString(702)
printString(705)

# This code is contributed by BHAVYA JAIN
```

Output

```
Z
AY
AZ
CB
YZ
ZZ
AAC
```

Time Complexity: O($\log_{26}$n), as we are using a loop and in each traversal, we decrement by floor division of 26.

Auxiliary Space: O(50), as we are using extra space for storing the result.

Method 2

The problem is similar to converting a decimal number to its binary representation but instead of a binary base system where we have two digits only 0 and 1, here we have 26 characters from A-Z.

So, we are dealing with base 26 instead of base binary.

That's not where the fun ends, we don't have zero in this number system, as A represents 1, B represents 2 and so on Z represents 26.

To make the problem easily understandable, we approach the problem in two steps:

1. Convert the number to base 26 representation, considering we have 0 also in the system.
2. Change the representation to the one without having 0 in its system.

HOW? Here is an example

Step 1:

Consider we have number 676, How to get its representation in the base 26 system? In the same way, we do for a binary system, Instead of division and remainder by 2, we do division and remainder by 26.

```
Base 26 representation of 676 is : 100
```

Step2

But Hey, we can't have zero in our representation. Right? Because it's not part of our number system. How do we get rid of zero? Well it's simple, but before doing that let's remind one simple math trick:

```
Subtraction:
5000 - 9, How do you subtract 9 from 0 ? You borrow
from next significant bit, right.
```

- In a decimal number system to deal with zero, we borrow 10 and subtract 1 from the next significant.
- In the Base 26 Number System to deal with zero, we borrow 26 and subtract 1 from the next significant bit.

So Convert 100_{26} to a number system that does not have '0', we get $(25\ 26)_{26}$

Symbolic representation of the same is: YZ

Here is the implementation of the same:

C++ Java Python3 C# Javascript

```python
def printString(n):

    arr = [0] * 10000
    i = 0

    # Step 1: Converting to number
    # assuming 0 in number system
    while (n > 0):
        arr[i] = n % 26
        n = int(n // 26)
        i += 1

    #Step 2: Getting rid of 0, as 0 is
    # not part of number system
    for j in range(0, i - 1):
        if (arr[j] <= 0):
            arr[j] += 26
            arr[j + 1] = arr[j + 1] - 1

    for j in range(i, -1, -1):
        if (arr[j] > 0):
            print(chr(ord('A') +
                  (arr[j] - 1)), end = "");

    print();

# Driver code
if __name__ == '__main__':

    printString(26);
    printString(51);
    printString(52);
    printString(80);
```

```
    printString(676);
    printString(702);
    printString(705);

# This code is contributed by Princi Singh
```

Output

```
Z
AY
AZ
CB
YZ
ZZ
AAC
```

Time Complexity: O($\log_{26}$n), as we are using a loop and in each traversal, we decrement by floor division of 26.

Auxiliary Space: O(10000), as we are using extra space for the array.

Method 3:

We can use a recursive function which definitely reduces the time and increase the efficiency:

Alphabets are in sequential order like: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. You have experienced while using excel when you see columns and rows numbering are done in Alphabetical ways.

Here's How I purposefully think about the logic of how it is arranged.

(In Mathematical terms, [a , b] means from 'a' to 'b').

[1,26] = [A,Z] (Understand by '1' stands for 'A' and '26" stands for "Z"). For [27,52] ,it will be like [AA,AZ], For [57,78] it will be [BA,BZ]

Logic is to append an Alphabet sequentially whenever it ends up numbering at 26.

For example, if the number is '27' which is greater than '26', then we simply need to divide by 26, and we get the remainder as 1, We see "1" as "A" and can be recursively done.

we will be using python for this.

Algorithm is:

1. Take an array and Sort the letters from A to Z . (You can also use the import string and string function to get "A to Z" in uppercase.)

2. **If** the number is less than or equal to '26', simply get the letter from the array and print it.

3. **If** it is greater than 26, use the Quotient Remainder rule, if the remainder is zero, there are 2 possible ways, **if** the quotient is "1", simply hash out the letter from the index [r-1]('r' is remainder), **else** call out the function from the num =(q-1) and append at the front to the letter indexing [r-1].

4. **If** the remainder is not equal to "0", call the function for the num = (q) and append at the front to the letter indexing [r-1].

The code concerned with this is:

C++ Java Python3 C# Javascript

```python
# Or you can simply take a string and perform this logic ,no issue i found in here.
alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
# defined a recursive function here.
# if number is less than "26", simply hash out (index-1)
# There are sub possibilities in possibilities,
# 1.if remainder is zero(if quotient is 1 or not 1) and
# 2. if remainder is not zero

def num_hash(num):
    if num < 26:
        return alpha[num-1]
    else:
        q, r = num//26, num % 26
        if r == 0:
            if q == 1:
                return alpha[r-1]
            else:
                return num_hash(q-1) + alpha[r-1]
        else:
            return num_hash(q) + alpha[r-1]


# Calling the function out here and printing the ALphabets
# This code is robust ,work for any positive integer only.
# You can try as much as you want
print(num_hash(26))
print(num_hash(51))
print(num_hash(52))
print(num_hash(80))
print(num_hash(676))
print(num_hash(702))
print(num_hash(705))
```

Output

```
Z
AY
AZ
CB
YZ
ZZ
AAC
```

Time Complexity: O($\log_{26}$n), as we are using recursion and in each recursive call, we decrement by floor division of 26.

Auxiliary Space: O(1), as we are not using any extra space.

Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it?

Last Updated : 18 Oct, 2023

Given a pointer to a node to be deleted, delete the node. Note that we don't have a pointer to the head node.

A **simple solution** is to traverse the linked list until you find the node you want to delete. But this solution requires a pointer to the head node, which contradicts the problem statement.
The **fast solution** is to copy the data from the next node to the node to be deleted and delete the next node. Something like the following.

```
// Find next node using next pointer
struct Node *temp  = node_ptr->next;

// Copy data of next node to this node
node_ptr->data  = temp->data;

// Unlink next node
node_ptr->next  = temp->next;

// Delete next node
free(temp);
```

Program:

Recommended Problem

Delete without head pointer

Topics: Linked List, Data Structures

Companies: Amazon, Microsoft, +4 more

Solve Problem

Easy 78.57% 2L

C++ C Java Python C# Javascript

```
# a class to define a node with
# data and next pointer
class Node():
    # constructor to initialize a new node
    def __init__(self, val = None):
        self.data = val
        self.next = None

# push a node to the front of the list
def push(head, val):

    # allocate new node
    newnode = Node(val)

    # link the first node of the old list to the new node
    newnode.next = head.next

    # make the new node as head of the linked list
    head.next = newnode

# function to print the list
def print_list(head):

    temp = head.next
    while(temp != None):
        print(temp.data, end = ' ')
        temp = temp.next
    print()

# function to delete the node
# the main logic is in this
def delete_node(node):

    prev = Node()

    if(node == None):
        return
    else:
        temp = Node()
        temp = node.next;
        node.data = temp.data;
        node.next = temp.next;
        temp = None;


if __name__ == '__main__':

    # allocate an empty header node
    # this is a node that simply points to the
    # first node in the list
    head = Node()

    # construct the below linked list
    # 1->12->1->4->1
    push(head, 1)
    push(head, 4)
    push(head, 1)
    push(head, 12)
    push(head, 1)
```

```
    print('list before deleting:')
    print_list(head)

    # deleting the first node in the list
    delete_node(head.next)

    print('list after deleting: ')
    print_list(head)

# This code is contributed by Adith Bharadwaj
```

Output:

```
Before deleting
1 12 1 4 1
After deleting
12 1 4 1
```

Time Complexity:

- **For printing linked list:** O(N)
- **For inserting node:** O(1)
- **For deleting node:** O(N)
- **Auxiliary Space:** O(1)

This solution doesn't work if the node to be deleted is the last node of the list. To make this solution work, we can mark the end node as a dummy node. But the programs/functions that are using this function should also be modified.
Exercise: Try this problem with the doubly linked list.

One line in the function deletenode():

C++ Java Python3 C# Javascript

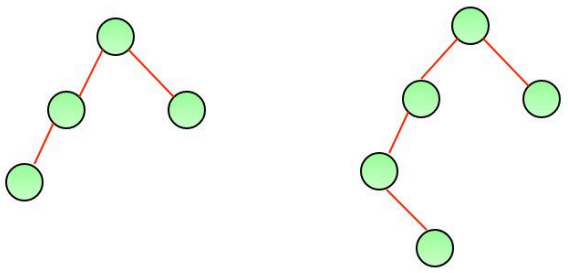
```
def deleteNode(Node Node):
   Node = (Node.next);

# This code is contributed by gauravrajput1
```

Balanced Binary Tree or Not

Last Updated : 14 Aug, 2024

A **height balanced binary tree** is a binary tree in which the height of the left subtree and right subtree of any node does not differ by more than 1 and both the left and right subtree are also height balanced.

> ***Examples:*** *The tree on the left is a height balanced binary tree. Whereas the tree on the right is not a height balanced tree. Because the left subtree of the root has a height which is 2 more than the height of the right subtree.*



A height balanced tree

Not a height balanced tree

> ***Corner Cases*** *: An empty binary tree (Root = NULL) and a Binary Tree with single node are considered balanced.*

Naive Approach: To check if a tree is height-balanced:

> *Get the height of left and right subtrees using **dfs** traversal. Return true if the difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.*

Below is the implementation of the above approach.

Recommended Problem

Balanced Tree Check

Topics: Tree Data Structures

Solve Problem

Companies: Amazon Microsoft +1 more

Easy 43.15% 3.1L

C++ C Java Python C# JavaScript

```python
"""
Python3 program to check if a tree is height-balanced
"""
# A binary tree Node


class Node:
    # Constructor to create a new Node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# function to find height of binary tree


def height(root):

    # base condition when binary tree is empty
    if root is None:
        return 0
    return max(height(root.left), height(root.right)) + 1

# function to check if tree is height-balanced or not


def isBalanced(root):

    # Base condition
    if root is None:
        return True

    # for left and right subtree height
    lh = height(root.left)
    rh = height(root.right)

    # allowed values for (lh - rh) are 1, -1, 0
    if (abs(lh - rh) <= 1) and isBalanced(
            root.left) is True and isBalanced(root.right) is True:
        return True

    # if we reach here means tree is not
    # height-balanced tree
    return False


# Driver function to test the above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
```

```
root.left.right = Node(5)
root.left.left.left = Node(8)
if isBalanced(root):
    print("Tree is balanced")
else:
    print("Tree is not balanced")

# This code is contributed by Shweta Singh
```

Output

```
Tree is not balanced
```

Time Complexity: O(n^2)
Auxiliary Space: O(n) space for call stack since using recursion

Efficient implementation: Above implementation can be optimized by

Calculating the height in the same recursion rather than calling a height() function separately.

- For each node make two recursion calls – one for left subtree and the other for the right subtree.
- Based on the heights returned from the recursion calls, decide if the subtree whose root is the current node is height-balanced or not.
- If it is balanced then return the height of that subtree. Otherwise, return **-1** to denote that the subtree is not height-balanced.

Below is the implementation of the above approach.

C++ C Java Python C# JavaScript

