Insights into Algorithm

Ge Shi, Oct 4, 2022

1. Resources

LeetCode Interview Top-150
Leetcode面试高频题分类刷题总结
Coderust: Hacking the Coding Interview

2. LinkedList

2.1. Reverse a LinkedList

Consider to use recursive method if apply a sequence of same actions to subsequences of linked lists
Recursive:

```
def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next:
            return head
        newhead = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return newhead
```

Iterative:

```
def reverseLinkedList(self, head, k):

    # Reverse k nodes of the given linked list.
    # This function assumes that the list contains
    # atleast k nodes.
    new_head, ptr = None, head
    while k:

        # Keep track of the next node to process in the
        # original list
        next_node = ptr.next

        # Insert the node pointed to by "ptr"
        # at the beginning of the reversed list
        ptr.next = new_head
        new_head = ptr

        # Move on to the next node
        ptr = next_node

        # Decrement the count of nodes to be reversed by 1
        k -= 1

    # Return the head of the reversed list
    return new_head
```

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

Examples:
LC 92. Reverse Linked List II
LC 25. Reverse Nodes in k-Group

2.2. Detect Cycle and Find Start Point of LinkedList

Use fast and slow pointers to check if there is a cycle and where the cycle starts.

```
def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        fast = head
        slow = head
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next
            if slow == fast:
                break
        if not fast or not fast.next:
            return None
        slow = head
        while slow!=fast:
            fast = fast.next
            slow = slow.next
        return slow
```

Examples:
LC 141. Linked List Cycle

2.3. Delete Node with Target Value

Note:
1) Use a dummy head pointer to be the father of head
2) Let the current pointer pointing to the dummy head
3) Always decide on the next pointer of the current pointer
4) Return the child of dummy head (especially when you do deletion, consider if head is the one node that should be deleted.)

```
class Solution:
    def removeElements(self, head: Optional[ListNode], val: int) -> 
Optional[ListNode]:
        pre = ListNode(val=-1, next=head)
        cur = pre
        while cur.next:
            if cur.next.val == val:
                cur.next = cur.next.next
            else:
                cur = cur.next
        return pre.next
```

2.4. Double LinkedList

A doubly linked list (DLL) is a type of linked list where each node contains a data element and two pointers (or references) to the next and previous nodes in the sequence. This structure allows traversal in both directions (forward and backward), making certain operations more flexible compared to a singly linked list.

Using dummy nodes for the head and tail of a doubly linked list can simplify boundary condition handling, such as inserting or removing nodes when the list is empty or has only one element.

Code Template:

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

class DoublyLinkedList:
    def __init__(self):
        # Dummy nodes
        self.head = Node()  # Dummy head
        self.tail = Node()  # Dummy tail
        # Initialize the list to point dummy head to dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def add_node_to_end(self, node):
        prev_tail = self.tail.prev
        node.next = self.tail
        self.tail.prev = node
        prev_tail.next = node
        node.prev = prev_tail

    def add_node_to_front(self, node):
        next_head = self.head.next
        node.next = next_head
        self.head.next = node
        next_head.prev = node
        node.prev = self.head

    def add_to_front(self, data):
        new_node = Node(data)
        new_node.next = self.head.next
        new_node.prev = self.head
        self.head.next.prev = new_node
        self.head.next = new_node

    def add_to_end(self, data):
        new_node = Node(data)
        new_node.prev = self.tail.prev
        new_node.next = self.tail
        self.tail.prev.next = new_node
        self.tail.prev = new_node

    def remove_node(self, node):
        node.next.prev = node.prev
        node.prev.next = node.next

    def remove_data(self, data):
        current = self.head.next
        while current != self.tail:
            if current.data == data:
                current.prev.next = current.next
                current.next.prev = current.prev
                return
            current = current.next
        print("Node not found.")

    def update_data(self, target_data, new_data):
        current = self.head.next
        while current != self.tail:
            if current.data == target_data:
                current.data = new_data
                return
            current = current.next
        print("Node not found.")
```

Examples:
LC 146. LRU Cache

2.5. Tree-based LinkedList

3. Sort/Search

3.1. Customize Compare Key

Lambda function:

```
sorted_arr = sorted(arr, key = lambda num: abs(num))
```

Uni-value:

```
# Define the custom key function
def custom_key(x):
    return abs(x)

# Sort using the custom key function
sorted_arr = sorted(arr, key=custom_key)
```

Bi-value:

```
from functools import cmp_to_key

# Define the custom comparator function
def custom_comparator(x, y):
    if x < y:
        return -1
    elif x > y:
        return 1
    else:
        return 0

# List to be sorted
arr = [3, 1, 4, 1, 5, 9, 2, 6]

# Convert the comparator to a key function
key_func = cmp_to_key(custom_comparator)

# Sort using the custom comparator
sorted_arr = sorted(arr, key=key_func)

# Class Implementation
from functools import cmp_to_key

class AbsoluteValueComparator:
    def __call__(self, x, y):
        if abs(x) < abs(y):
            return -1
        elif abs(x) > abs(y):
            return 1
        else:
            return 0