```
"""
Python3 program to check if a tree is height-balanced
"""
# A binary tree Node

class Node:

    # Constructor to create a new Node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to check if tree is height-balanced or not


def isBalanced(root):

    # Base condition
    if root is None:
        return True

    # Compute height of left subtree
    lh = isBalanced(root.left)

    # If left subtree is not balanced,
    # return 0
    if lh == 0:
        return 0

    # Do same thing for the right subtree
    rh = isBalanced(root.right)
    if rh == 0:
        return 0

    # Allowed values for (lh - rh) are 1, -1, 0
    if (abs(lh - rh) > 1):
        return 0

    # If we reach here means tree is
    # height-balanced tree, return height
    # in this case
    else:
        return max(lh, rh) + 1


# Driver code
if __name__ == '__main__':
    root = Node(10)
    root.left = Node(5)
    root.right = Node(30)
    root.right.left = Node(15)
    root.right.right = Node(20)
    if (isBalanced(root) == 0):
        print("Not Balanced")
    else:
        print("Balanced")

# This code is contributed by Shweta Singh
```

Output

```
Balanced
```

Time Complexity: O(n)

- Because we are only one dfs call and utilizing the height returned from that to determine the height balance, it is performing the task in linear time.

Auxiliary Space: O(h) where h is height of the binary tree.

Write a function to get the intersection point of two Linked Lists

Last Updated : 14 Aug, 2024

Given two singly linked lists that merge into a single Y-shaped list. The two lists initially have distinct paths but eventually converge at a common node, forming a Y-shape, the task is to find and return the node where the two lists merge.

Intersection Point of two Linked Lists

The above diagram shows an example with two linked lists having 15 as intersection point.

Table of Content

[Naive Approach] By using two nested loops – O(M*N) Time and O(1) Space

The problem can be easily solved using 2 nested for loops. The outer loop will be for each node of the 1st list and the inner loop will be for each node of the 2nd list. In the inner loop, check if any of the nodes of the 2nd list is the same as the current node of the first linked list. The first node which is same as the current node is the intersection point.

Below is the implementation of above approach:

C++ C Java Python C# JavaScript

```python
# Python Program to get intersection point of two linked
# lists using two nested loops

class Node:
    def __init__(self, new_data):
        self.data = new_data
        self.next = None


# function to get the intersection point of two linked
# lists head1 and head2
def get_intersection_node(head1, head2):

    # Traverse the second list
    while head2 is not None:
        temp = head1

        # Traverse the first list to check if the node matches head2
        while temp is not None:
            if temp is head2:
                return head2
            temp = temp.next
        head2 = head2.next

    # If intersection is not present, return None
    return None


if __name__ == "__main__":

    # Create two linked lists
    # 1st list: 10 -> 15 -> 30
    # 2nd list: 3 -> 6 -> 9 -> 15 -> 30
    # 15 is the intersection point

    # Creating the first linked list
    head1 = Node(10)
    head1.next = Node(15)
    head1.next.next = Node(30)

    # Creating the second linked list
    head2 = Node(3)
    head2.next = Node(6)
    head2.next.next = Node(9)
    head2.next.next.next = head1.next

    # Finding the intersection point
    intersection_point = get_intersection_node(head1, head2)

    if intersection_point is None:
        print("No Intersection Point")
    else:
        print("Intersection Point:", intersection_point.data)
```

Output

```
Intersection Point: 15
```

Time Complexity: O(M*N), where **M** and **N** are number of nodes in the two linked list.
Auxiliary Space: O(1)

[Better Approach] Using Hashing – O(M+N) Time and O(M+N) Space

*The idea is to use hashing to store all the nodes of the first list in a **hash set** and then iterate over second list checking if the node is present in the set. If we find a node which is present in the hash set, we return the node.*

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript

```python
# Pyhton program to get intersection point of two linked
# list using hashing

class Node:
    def __init__(self, new_data):
        self.data = new_data
        self.next = None

# Function to get the intersection point of two linked lists
def get_intersection_node(head1, head2):
    visited_nodes = set()

    # Traverse the first list and store all nodes in a set
    curr1 = head1
    while curr1 is not None:
        visited_nodes.add(curr1)
        curr1 = curr1.next

    # Traverse the second list and check if any node is in the set
    curr2 = head2
    while curr2 is not None:
        if curr2 in visited_nodes:
            return curr2  # Intersection point found
        curr2 = curr2.next

    return None


if __name__ == "__main__":
    # Create two linked lists
    # 1st list: 10 -> 15 -> 30
    # 2nd list: 3 -> 6 -> 9 -> 15 -> 30
    # 15 is the intersection point

    # Creating the first linked list
    head1 = Node(10)
    head1.next = Node(15)
    head1.next.next = Node(30)

    # Creating the second linked list
    head2 = Node(3)
    head2.next = Node(6)
    head2.next.next = Node(9)
    head2.next.next.next = head1.next

    # Finding the intersection point
    intersection_point = get_intersection_node(head1, head2)

    if intersection_point is None:
        print("No Intersection Point")
    else:
        print("Intersection Point:", intersection_point.data)
```

Output

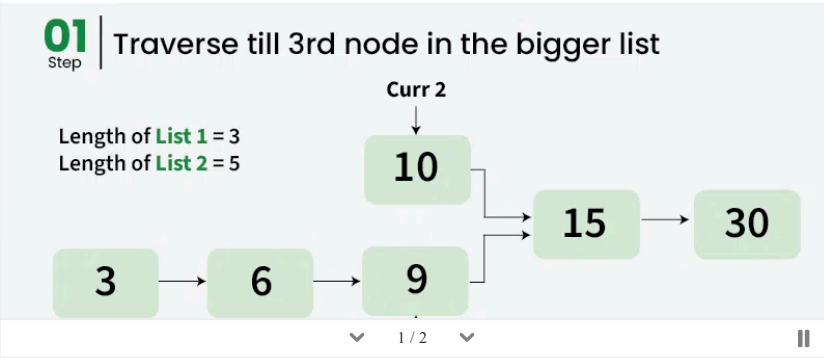
```
Intersection Point: 15
```

Time complexity: O(M + N), where **M** and **N** are the lengths of the two lists.
Auxiliary Space: O(M + N)

[Expected Approach 1] – Using difference in node counts – O(M + N) Time and O(1) Space

*The idea is to find the **difference (d)** between the count of nodes in both the lists. Initialize the start pointer in both the lists and then increment the start pointer of the longer list by D nodes. Now, we have both start pointers at the same distance from the intersection point, so we can keep incrementing both the start pointers until we find the intersection point.*

Below is the Illustration of above approach:



Below is the implementation of the above approach :

C++ C Java Python C# JavaScript

```python
# Python program to get intersection point of two linked list
# using count of nodes

class Node:
    def __init__(self, x):
        self.data = x
        self.next = None

# Function to get the count of nodes in a linked list
def get_count(head):
    count = 0
    curr = head
    while curr is not None:
        count += 1
        curr = curr.next
    return count

# Function to get the intersection point of two
# linked lists where head1 has d more nodes than head2
def _get_intersection_node(d, head1, head2):
    curr1 = head1
    curr2 = head2

    # Move the pointer forward by d nodes
    for _ in range(d):
        if curr1 is None:
            return None
        curr1 = curr1.next

    # Move both pointers until they intersect
    while curr1 is not None and curr2 is not None:
        if curr1 == curr2:
            return curr1
        curr1 = curr1.next
        curr2 = curr2.next

    return None

# Function to get the intersection point of two linked lists
def get_intersection_node(head1, head2):

    # Count the number of nodes in both linked lists
    len1 = get_count(head1)
    len2 = get_count(head2)

    diff = 0

    # If the first list is longer
    if len1 > len2:
        diff = len1 - len2
        return _get_intersection_node(diff, head1, head2)
    else:
        diff = len2 - len1
        return _get_intersection_node(diff, head2, head1)

if __name__ == "__main__":

    # Create two linked lists
    # 1st list: 10 -> 15 -> 30
    # 2nd list: 3 -> 6 -> 9 -> 15 -> 30
    # 15 is the intersection point

    # creation of first list
    head1 = Node(10)
    head1.next = Node(15)
    head1.next.next = Node(30)

    # creation of second list
    head2 = Node(3)
    head2.next = Node(6)
    head2.next.next = Node(9)
    head2.next.next.next = head1.next

    intersection_point = get_intersection_node(head1, head2)

    if intersection_point is None:
        print("No Intersection Point")
    else:
        print(f"Intersection Point: {intersection_point.data}")
```

Output

```
The node of intersection is 15
```

Time complexity: O(M+N), where M and N are the lengths of the two lists.
Auxiliary Space: O(1)

[Expected Approach 2] Using Two Pointer Technique – O(M+N) Time and O(1) Space

*This algorithm works by traversing the two linked lists **simultaneously**, using **two** pointers. When one pointer reaches the end of its list, it is reassigned to the **head** of the other list. This process continues until the two pointers meet, which indicates that they have reached the **intersection point**.*

Follow the steps below to solve the problem:

- Initialize two pointers **ptr1** and **ptr2** at **head1** and **head2** respectively.
- Traverse through the lists, one node at a time.
    - When **ptr1** reaches the end of a list, then redirect it to **head2**.
    - Similarly, when **ptr2** reaches the end of a list, redirect it to the **head1**.
    - Once both of them go through reassigning, they will be at equal distance from the collision point.
    - If at any node **ptr1** meets **ptr2**, then it is the intersection node.
- After the second iteration if there is no intersection node , returns **NULL**.

Below is the implementation of the above approach:

C++ C Java Python C# JavaScript

```python
# Python program to find the intersection point of two
# linked lists using Two Pointers Technique


class Node:
    def __init__(self, x):
        self.data = x
        self.next = None

def get_intersection_node(head1, head2):

    # Maintain two pointers ptr1 and ptr2 at the
    # heads of the lists
    ptr1 = head1
    ptr2 = head2

    # If either of the heads is None, there is no intersection
    if not ptr1 or not ptr2:
        return None

    # Traverse through the lists until both pointers meet
    while ptr1 != ptr2:

        # Move to the next node in each list and if one
        # pointer reaches None, start from the other linked list
        ptr1 = ptr1.next if ptr1 else head2
        ptr2 = ptr2.next if ptr2 else head1

    # Return the intersection node, or None if no intersection
    return ptr1

def print_list(node):
    while node:
        print(node.data, end=" ")
        node = node.next
    print()

# Main function to test the implementation
if __name__ == "__main__":

    # Create two linked lists
    # 1st list: 10 -> 15 -> 30
    # 2nd list: 3 -> 6 -> 9 -> 15 -> 30
    # 15 is the intersection point

    # Creation of the first list
    head1 = Node(10)
    head1.next = Node(15)
    head1.next.next = Node(30)

    # Creation of the second list
    head2 = Node(3)
    head2.next = Node(6)
    head2.next.next = Node(9)
    head2.next.next.next = head1.next

    intersection_point = get_intersection_node(head1, head2)

    if intersection_point is None:
        print("No Intersection Point")
    else:
        print(f"Intersection Point: {intersection_point.data}")
```

Output

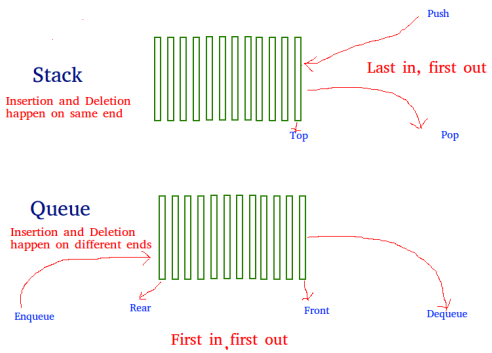
```
Intersection Point = 15
```

Time complexity: O(M + N), where **M** and **N** are the lengths of the two lists.
Auxiliary Space: O(1)

Queue using Stacks

Last Updated : 10 Jul, 2023

The problem is opposite of this post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.



A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly): This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q, x):

- *While stack1 is not empty, push everything from stack1 to stack2.*
- *Push x to stack1 (assuming size of stacks is unlimited).*
- *Push everything back to stack1.*

Here time complexity will be O(n)

deQueue(q):

- *If stack1 is empty then error*
- *Pop an item from stack1 and return it*

Here time complexity will be O(1)

Below is the implementation of the above approach:

Recommended Problem

Queue using stack

Topics: Stack Queue +2 more

Companies: Microsoft

Solve Problem

Easy 73.87% 24.9K

C++ Java Python3 C# Javascript

```python
# Python3 program to implement Queue using
# two stacks with costly enQueue()

class Queue:
    def __init__(self):
        self.s1 = []
        self.s2 = []

    def enQueue(self, x):

        # Move all elements from s1 to s2
        while len(self.s1) != 0:
            self.s2.append(self.s1[-1])
            self.s1.pop()

        # Push item into self.s1
        self.s1.append(x)

        # Push everything back to s1
        while len(self.s2) != 0:
            self.s1.append(self.s2[-1])
            self.s2.pop()

    # Dequeue an item from the queue
    def deQueue(self):

            # if first stack is empty
        if len(self.s1) == 0:
            return -1;

        # Return top of self.s1
        x = self.s1[-1]
        self.s1.pop()
        return x
```

```
# Driver code
if __name__ == '__main__':
    q = Queue()
    q.enQueue(1)
    q.enQueue(2)
    q.enQueue(3)

    print(q.deQueue())
    print(q.deQueue())
    print(q.deQueue())

# This code is contributed by PranchalK
```

Output:

```
1
2
3
```

Complexity Analysis:

- **Time Complexity:**
  - **Push operation:** O(N).
    In the worst case we have empty whole of stack 1 into stack 2.
  - **Pop operation:** O(1).
    Same as pop operation in stack.
- **Auxiliary Space:** O(N).
  Use of stack for storing values.

Method 2 (By making deQueue operation costly): In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enQueue(q,  x)
  1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from stack1 to stack2.
  3) Pop the element from stack2 and return it.
Here time complexity will be O(n)
```

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty. So, the amortized complexity of the dequeue operation becomes $\Theta(1)$

Implementation of method 2:

C++ C Java Python3 C# Javascript

```
# Python3 program to implement Queue using
# two stacks with costly deQueue()

class Queue:
    def __init__(self):
        self.s1 = []
        self.s2 = []

    # EnQueue item to the queue
    def enQueue(self, x):
        self.s1.append(x)

    # DeQueue item from the queue
    def deQueue(self):

        # if both the stacks are empty
        if len(self.s1) == 0 and len(self.s2) == 0:
            return -1

        # if s2 is empty and s1 has elements
        elif len(self.s2) == 0 and len(self.s1) > 0:
            while len(self.s1):
                temp = self.s1.pop()
                self.s2.append(temp)
            return self.s2.pop()

        else:
            return self.s2.pop()

    # Driver code
if __name__ == '__main__':
    q = Queue()
    q.enQueue(1)
    q.enQueue(2)
    q.enQueue(3)
```

```
    print(q.deQueue())
    print(q.deQueue())
    print(q.deQueue())

# This code is contributed by Pratyush Kumar
```

Output:

```
1 2 3
```

Complexity Analysis:

- **Time Complexity:**
  - **Push operation:** O(1).
    Same as pop operation in stack.
  - **Pop operation:** O(N) in general and O(1) amortized time complexity.
    In the worst case we have to empty the whole of stack 1 into stack 2 so its O(N). Amortized time is the way to express the time complexity when an algorithm has the very bad time complexity only once in a while besides the time complexity that happens most of time. So its O(1) amortized time complexity, since we have to empty whole of stack 1 only when stack 2 is empty, rest of the times the pop operation takes O(1) time.
- **Auxiliary Space:** O(N).
  Use of stack for storing values.

Queue can also be implemented using one user stack and one Function Call Stack. Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```
enQueue(x)
  1) Push x to stack1.
```

```
deQueue:
  1) If stack1 is empty then error.
  2) If stack1 has only one element then return it.
  3) Recursively pop everything from the stack1, store the popped item
    in a variable res,  push the res back to stack1 and return res
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in deQueue() and all other items are pushed back in step

3. Implementation of method 2 using Function Call Stack:

C++ C Java Python3 C# Javascript

```
# Python3 program to implement Queue using
# one stack and recursive call stack.
class Queue:
    def __init__(self):
        self.s = []

    # Enqueue an item to the queue
    def enQueue(self, data):
        self.s.append(data)

    # Dequeue an item from the queue
    def deQueue(self):
        # Return if queue is empty
        if len(self.s) <= 0:
            return -1

        # pop an item from the stack
        x = self.s[len(self.s) - 1]
        self.s.pop()

        # if stack become empty
        # return the popped item
        if len(self.s) <= 0:
            return x

        # recursive call
        item = self.deQueue()

        # push popped item back to
        # the stack
        self.s.append(x)

        # return the result of
        # deQueue() call
        return item

# Driver code
if __name__ == '__main__':
    q = Queue()
    q.enQueue(1)
    q.enQueue(2)
    q.enQueue(3)

    print(q.deQueue())
```

```
    print(q.deQueue())
    print(q.deQueue())

# This code is contributed by iArman
```

Output:

```
1 2 3
```

Complexity Analysis:

- **Time Complexity:**
    - **Push operation :** O(1).
      Same as pop operation in stack.
    - **Pop operation :** O(N).
      The difference from above method is that in this method element is returned and all elements are restored back in a single call.
- **Auxiliary Space:** O(N).
  Use of stack for storing values.

Merge Sort – Data Structure and Algorithms Tutorials

Last Updated : 06 Aug, 2024

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Merge Sort Algorithm

Table of Content

How does Merge Sort work?

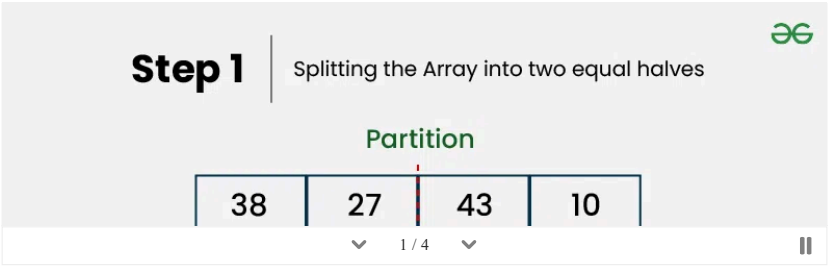
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

Let's sort the array or list **[38, 27, 43, 10]** using Merge Sort



Let's look at the working of above example:

Divide:

- ***[38, 27, 43, 10]*** *is divided into* ***[38, 27]*** *and* ***[43, 10]*** .
- ***[38, 27]*** *is divided into* ***[38]*** *and* ***[27]*** .
- ***[43, 10]*** *is divided into* ***[43]*** *and* ***[10]*** .

Conquer:

- ***[38]*** *is already sorted.*
- ***[27]*** *is already sorted.*
- ***[43]*** *is already sorted.*
- ***[10]*** *is already sorted.*

Merge:

- *Merge* ***[38]*** *and* ***[27]*** *to get* ***[27, 38]*** .
- *Merge* ***[43]*** *and* ***[10]*** *to get* ***[10,43]*** .
- *Merge* ***[27, 38]*** *and* ***[10,43]*** *to get the final sorted list* ***[10, 27, 38, 43]***

Therefore, the sorted list is ***[10, 27, 38, 43]*** .

Implementation of Merge Sort:

Recommended Problem

Merge Sort

Topics: Divide And Conquer, Sorting, +1 more

Companies: Paytm, Amazon, +10 more

Solve Problem

Medium 54.1% 1.8L

C++ C Java Python C# JavaScript PHP

```python
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = left  # Initial index of merged subarray

    # Merge the temp arrays back
    # into arr[left..right]
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[],
    # if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[],
    # if there are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

def print_list(arr):
    for i in arr:
        print(i, end=" ")
    print()

# Driver code
if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is")
    print_list(arr)

    merge_sort(arr, 0, len(arr) - 1)

    print("\nSorted array is")
    print_list(arr)
```

Output

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```

Recurrence Relation of Merge Sort:

The recurrence relation of merge sort is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- T(n) Represents the total time time taken by the algorithm to sort an array of size n.
- 2T(n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).
- O(n) represents the time taken to merge the two sorted halves

Complexity Analysis of Merge Sort:

- **Time Complexity:**
    - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
    - **Average Case:** O(n log n), When the array is randomly ordered.
    - **Worst Case:** O(n log n), When the array is sorted in reverse order.

- **Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging.

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting
- Merge Sort and its variations are used in library methods of programming languages. For example its variation TimSort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort. For example Arrays.sort in Java uses QuickSort while Collections.sort uses MergeSort.
- It is a preferred algorithm for sorting Linked lists.
- It can be easily parallelized as we can independently sort subarrays and then merge.
- The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays.

Advantages of Merge Sort:

- **Stability** : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of **O(N logN)** , which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.
- **Naturally Parallel** : We independently merge subarrays that makes it suitable for parallel processing.

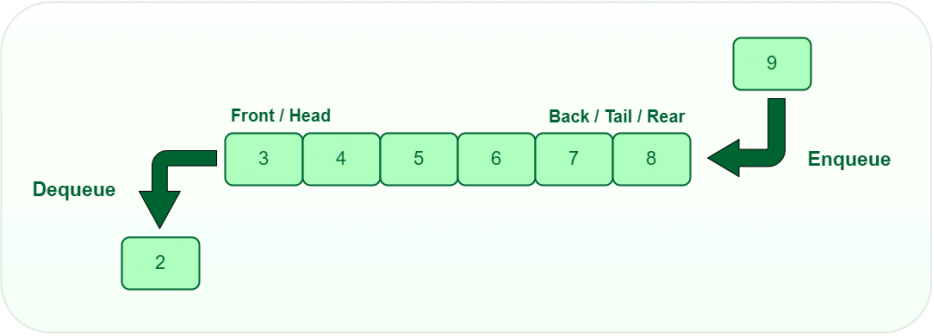
Disadvantages of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- S**lower than QuickSort in general**. QuickSort is more cache friendly because it works in-place.

Introduction and Array Implementation of Queue

Last Updated : 25 May, 2023

Similar to Stack, Queue is a linear data structure that follows a particular order in which the operations are performed for storing data. The order is First In First Out **(FIFO)**. One can imagine a queue as a line of people waiting to receive something in sequential order which starts from the beginning of the line. It is an ordered list in which insertions are done at one end which is known as the rear and deletions are done from the other end known as the front. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Queue Data structure

Basic Operations on Queue:

- **enqueue():** Inserts an element at the end of the queue i.e. at the rear end.
- **dequeue():** This operation removes and returns an element that is at the front end of the queue.
- **front():** This operation returns the element at the front end without removing it.
- **rear():** This operation returns the element at the rear end without removing it.
- **isEmpty():** This operation indicates whether the queue is empty or not.
- **isFull():** This operation indicates whether the queue is full or not.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Types of Queues:

- **Simple Queue:** Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end. Here problem is that if we pop some item from front and then rear reach to the capacity of the queue and although there are empty spaces before front means the queue is not full but as per condition in isFull() function, it will show that the queue is full then. To solve this problem we use circular queue.
- **Circular Queue:** In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position using modulo capacity(*%n*).
- **Priority Queue:** This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority. The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values. The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values. In pre-define priority queue, C++ gives priority to highest value whereas Java gives priority to lowest value.
- **Dequeue:** Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both ends of the queue, unlike the other queues in which it can be done only from one end. Because of this property, it may not obey the First In First Out property.

Applications of Queue:

Queue is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
- Queue can be used as an essential component in various other data structures.

Array implementation Of Queue:

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in circular manner.

Steps for enqueue:

1. Check the queue is full or not
2. If full, print overflow and exit
3. If queue is not full, increment tail and add the element

Steps for dequeue:

1. Check queue is empty or not
2. if empty, print underflow and exit
3. if not empty, print element at the head and increment head

Below is a program to implement above operation on queue

Recommended Problem

Implement Queue using array

Topics: Arrays Queue +1 more

Companies: Amazon Goldman Sachs

Solve Problem

Basic 47.24% 1.6L

C++ C Java Python3 C# Javascript

```python
# Python3 program for array implementation of queue

# Class Queue to represent a queue
class Queue:

    # __init__ function
    def __init__(self, capacity):
        self.front = self.size = 0
        self.rear = capacity -1
        self.Q = [None]*capacity
        self.capacity = capacity

    # Queue is full when size becomes
    # equal to the capacity
    def isFull(self):
        return self.size == self.capacity

    # Queue is empty when size is 0
    def isEmpty(self):
        return self.size == 0

    # Function to add an item to the queue.
    # It changes rear and size
    def EnQueue(self, item):
        if self.isFull():
            print("Full")
            return
        self.rear = (self.rear + 1) % (self.capacity)
        self.Q[self.rear] = item
        self.size = self.size + 1
        print("% s enqueued to queue"  % str(item))

    # Function to remove an item from queue.
    # It changes front and size
    def DeQueue(self):
        if self.isEmpty():
            print("Empty")
            return

        print("% s dequeued from queue" % str(self.Q[self.front]))
        self.front = (self.front + 1) % (self.capacity)
        self.size = self.size -1

    # Function to get front of queue
    def que_front(self):
        if self.isEmpty():
            print("Queue is empty")

        print("Front item is", self.Q[self.front])

    # Function to get rear of queue
    def que_rear(self):
        if self.isEmpty():
            print("Queue is empty")
        print("Rear item is",  self.Q[self.rear])


# Driver Code
if __name__ == '__main__':

    queue = Queue(30)
    queue.EnQueue(10)
    queue.EnQueue(20)
    queue.EnQueue(30)
    queue.EnQueue(40)
    queue.DeQueue()
    queue.que_front()
```

```
queue.que_rear()
```

Output

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue
Front item is 20
Rear item is 40
```

Complexity Analysis:

- **Time Complexity**

Operations	Complexity
Enqueue(insertion)	O(1)
Deque(deletion)	O(1)
Front(Get front)	O(1)
Rear(Get Rear)	O(1)
IsFull(Check queue is full or not)	O(1)
IsEmpty(Check queue is empty or not)	O(1)

- **Auxiliary Space:**
  O(N) where N is the size of the array for storing elements.

Advantages of Array Implementation:

- Easy to implement.
- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.

Disadvantages of Array Implementation:

- Static Data Structure, fixed size.
- If the queue has a large number of enqueue and dequeue operations, at some point (in case of linear increment of front and rear indexes) we may not be able to insert elements in the queue even if the queue is empty (this problem is avoided by using circular queue).
- Maximum size of a queue must be defined prior.

Quick Sort Algorithm

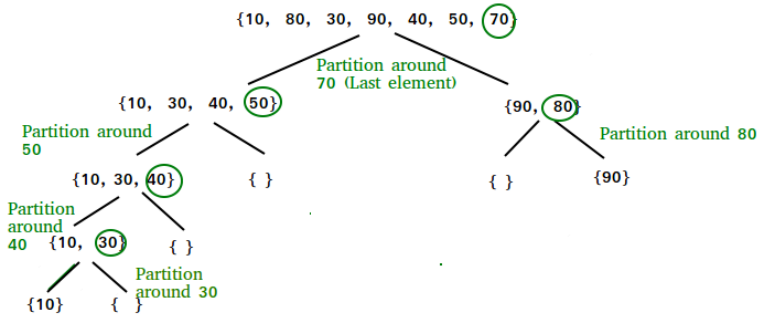
Last Updated : 11 Aug, 2024

QuickSort *is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.*

How does QuickSort work?

The key process in ***quickSort*** *is a* ***partition()*** *. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.*

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



How Quicksort works

Choice of Pivot:

There are many different choices for picking pivots.

- Always pick the first element as a pivot .
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot .
- Pick the middle as the pivot.

Partition Algorithm:

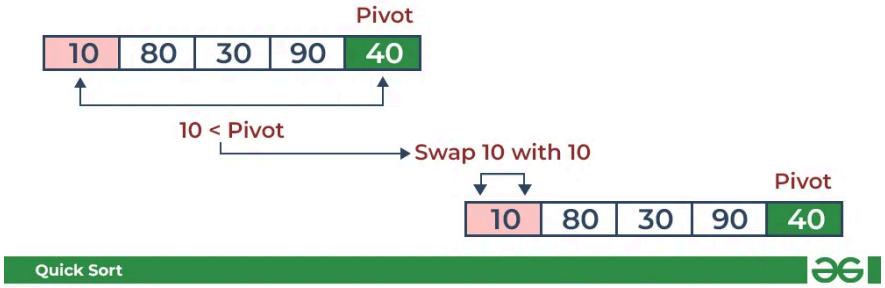
There exist two main algorithms for partitioning Lomuto and Hoare. In this post, we have considered Lomuto as it is easy to understand and moves pivot to its correct position. However Hoare's is faster than Lomuto.

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as ***i*** *. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.*

Let us understand the working of partition and the Quick Sort algorithm with the help of the following example:

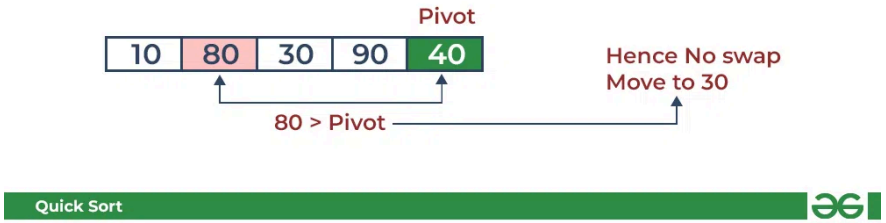
Consider: arr[] = {10, 80, 30, 90, 40}.

- *Compare 10 with the pivot and as it is less than pivot arrange it accordingly.*



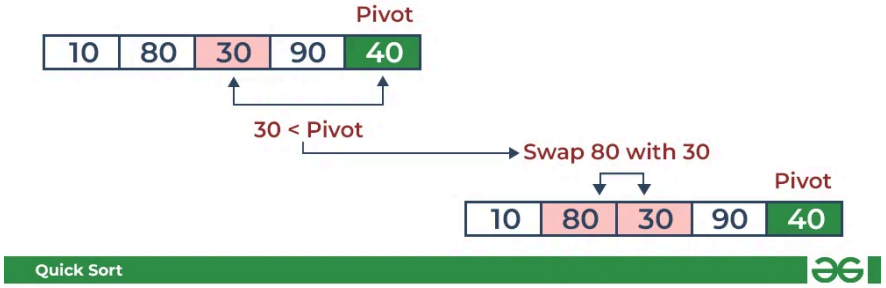
Partition in QuickSort: Compare pivot with 10

- *Compare 80 with the pivot. It is greater than pivot.*



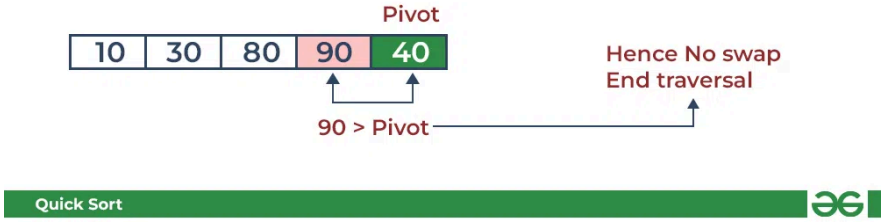
Partition in QuickSort: Compare pivot with 80

- *Compare 30 with pivot. It is less than pivot so arrange it accordingly.*



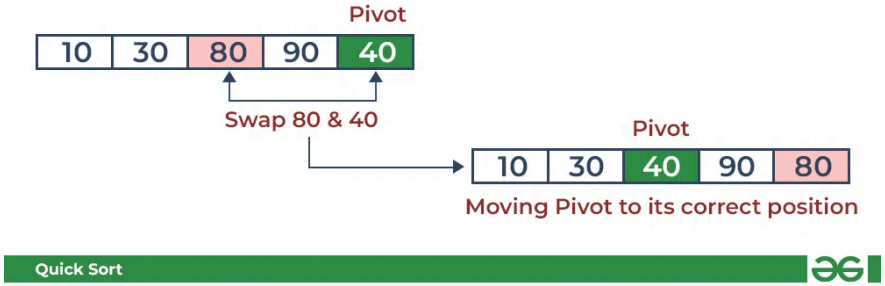
Partition in QuickSort: Compare pivot with 30

- *Compare 90 with the pivot. It is greater than the pivot.*



Partition in QuickSort: Compare pivot with 90

- *Arrange the pivot in its correct position.*



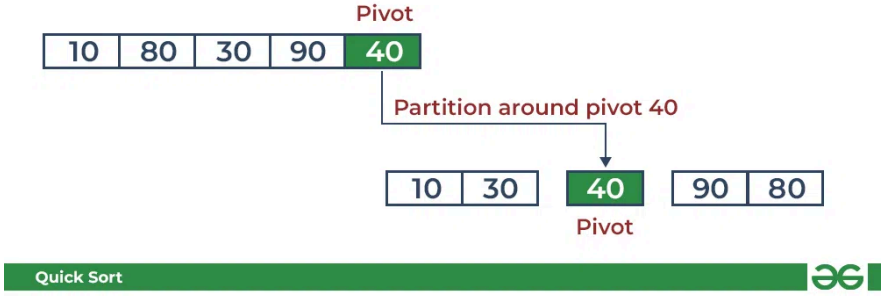
Partition in QuickSort: Place pivot in its correct position

Illustration of Quicksort:

As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.

Follow the below images to understand how the recursive implementation of the partition algorithm helps to sort the array.

- *Initial partition on the main array:*



Quicksort: Performing the partition

- *Partitioning of the subarrays:*

Quicksort: Performing the partition

Quick Sort is a crucial algorithm in the industry, but there are other sorting algorithms that may be more optimal in different cases. To gain a deeper understanding of sorting and other essential algorithms, check out our course **Tech Interview 101 – From DSA to System Design** . This course covers almost every standard algorithm and more.

Code implementation of the Quick Sort:

Recommended Problem

Quick Sort

Topics: Divide And Conquer Sorting +1 more

Companies: VMWare Amazon +11 more

Solve Problem

Medium 55.23% 2.1L

C++ C Java Python C# JavaScript PHP

```python
def partition(arr, low, high):

    # Choose the pivot
    pivot = arr[high]

    i = low - 1

    # Traverse arr[low..high] and move all smaller
    # elements on the left side. Elements from low to
    # i are smaller after every iteration
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Move pivot after smaller elements and
    # return its position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# The QuickSort function implementation
def quick_sort(arr, low, high):
    if low < high:
        # pi is the partition return index of pivot
        pi = partition(arr, low, high)

        # Recursion calls for smaller elements
        # and greater or equals elements
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Function to print an array
def print_array(arr):
    for i in arr:
        print(i, end=" ")
    print()

# Driver code
if __name__ == "__main__":
    arr = [10, 7, 8, 9, 1, 5]
    print("Given array is")
    print_array(arr)

    quick_sort(arr, 0, len(arr) - 1)

    print("\nSorted array is")
    print_array(arr)
```

Output

```
Sorted Array
1 5 7 8 9 10
```

Complexity Analysis of Quick Sort :

Time Complexity:

- **Best Case** : Ω (N log (N))
  The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- **Average Case:** θ (N log (N))
  Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

- **Worst Case:** O(N ^ 2)
  The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.
- **Auxiliary Space:** O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make $O(N)$.

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
- It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
- Fastest general purpose algorithm for large data when stability is not required.
- It is **tail recursive** and hence all the tail call optimization can be done.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Flattening a Linked List

Last Updated : 02 Aug, 2024

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:

- **next** pointer to the next node
- **bottom** pointer to a linked list where this node is head.

Each of the sub-linked-list is in sorted order. Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.

Note: The flattened list will be printed using the bottom pointer instead of next pointer.

Examples:

> ***Input***:
>
> ***Output:*** *5->7->8->10->19->22->28->30->50*
>
> ***Input:***
>
> ***Output:*** *5->7->8->10->19->20->22->28->30->35->40->45->50*

Table of Content

Flattening a Linked List by Sorting nodes in array – O(N * M * log(N * M)) Time and O(N * M) Space

> *The idea is to traverse the linked list and push values of all the nodes in an array. Now, we can sort the array in ascending order, and create a new linked list by traversing the sorted array. While creating the new linked list, we will append the nodes using the bottom pointer of flattened linked list.*

Recommended Problem

Flattening a Linked List

Topics: Linked List, Data Structures

Companies: Paytm, Flipkart, +9 more

Solve Problem

Medium 51.53% 1.5L

C++ C Java Python C# JavaScript

```python
class Node:
    def __init__(self, new_data):
        self.data = new_data
        self.next = None
        self.bottom = None

# Function to flatten the linked list
def flatten(root):
    values = []

    # Push values of all nodes into an array
    while root is not None:
        # Push the head node of the sub-linked-list
        values.append(root.data)

        # Push all the nodes of the sub-linked-list
        temp = root.bottom
        while temp is not None:
            values.append(temp.data)
            temp = temp.bottom

        # Move to the next head node
        root = root.next

    # Sort the node values in ascending order
    values.sort()

    # Construct the new flattened linked list
    tail = None
    head = None
    for value in values:
        newNode = Node(value)

        # If this is the first node of the linked list,
        # make the node as head
        if head is None:
            head = newNode
        else:
            tail.bottom = newNode
        tail = newNode

    return head

def print_list(head):
    temp = head
    while temp is not None:
```

```
            print(temp.data, end=" ")
            temp = temp.bottom
        print()

if __name__ == "__main__":

    # Create a hard-coded linked list:
    # 5 -> 10 -> 19 -> 28
    # |    |     |
    # V    V     V
    # 7    20    22
    # |          |
    # V          V
    # 8          50
    # |
    # V
    # 30
    head = Node(5)
    head.bottom = Node(7)
    head.bottom.bottom = Node(8)
    head.bottom.bottom.bottom = Node(30)

    head.next = Node(10)
    head.next.bottom = Node(20)

    head.next.next = Node(19)
    head.next.next.bottom = Node(22)
    head.next.next.bottom.bottom = Node(50)

    head.next.next.next = Node(28)

    # Function call
    head = flatten(head)

    print_list(head)
```

Output

```
5 7 8 10 19 20 22 28 30 50
```

Flattening a Linked List by Merging lists recursively – O(N * N * M) Time and O(N) Space

*The idea is similar to Merge two sorted linked lists. Lets assume that we have only two linked list, **head1** and **head2** which we need to flatten. We can maintain a dummy node as the start of the result list and a pointer, say **tail** which always points to the last node in the result list, so appending new nodes is easy. Initially tail node points to the dummy node. Now, traverse till both either head1 or head2 does not reach NULL.*

- *If **value of head1 is smaller than or equal to value of head2**, then, append head1 after the tail node and update tail to head1.*
- *Similarly, if **value of head2 is smaller than value of head1**, then append head2 after the tail node and update tail to head2.*

When either head1 or head2 reaches NULL, simply append the remaining nodes after the tail node.

We can use the above idea to recursively merge the linked list, starting from the last two sub-linked-lists. At each node, we can have two cases:

- *If the node is the last node of linked list, simply return the head.*
- *Otherwise, call the recursive function with the next of head to get the flattened linked list and merge the flattened linked list with the current sub-linked-list.*

C++ C Java Python C# JavaScript

```
# Python3 program for flattening a Linked List

class Node:
    def __init__(self, new_data):
        self.data = new_data
        self.next = None
        self.bottom = None

# Utility function to merge two sorted linked lists
# using their bottom pointers
def merge(head1, head2):
    # A dummy first node to store the result list
    dummy = Node(-1)
    tail = dummy

    # Iterate till either head1 or head2 does not reach None
    while head1 and head2:
        if head1.data <= head2.data:
            # Append head1 to the result
            tail.bottom = head1
            head1 = head1.bottom
        else:
            # Append head2 to the result
            tail.bottom = head2
            head2 = head2.bottom

        # Move tail pointer to the next node
        tail = tail.bottom

    # Append the remaining nodes of the non-null linked list
    if head1:
        tail.bottom = head1
    else:
```

```
            tail.bottom = head2

        return dummy.bottom

    # Function to flatten the linked list
    def flatten(root):
        # Base Cases
        if root is None or root.next is None:
            return root

        # Recur for next list
        root.next = flatten(root.next)

        # Now merge the current and next list
        root = merge(root, root.next)

        # Return the root
        return root

    def print_list(head):
        temp = head
        while temp is not None:
            print(temp.data, end=" ")
            temp = temp.bottom
        print()

    if __name__ == "__main__":

        # Create a hard-coded linked list:
        #     5 -> 10 -> 19 -> 28
        #     |    |     |
        #     V    V     V
        #     7    20    22
        #     |          |
        #     V          V
        #     8          50
        #     |
        #     V
        #     30

        head = Node(5)
        head.bottom = Node(7)
        head.bottom.bottom = Node(8)
        head.bottom.bottom.bottom = Node(30)

        head.next = Node(10)
        head.next.bottom = Node(20)

        head.next.next = Node(19)
        head.next.next.bottom = Node(22)
        head.next.next.bottom.bottom = Node(50)

        head.next.next.next = Node(28)

        # Function call
        head = flatten(head)

        print_list(head)
```

Output

```
5 7 8 10 19 20 22 28 30 50
```

Time Complexity: O(N * N * M) – where **N** is the no of nodes in the main linked list and **M** is the no of nodes in a single sub-linked list.

- After adding the first 2 lists, the time taken will be O(M+M) = O(2M).
- Then we will merge another list to above merged list -> time = O(2M + M) = O(3M).
- We will keep merging lists to previously merged lists until all lists are merged.
- Total time taken will be O(2M + 3M + 4M + …. N*M) = (2 + 3 + 4 + … + N) * M = **O(N * N * M)**

Auxiliary Space: O(N), the recursive functions will use a recursive stack of a size equivalent to a total number of nodes in the main linked list.

Flattening a Linked List using Priority Queues – O(N * M * log(N)) Time and O(N) Space

> *The idea is to use a Min Heap and push all the nodes that are on the first level into it. While the priority queue is not empty, pop the smallest element from the priority queue and append it to the flattened linked list. Also, check if there is a node at the bottom of the popped node then push that bottom node to the priority queue. Since all the sub-linked -lists are sorted, all the nodes will be popped in sorted order.*

C++ Java Python C# JavaScript

```
    # Python program for flattening a Linked List

    from heapq import heappush, heappop

    class Node:
        def __init__(self, d):
            self.data = d
            self.next = self.bottom = None


    # Utility function to insert a node at beginning of the
    # linked list
    def push(head, data):

        # 1 & 2: Allocate the Node &
        # Put in the data
```

```
    new_node = Node(data)

    # Make next of new Node as head
    new_node.bottom = head_ref

    # 4. Move the head to point to new Node
    head_ref = new_node

    # 5. return to link it back
    return head_ref


def printList(head):

    temp = head
    while(temp != None):
        print(temp.data, end=" ")
        temp = temp.bottom
    print()


# class to compare two node objects
class Cmp:
    def __init__(self, node):
        self.node = node

    def __lt__(self, other):
        return self.node.data < other.node.data


def flatten(root):
    pq = []
    head = None
    tail = None

    # pushing main link nodes into priority_queue
    while root:
        heappush(pq, Cmp(root))
        root = root.next

    # Extracting the minimum node while the priority queue is not empty
    while pq:
        minNode = heappop(pq).node

        if head is None:
            head = minNode
            tail = minNode
        else:
            tail.bottom = minNode
            tail = tail.bottom

        # If we have another node at the bottom of the
        # popped node, push that node into the priority
        # queue
        if minNode.bottom:
            heappush(pq, Cmp(minNode.bottom))
            minNode.bottom = None

    return head
```

```
if __name__ == '__main__':
    head = Node(5)
    head.bottom = Node(7)
    head.bottom.bottom = Node(8)
    head.bottom.bottom.bottom = Node(30)

    head.next = Node(10)
    head.next.bottom = Node(20)

    head.next.next = Node(19)
    head.next.next.bottom = Node(22)
    head.next.next.bottom.bottom = Node(50)

    head.next.next.next = Node(28)

    # Function call
    head = flatten(head)

    printList(head)
```

Output

```
5 7 8 10 19 20 22 28 30 50
```

Time Complexity: O(N * M * log(N)) – where N is the no of nodes in the main linked list (reachable using the next pointer) and M is the no of nodes in a single sub-linked list (reachable using a bottom pointer).
Auxiliary Space: O(N)

Application and uses of Quicksort

Last Updated : 06 Aug, 2024

Quicksort: Quick sort is a Divide Conquer algorithm and the fastest sorting algorithm. In quick sort, it creates two empty arrays to hold elements less than the pivot element and the element greater than the pivot element and then recursively sort the sub-arrays. There are many versions of Quicksort that pick pivot in different ways:

- Always pick the first element as pivot.
- Always pick the last element as pivot.
- Pick a random element as a pivot.
- Pick median as a pivot.

The principle of the Quicksort algorithm is given below:

- Select an element as **pivot**.
- Split the array into 3 parts: by following the below-given rules:
    - First part: All elements in this part should less than the pivot element.
    - Second part: The single element i.e. the pivot element.
    - Third part: All elements in this part should greater than or equal to the pivot element.
- Then, applying this algorithm to the first and the third part (recursively).

Below given the uses and real-time application of Quicksort:

- **Commercial Computing** is used in various government and private organizations for the purpose of sorting various data like sorting files by name/date/price, sorting of students by their roll no., sorting of account profile by given id, etc as it is the fastest general purpose algorithm for large data.
- Since the two subarrays can be independently processed, we can easily do parallel processing.
- It is used almost everywhere where a **stable sort** is not needed. For stability, Merge Sort is typically used.
- Quicksort is a cache-friendly algorithm as it has a good locality of reference when used for arrays.
- It is **tail recursive** and hence all the tail call optimization can be done.
- It is an in-place sort that does not require any extra storage memory for storing elements (unlike Merge Sort that required extra space for merging)
- Numerical computations and in scientific research, for accuracy in calculations most of the efficiently developed algorithm uses priority queue and quick sort is used for sorting.
- Variants of Quicksort are used to separate the **Kth smallest** or **largest elements**.
- It is used to implement primitive type methods.
- If data is sorted then the search for information became easy and efficient using algorithms like binary search.
- QuickSort is typically implemented in library functions of programming languages to sort arrays. For example qsort in C, Arrays.sort in Java, sort in C++ use a Hybrid Sorting where QuickSort is primary sorting.

Merge Overlapping Intervals

Last Updated : 19 Oct, 2023

Given a set of time intervals in any order, our task is to merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals.

Example:

Input: *Intervals = {{1,3},{2,4},{6,8},{9,10}}*
Output: *{{1, 4}, {6, 8}, {9, 10}}*
Explanation: *Given intervals: [1,3],[2,4],[6,8],[9,10], we have only two overlapping intervals here, [1,3] and [2,4]. Therefore we will merge these two and return [1,4],[6,8], [9,10].*

Input: *Intervals = {{6,8},{1,9},{2,4},{4,7}}*
Output: *{{1, 9}}*

Brute Force Approach:

*A **simple approach** is to start from the first interval and compare it with all other intervals for overlapping, if it overlaps with any other interval, then remove the other interval from the list and merge the other into the first interval. Repeat the same steps for the remaining intervals after the first. This approach cannot be implemented in better than **O(n^2)** time.*

Merge Overlapping Intervals using Sorting (Optimized Approach):

*To solve this problem optimally we have to first sort the intervals according to the starting time. Once we have the sorted intervals, we can combine all intervals in a linear traversal. The idea is, in sorted array of intervals, if **interval[i]** doesn't overlap with **interval[i-1]**, then **interval[i+1]** cannot overlap with **interval[i-1]** because starting time of **interval[i+1]** must be greater than or equal to **interval[i]**.*

Follow the steps mentioned below to implement the approach:

- Sort the intervals based on the increasing order of starting time.
- Push the first interval into a stack.
- For each interval do the following:
  - If the current interval does not overlap with the top of the stack then, push the current interval into the stack.
  - If the current interval overlap with the top of the stack then, update the stack top with the ending time of the current interval.
- The end stack contains the merged intervals.

Below is the implementation of the above approach:

Recommended Problem

Overlapping Intervals

Topics: Arrays Hash +3 more

Companies: Amazon Microsoft +2 more

Solve Problem

Medium 57.41% 63.8K

C++ Java Python3 C# Javascript

```python
# Python3 program for merging overlapping intervals
def mergeIntervals(intervals):
    # Sort the array on the basis of start values of intervals.
    intervals.sort()
    stack = []
    # insert first interval into stack
    stack.append(intervals[0])
    for i in intervals[1:]:
        # Check for overlapping interval,
        # if interval overlap
        if stack[-1][0] <= i[0] <= stack[-1][-1]:
            stack[-1][-1] = max(stack[-1][-1], i[-1])
        else:
            stack.append(i)

    print("The Merged Intervals are :", end=" ")
    for i in range(len(stack)):
        print(stack[i], end=" ")


arr = [[6, 8], [1, 9], [2, 4], [4, 7]]
mergeIntervals(arr)
```

Output

```
The Merged Intervals are: [1, 9]
```

Time complexity: O(N*log(N))
Auxiliary Space: O(N)

Merge Overlapping Intervals using Sorting (Space Optimized):

The above solution requires O(n) extra space for the stack. We can avoid the use of extra space by doing merge operations in place. Below are detailed steps.

Follow the steps mentioned below to implement the approach:

- Sort all intervals in increasing order of start time.
- Traverse sorted intervals starting from the first interval,
- Do the following for every interval.
    - If the current interval is not the first interval and it overlaps with the previous interval, then merge it with the previous interval. Keep doing it while the interval overlaps with the previous one.
    - Otherwise, Add the current interval to the output list of intervals.

Below is the implementation of the above approach:

C++14 C Java Python3 C# Javascript

```python
# Python program to merge overlapping Intervals in
# O(n Log n) time and O(1) extra space


def mergeIntervals(arr):

    # Sorting based on the increasing order
    # of the start intervals
    arr.sort(key=lambda x: x[0])

    # Stores index of last element
    # in output array (modified arr[])
    index = 0

    # Traverse all input Intervals starting from
    # second interval
    for i in range(1, len(arr)):

        # If this is not first Interval and overlaps
        # with the previous one, Merge previous and
        # current Intervals
        if (arr[index][1] >= arr[i][0]):
            arr[index][1] = max(arr[index][1], arr[i][1])
        else:
            index = index + 1
            arr[index] = arr[i]

    print("The Merged Intervals are :", end=" ")
    for i in range(index+1):
        print(arr[i], end=" ")


# Driver code
arr = [[6, 8], [1, 9], [2, 4], [4, 7]]
mergeIntervals(arr)
```

Output

```
The Merged Intervals are: [1, 9]
```

Time Complexity: O(N*log(N))
Auxiliary Space Complexity: O(1)

Trapping Rain Water Problem – Tutorial with Illustrations

Last Updated : 25 Jun, 2024

Trapping Rainwater Problem states that given an array of N non-negative integers **arr[]** representing an elevation map where the width of each bar is **1**, compute how much water it can trap after rain.

Trapping Rainwater Problem

Examples:

Let us understand Trapping Rainwater problem with the help of some examples:

Input: *arr[] = {3, 0, 1, 0, 4, 0, 2}*
Output: *10*
Explanation: *The expected rainwater to be trapped is shown in the above image.*

Input: *arr[] = {3, 0, 2, 0, 4}*
Output: *7*
Explanation: *Structure is like below.*
We can trap "3 units" of water between 3 and 2,
"1 unit" on top of bar 2 and "3 units" between 2 and 4.

Table of Content

Intuition to solve the Trapping Rainwater Problem:

The basic intuition of the problem is as follows:

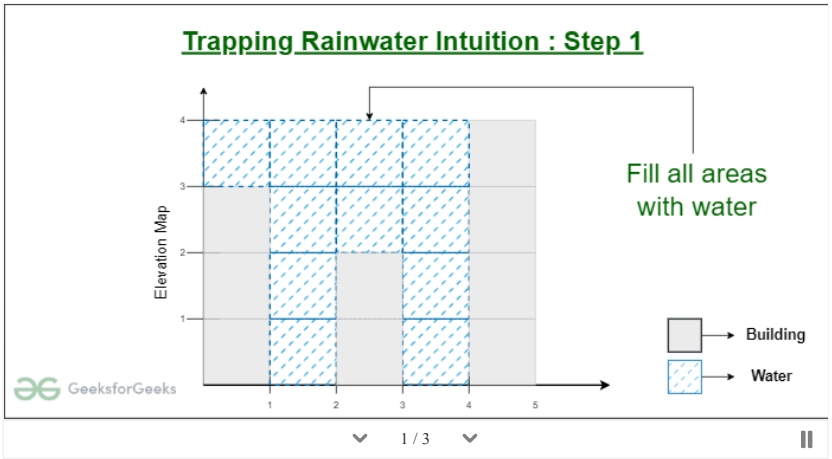
- An element of the array can store water if there are higher bars on the left and the right.
- The amount of water to be stored in every position can be found by finding the heights of bars on the left and right sides.
- The total amount of water stored is the summation of the water stored in each index.

Illustration to solve Trapping Rainwater Problem with help of basic Intuition

Consider the second example where array **arr[] = {3, 0, 2, 0, 4}**.

How to begin thinking about solution?

1. First thought is to fill all cells till the height and width of the elevation map
2. Then you need to eliminate the corner cases, i.e.:
   - No water can be filled if there is no boundary on both sides.
   - Water can only occupy the area which are not occupied with buildings
   - This means that the blocks marked as red in step 2, will get eliminated.
3. The remaining blocks of water, will give the required amount of rain water blocks that can trapped.



Therefore for above problem, Three units of water can be stored in two indexes 1 and 3, and one unit of water at index 2.
Water stored in each index = 0 + 3 + 1 + 3 + 0 = **7**

Approaches to Solve Trapping Rainwater Problem

Approach 1 – Brute Force Approach – O(N^2) Time and O(1) Space:

This approach is the **brute approach**. The idea is to:

> *Traverse every array element and find the highest bars on the left and right sides. Take the smaller of two heights. The difference between the smaller height and the height of the current element is the amount of water that can be stored in this array element.*

Follow the steps mentioned below to implement the idea:

- Traverse the array from start to end:
  - For every element:
    - Traverse the array from start to that index and find the maximum height *(a)* and
    - Traverse the array from the current index to the end, and find the maximum height *(b)*.
- The amount of water that will be stored in this column is min(a,b) – array[i], add this value to the total amount of water stored
- Print the total amount of water stored.

Below is the implementation of the above approach.

Recommended Practice

Trapping Rain Water

Try It!

Recommended Problem

Trapping Rain Water

Topics: Arrays, Dynamic Programming, +2 more

Companies: Flipkart, Amazon, +2 more

Solve Problem

Hard 33.14% 4.3L

C++ C Java Python C# JavaScript

```python
# Python3 implementation of the approach

# Function to return the maximum
# water that can be stored
def maxWater(arr, n):

    # To store the maximum water
    # that can be stored
    res = 0

    # For every element of the array
    for i in range(1, n - 1):

        # Find the maximum element on its left
        left = arr[i]
        for j in range(i):
            left = max(left, arr[j])

        # Find the maximum element on its right
        right = arr[i]

        for j in range(i + 1, n):
            right = max(right, arr[j])

        # Update the maximum water
        res = res + (min(left, right) - arr[i])

    return res

# Driver code
if __name__ == "__main__":

    arr = [0, 1, 0, 2, 1, 0,
           1, 3, 2, 1, 2, 1]
    n = len(arr)

    print(maxWater(arr, n))

# This code is contributed by AnkitRai01
```

Output

```
6
```

Complexity Analysis:

- **Time Complexity:** O(N^2). There are two nested loops traversing the array.
- **Auxiliary Space :** O(1). No extra space is required.

Approach 2 – Precalculation Approach – O(N) Time and O(N) Space:

This is an efficient solution based on the precalculation concept:

> *In previous approach, for every element we needed to calculate the highest element on the left and on the right.*

So, to reduce the time complexity:

- *For every element we can precalculate and store the highest bar on the left and on the right (say stored in arrays **left[]** and **right[]**).*
- *Then iterate the array and use the precalculated values to find the amount of water stored in this index,*
  *which is the same as (**min(left[i], right[i]) – arr[i]**)*

Follow the below illustration for a better understanding:

Illustration:

*Consider **arr[] = {3, 0, 2, 0, 4}***

*Therefore, **left[] = {3, 3, 3, 3, 4}** and **right[] = {4, 4, 4, 4, 4}***
*Now consider iterating using **i** from 0 to end*

For i = 0:
=> left[0] = 3, right[0] = 4 and arr[0] = 3
*=> Water stored = min(left[0], right[0]) – arr[0] = min(3, 4) – 3 = 3 – 3 = **0***
*=> Total = 0 + 0 = **0***

For i = 1:
=> left[1] = 3, right[1] = 4 and arr[1] = 0
*=> Water stored = min(left[1], right[1]) – arr[1] = min(3, 4) – 0 = 3 – 0 = **3***
*=> Total = 0 + 3 = **3***

For i = 2:
=> left[2] = 3, right[2] = 4 and arr[2] = 2
*=> Water stored = min(left[2], right[2]) – arr[2] = min(3, 4) – 2 = 3 – 2 = **1***
*=> Total = 3 + 1 = **4***

For i = 3:
=> left[3] = 3, right[3] = 4 and arr[3] = 0
*=> Water stored = min(left[3], right[3]) – arr[3] = min(3, 4) – 0 = 3 – 0 = **3***
*=> Total = 4 + 3 = **7***

For i = 4:
=> left[4] = 4, right[4] = 4 and arr[4] = 4
*=> Water stored = min(left[4], right[4]) – arr[4] = min(4, 4) – 4 = 4 – 4 = **0***
*=> Total = 7 + 0 = **7***

*So total rain water trapped = **7***

Follow the steps mentioned below to implement the approach:

- Create two arrays **left[]** and **right[]** of size **N**. Create a variable (say **max**) to store the maximum found till a certain index during traversal.
- Run one loop from start to end:
  - In each iteration update max and also assign **left[i] = max**.
- Run another loop from end to start:
  - In each iteration update max found till now and also assign **right[i] = max**.
- Traverse the array from start to end.
  - The amount of water that will be stored in this column is **min(left[i], right[i]) – array[i]**
  - Add this value to the total amount of water stored
- Print the total amount of water stored.

Below is the implementation of the above approach.

C++ C Java Python C# JavaScript PHP

```python
# Python program to find maximum amount of water that can
# be trapped within given set of bars.

def findWater(arr, n):

    # left[i] contains height of tallest bar to the
    # left of i'th bar including itself
    left = [0]*n

    # Right [i] contains height of tallest bar to
    # the right of ith bar including itself
    right = [0]*n

    # Initialize result
    water = 0

    # Fill left array
    left[0] = arr[0]
    for i in range(1, n):
        left[i] = max(left[i-1], arr[i])

    # Fill right array
    right[n-1] = arr[n-1]
    for i in range(n-2, -1, -1):
        right[i] = max(right[i + 1], arr[i])

    # Calculate the accumulated water element by element
```

```
        # consider the amount of water on i'th bar, the
        # amount of water accumulated on this particular
        # bar will be equal to min(left[i], right[i]) - arr[i] .
        for i in range(0, n):
            water += min(left[i], right[i]) - arr[i]

        return water


    # Driver program
    if __name__ == '__main__':
        arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
        n = len(arr)
        print(findWater(arr, n))

    # This code is contributed by
    # Smitha Dinesh Semwal
```

Output

```
6
```

Complexity Analysis:

- **Time Complexity:** O(N). Only one traversal of the array is needed, So time Complexity is O(N).
- **Auxiliary Space :** O(N). Two extra arrays are needed, each of size N.

Approach 3 – Efficient Solution using Two Pointer Approach – O(N) Time and O(1) Space:

The idea for this approach is as follows:

At every index, The amount of rainwater stored is the difference between the current index height and a minimum of left max height and right max-height.

Here we can use the two-pointer approach to find the minimum among the left-max and right-max of the probable outermost boundary for any index and iterate likewise.

For example:

- *Say we have indices **i, j** and a boundary of **(left, right)**. where **i** is the left pointer and **j** is the right pointer.*
- *If the minimum is **arr[left]**, we can say that **i** is bounded in one side by **left** and no matter whatever the values are in between **(i, right)**, the rightmost boundary of **i** will at least have height **arr[right]** which is the probable outermost boundary for **i**.*
- *So the water height of water column at index **i** is **arr[left] – arr[i]** and we can increment **i** then.*
- *Similar things happen for **j** also.*

Follow the below illustration for a better understanding:

Illustration:

*Consider **ar[] = {3, 0, 2, 0, 4}***
left = 0, right = 4, l_max = 0, r_max = 0.

left = 0, right = 4:
=> *l_max = r_max.*
=> *water added = **0***
=> *r_max = 4, l_max = 0*
=> *right = 3, left = 0*

left = 0, right = 3:
=> *l_max < r_max.*
=> *water added = **0***
=> *r_max = 4, l_max = 3*
=> *right = 3, left = 1*

left = 1, right = 3:
=> *l_max < r_max.*
=> *water added = 3 – 0 = **3***
=> *r_max = 4, l_max = 3*
=> *right = 3, left = 2*
=> *Water trapped = **0 + 3 = 3***

left = 2, right = 3:
=> *l_max < r_max.*
=> *water added = 3 – 2 = **1***
=> *r_max = 4, l_max = 3*
=> *right = 3, left = 2*
=> *Water trapped = **3 + 1 = 4***

left = 3, right = 3:
=> *l_max < r_max.*
=> *water added = 3 – 0 = **3***
=> *r_max = 4, l_max = 3*

=> right = 3, left = 4

*=> Water trapped = **4 + 3 = 7***

So total water trapped = 7

Follow the steps mentioned below to implement the idea:

- Take two pointers **l** and **r**. Initialize **l** to the starting index **0** and **r** to the last index **N-1**.
- Since **l** is the first element, **left_max** would be **0**, and **right_max** for **r** would be **0**.
- While **l ? r**, iterate the array. We have two possible conditions
- **Condition1** : **left_max <= right max**
    - Consider Element at index **l**
    - Since we have traversed all elements to the left of l, **left_max is known**
    - For the right max of **l**, We can say that the **right max would always be >= current r_max here**
    - So, **min(left_max,right_max)** would always equal to **left_max** in this case
    - Increment **l**.
- **Condition2 : left_max > right max**
    - Consider Element at index **r**
    - Since we have traversed all elements to the right of r, **right_max is known**
    - For the left max of l, We can say that the **left max would always be >= current l_max here**
    - So, **min(left_max,right_max)** would always equal to **right_max** in this case
    - Decrement **r**.

Below is the implementation of the above approach.

C++ C Java Python C# JavaScript

```python
# Python3 implementation of the approach

# Function to return the maximum
# water that can be stored

def maxWater(arr, n):

    # Indices to traverse the array
    left = 0
    right = n-1

    # To store Left max and right max
    # for two pointers left and right
    l_max = 0
    r_max = 0

    # To store the total amount
    # of rain water trapped
    result = 0
    while (left <= right):

        # We need check for minimum of left
        # and right max for each element
        if r_max <= l_max:

            # Add the difference between
            #current value and right max at index r
            result += max(0, r_max-arr[right])

            # Update right max
            r_max = max(r_max, arr[right])

            # Update right pointer
            right -= 1
        else:

            # Add the difference between
            # current value and left max at index l
            result += max(0, l_max-arr[left])

            # Update left max
            l_max = max(l_max, arr[left])

            # Update left pointer
            left += 1
    return result


# Driver code
if __name__ == '__main__':
    arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
    N = len(arr)
    print(maxWater(arr, N))

# This code is contributed by Nikhil Chatragadda
```

Output

```
6
```

Time Complexity: O(N)
Auxiliary Space: O(1)

Other Possible Approaches

Some other approaches based on above solutions to solve Trapping Rainwater problems are:

Approach – Using Stack – O(N) Time and O(N) Space:

The idea to solve the problem using **stack** is as follows:

We can use a stack to track the bars that are bounded by the higher left and right bars. This can be done using only one iteration.

- *For this we will keep pushing elements in stack, until an element with higher value than the stack top is found. This denotes that there is a chance of storing position on the left side of the current element with the current bar being an end.*
- *So remove the smaller element on left and find the left bar (which is the current top of stack) and the amount of water stored by the left end bar and the current bar being the right end. Continue this till the stack is not empty or a higher value element is found.*

Follow the below illustration for a better understanding.

Illustration:

*Consider the array **arr[] = {3, 0, 2, 0, 4}** and an empty stack **st**.*

For i = 0:

=> The stack is empty. So no elements with higher value on left.

*=> Push the index into the stack. **st = {0}** [to keep track of the distance in between]*

For i = 1:

=> arr[1] is less than arr[stack top]. So arr[1] has a higher left bound.

*=> Push the index into stack. **st = {0, 1}***

For i = 2:

=> arr[2] is greater than arr[stack top]. So arr[2] is the higher right bound of current stack top.

=> Calculate the water stored in between the left and right bound of the stack top.

=> The stack top is the base height in between the left and right bound.

*=> Pop the stack top. So **st = {0}**.*

*=> Water stored in between when arr[0] and arr[2] are the bound= {min(arr[0], arr[2]) – arr[1]} * (2 – 0 – 1) = **2**.*

*=> arr[0] is greater than arr[2] Push the index into stack. **st = {0, 2}.***

*=> Total water stored = **0 + 2 = 2**.*

For i = 3:

=> arr[3] is less than arr[2]. So arr[3] has a higher left bound.

*=> Push the index into the stack. **st = {0, 2, 3}**.*

For i = 4:

=> arr[4] is greater than arr[stack top]. So arr[4] is the higher right bound of current stack top.

=> Calculate the water stored in same way as for i = 2. The base height is arr[3].

*=> Pop the stack top. So **st = {0, 2}**.*

*=> Water stored in between when arr[4] and arr[2] are the bound= {min(arr[4], arr[2]) – arr[3]} * (4 – 2 – 1) = **2**.*

=> arr[4] is greater than arr[2].

*=> Pop the stack. **st = {0}**.*

*=> Water stored in between arr[0] and arr[4] when arr[2] is the base height = {min(3, 4) – 2} * (4 – 0 – 1) = **3***

*=> arr[0] less than arr[4]. So pop stack. **st = {}**.*

*=> As no element left in the stack push the index. **st = {4}**.*

*=> Total water stored = **2 + 2 + 3 = 7**.*

*So the total amount of water stored = **7**.*

Follow the steps mentioned below to implement the idea:

- Loop through the indices of the bar array.
- For each bar, do the following:
    - Loop while the Stack is not empty and the current bar has a height greater than the top bar of the stack,
        - Store the index of the top bar in **pop_height** and pop it from the Stack.
        - Find the distance between the left bar(current top) of the popped bar and the current bar.
        - Find the minimum height between the **top bar** and the **current bar**.
        - The maximum water that can be trapped is **distance * min_height**.
        - The water trapped, including the popped bar, is **(distance * min_height) – height[pop_height]**.
        - Add that to the answer.
- Return the amount received as the total amount of water

Below is the implementation of the above approach.

C++ Java Python C# JavaScript

```
# Python implementation of the approach

# Function to return the maximum
```

```python
# water that can be stored

def maxWater(height):

    # Stores the indices of the bars
    stack = []

    # size of the array
    n = len(height)

    # Stores the final result
    ans = 0

    # Loop through the each bar
    for i in range(n):

        # Remove bars from the stack
        # until the condition holds
        while(len(stack) != 0 and (height[stack[-1]] < height[i])):

            # store the height of the top
            # and pop it.
            pop_height = height[stack[-1]]
            stack.pop()

            # If the stack does not have any
            # bars or the popped bar
            # has no left boundary
            if(len(stack) == 0):
                break

            # Get the distance between the
            # left and right boundary of
            # popped bar
            distance = i - stack[-1] - 1

            # Calculate the min. height
            min_height = min(height[stack[-1]], height[i])-pop_height

            ans += distance * min_height

        # If the stack is either empty or
        # height of the current bar is less than
        # or equal to the top bar of stack
        stack.append(i)

    return ans


# Driver code
if __name__ == '__main__':
    arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
    print(maxWater(arr))

# This code is contributed by rag2127
```

Output

```
6
```

Time Complexity: O(N)
Auxiliary Space: O(N)

Approach – Efficient solution Horizontal scan method – O(max (max_height, N) Time and O(1) Space:

The idea is as follows:

- *If max_height is the height of the tallest block, then it will be the maximum possible height for any trapped rainwater.*
- *And if for each unit of height we find the leftmost and the rightmost boundary for that height, we can consider all the blocks in between contain that amount of water.*
- *But this will consider the height of the bars also. So we have to subtract that from the total water trapped.*

This can be justified as follows. Say the sections for a certain height is $\{i_1, i_2\}, \{i_2, i_3\}, \ldots, \{i_{k-1}, i_k\}$
So the water stored in between for a single unit of height is the difference in between the indices $= (i_2 - i_1 - 1) + (i_3 - i_2 - 1) + \ldots + (i_k - i_{k-1} - 1\}) = \mathbf{i_k - i_1 - (k-1)}$ *where* $\mathbf{k}$ *is the number of bars in between.*

But as we are considering all the blocks in between left and right boundary, it considers all the bars also.
So the trapped water for a single unit becomes $\mathbf{(i_k - i_1 + 1)}$

Follow the below illustration for a better understanding.

Illustration:

Consider array ***arr[] = {3, 0, 2, 0, 4}****.*
max_height = 4 *and sum of all blocks = 2 + 3 + 4 =* ***9****.*

For height = 1:

=> leftmost boundary = 0 and rightmost boundary = 4.
=> All blocks in between contain 1 height of water.

*=> So amount of water trapped = (4 – 0 + 1) = **5***

*=> Total trapped water = 0+5 = **5***

For height = 2:

=> leftmost boundary = 0 and rightmost boundary = 4.

=> All blocks in between contain 2 height of water.

=> Earlier we have considered water columns with height 1. So there is increase of 1 unit in height.

*=> So amount of water trapped = (4 – 0 + 1) = **5***

*=> Total trapped water = 5 + 5 = **10***

For height = 3:

=> leftmost boundary = 0 and rightmost boundary = 4.

=> All blocks in between contain 3 height of water.

=> Earlier we have considered water columns with height 2. So there is increase of 1 unit in height.

*=> So amount of water trapped = (4 – 0 + 1) = **5***

*=> Total trapped water = 10 + 5 = **15***

For height = 4:

=> leftmost boundary = 4 and rightmost boundary = 4.

=> All blocks in between contain 4 height of water.

=> Earlier we have considered water columns with height 3. So there is increase of 1 unit in height.

*=> So amount of water trapped = (4 – 4 + 1) = **1***

*=> Total trapped water = 15 + 1 = **16***

*So total water trapped = 16 – total space taken by bars = **16 – 9 = 7**.*

Follow the steps mentioned below to implement the idea:

- Find the total number of blocks, i.e., the sum of the heights array, **num_blocks**
- Find the maximum height, **max_height**
- Store total water in a variable, **total** = 0
- Keep two pointers, **left = 0** and **right = N -1**, to store the leftmost and the rightmost boundaries for each unit of height
- For each height, **i** from **1 to max_height**, do the following
    - Find the leftmost and the rightmost boundary for the current height.
    - As mentioned earlier we can consider all the blocks in between these to have at least **i** unit of water.
    - Add this amount of water to the total trapped water.
- After the iteration is over, subtract **num_blocks** from **total** as we have considered them as water height during calculation.

Below is the implementation of the above approach.

C++ Java Python C# JavaScript

```python
# Python code to implement the approach

def trappedWater(arr):
    num_blocks = 0
    n = 0
    max_height = float('-inf')

    # Find total blocks, max height and length of array
    for height in arr:
        num_blocks += height
        n += 1
        max_height = max(max_height, height)

    # Total water, left pointer and right pointer
    # initialized to 0 and n - 1
    total = 0
    left = 0
    right = n - 1

    for i in range(1, max_height+1):

        # Find leftmost point greater than current row (i)
        while arr[left] < i:
            left += 1

        # Find rightmost point greater than current row (i)
        while arr[right] < i:
            right -= 1

        # Water in this row = right - left + 1
        total += (right - left + 1)

    total -= num_blocks
    return total

# Driver code
if __name__ == "__main__":
    arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
    print(trappedWater(arr))
```

Output

6

Time Complexity: O(max (max_height, N))
Auxiliary Space : O(1)

Approach – Optimal Solution similar to linear search – O(N) Time and O(1) Space:

Here another efficient solution has been shown.

> *The concept is that if there is a larger bar to the right, then the water can be retained with a height equal to the smaller bar on the left.*
> *If there are no larger bars to the right, then start from the right. There must be a larger bar to the left now.*

Follow the steps mentioned below to implement the idea:

- Loop from index 0 to the end of the given array.
  - If a bar greater than or equal to the previous bar is encountered, then store the index of that bar (say prev_index).
  - Keep adding the previous bar's height minus the current (i^{th}) bar to the final answer.
  - Have a temporary variable to store the amount of water in the current segment.
  - If no bar greater than or equal to the previous bar is found, then quit.
- If prev_index < size of the input array, then subtract the temporary variable from the answer, because we have considered the last segment that has no higher right bound.
  - Loop from the end of the input array to **prev_index**.
  - Find a bar greater than or equal to the previous bar (in this case, the last bar from backward).

Below is the implementation of the above approach.

C++ C Java Python C# JavaScript

```python
# Python3 implementation of the approach

# Function to return the maximum
# water that can be stored

def maxWater(arr, n):
    size = n - 1

    # Let the first element be stored as
    # previous, we shall loop from index 1
    prev = arr[0]

    # To store previous wall's index
    prev_index = 0
    water = 0

    # To store the water until a larger wall
    # is found, if there are no larger walls
    # then delete temp value from water
    temp = 0
    for i in range(1, size + 1):

        # If the current wall is taller than
        # the previous wall then make current
        # wall as the previous wall and its
        # index as previous wall's index
        # for the subsequent loops
        if (arr[i] >= prev):
            prev = arr[i]
            prev_index = i

            # Because larger or same height wall is found
            temp = 0
        else:

            # Since current wall is shorter than
            # the previous, we subtract previous
            # wall's height from the current wall's
            # height and add it to the water
            water += prev - arr[i]

            # Store the same value in temp as well
            # If we dont find any larger wall then
            # we will subtract temp from water
            temp += prev - arr[i]

    # If the last wall was larger than or equal
    # to the previous wall then prev_index would
    # be equal to size of the array (last element)
    # If we didn't find a wall greater than or equal
    # to the previous wall from the left then
    # prev_index must be less than the index
    # of the last element
    if (prev_index < size):

        # Temp would've stored the water collected
        # from previous largest wall till the end
        # of array if no larger wall was found then
        # it has excess water and remove that
        # from 'water' var
        water -= temp

        # We start from the end of the array, so previous
        # should be assigned to the last element
        prev = arr[size]

        # Loop from the end of array up to the 'previous index'
        # which would contain the "largest wall from the left"
```

```
            for i in range(size, prev_index - 1, -1):

                # Right end wall will be definitely smaller
                # than the 'previous index' wall
                if (arr[i] >= prev):
                    prev = arr[i]
                else:
                    water += prev - arr[i]

        # Return the maximum water
        return water


    # Driver code
    if __name__ == '__main__':
        arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
        N = len(arr)
        print(maxWater(arr, N))

    # This code is contributed by Mohit Kumar
```

Output

```
6
```

Time Complexity: O(N)
Auxiliary Space: O(1).