# List to be sorted
arr = [-3, 1, -2, 4, 0, -1]

# Create an instance of the comparator class
comparator = AbsoluteValueComparator()

# Convert the comparator to a key function
key_func = cmp_to_key(comparator)

# Sort using the comparator class
sorted_arr = sorted(arr, key=key_func)
```

To define a class with a custom comparator in Python, you can implement special methods that define how instances of the class should be compared. The most common methods are __lt__ (less than), __le__ (less than or equal to), __eq__ (equal to), __ne__ (not equal to), __gt__ (greater than), and __ge__ (greater than or equal to).

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __lt__(self, other):
        return self.age < other.age

    def __le__(self, other):
        return self.age <= other.age

    def __eq__(self, other):
        return self.age == other.age

    def __ne__(self, other):
        return self.age != other.age

    def __gt__(self, other):
        return self.age > other.age

    def __ge__(self, other):
        return self.age >= other.age
    def __repr__(self):
        return f"{self.name} ({self.age})"

# Example usage
people = [Person("Alice", 30), Person("Bob", 25), Person("Charlie", 35)]
sorted_people = sorted(people)
```

Argsort based on given iterable items:

```
sentences = ["apple", "banana", "cherry"]
times = [5, 3, 5]

indices = list(range(len(sentences)))
sorted_indices = sorted(indices, key=lambda x: (-times[x], setences[x]))
# Sort the sentences according to the specified rules and return the
desired sorted indices.
```

3.2. Dutch National Flag Sorting

The most common solution is known as the three-way partitioning algorithm or the three-way quicksort. It uses three pointers to partition the array into three sections:

```
def sortColors(self, nums: List[int]) -> None:
    """
    Dutch National Flag problem solution.
    """
    # For all idx < p0 : nums[idx < p0] = 0
    # curr is an index of elements under consideration
    p0 = curr = 0

    # For all idx > p2 : nums[idx > p2] = 2
    p2 = len(nums) - 1

    while curr <= p2:
        if nums[curr] == 0:
            nums[p0], nums[curr] = nums[curr], nums[p0]
            p0 += 1
            curr += 1
        elif nums[curr] == 2:
            nums[curr], nums[p2] = nums[p2], nums[curr]
            p2 -= 1
        else:
            curr += 1
```

75. Sort Colors

3.3. Kth Partition

Creates a copy of the array and partially sorts it in such a way that the value of the element in k-th position is in the position it would be in a sorted array. In the output array, all elements smaller than the k-th element are located to the left of this element and all equal or greater are located to its right. The ordering of the elements in the two partitions on the either side of the k-th element in the output array is undefined.

```
def partition(arr, left, right):
    pivot = arr[right] # or pivot = random.choice(arr)
    i = left - 1

    for j in range(left, right):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[right] = arr[right], arr[i + 1]
    return i + 1

def partition_around_k(arr, k):
    left, right = 0, len(arr) - 1

    while left < right:
        pivot_index = partition(arr, left, right)

        if pivot_index == k:
            return arr
        elif pivot_index < k:
            left = pivot_index + 1
        else:
            right = pivot_index - 1

# Test the partition_around_k implementation
```

```
arr = [3, 2, 1, 5, 4]
k = 2
partitioned_array = partition_around_k(arr, k-1)
```

Partition Function: This function rearranges the elements in the array such that elements less than the pivot are on the left, and elements greater than or equal to the pivot are on the right. It returns the index of the pivot element.

Partition Around k Function: This function repeatedly partitions the array until the pivot element is at the k-th position. It adjusts the search range (left and right) based on the position of the pivot.

It can be used for Quickselect, also known as Hoare's selection algorithm, is an algorithm for finding the k_th smallest (or largest) element in an unordered list. It is significant because it has an average runtime of $O(n)$.

Steps:

- Partition the Array: Choose a pivot element and partition the array into two subarrays: elements less than the pivot and elements greater than or equal to the pivot.
- Determine the Position: Determine the position of the pivot element in the sorted array.
- Recurse: If the pivot's position matches k, return the pivot element. Otherwise, recurse into the appropriate partition that contains the k-th smallest element.

Time Complexity: Best O(n), Average O(n), Worst Case O(n^2)
Space Complexity: In-place O(1), recursive call stack O(logn), O(n) in the worst case

215. Kth Largest Element in an Array

3.4. Quick Sort

Quick Sort is a highly efficient sorting algorithm and is based on the divide-and-conquer approach. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Steps:

- Choose a Pivot: Select an element from the array as the pivot. Common strategies include choosing the first element, the last element, a random element, or the median.
- Partition the Array: Rearrange the elements so that all elements less than the pivot are on its left, and all elements greater than or equal to the pivot are on its right.
- Recursively Apply: Apply the above steps to the sub-arrays of elements with smaller values and separately to the sub-array of elements with greater values.

```python
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Test the quick sort implementation
arr = [10, 7, 8, 9, 1, 5]
quick_sort(arr, 0, len(arr) - 1)
```

The choice of pivot and partitioning strategy significantly affects its performance, especially in the worst case.

Time Complexity: best O(nlogn), average O(nlogn), worst O(n^2)
Space Complexity: O(logn)
Stability: The algorithm is not stable.

3.5. Bucket/Counting Sorting

Counting sort is a non-comparison sorting algorithm. It can be used to sort when the range of the numbers are bounded.
X_shift = X_original - lowestValue

Counting Sort: Efficient when the range of input values (k) is not significantly larger than the number of elements (n). Uses extra space proportional to the range of the input values.

Steps:

- Find the range of the input values.
- Create a count array to store the count of each unique value.
- Modify the count array by adding the previous counts (cumulative sum).
- Build the output array by placing the elements in their correct positions.

```
def counting_sort(arr):
    max_val = max(arr)
    min_val = min(arr)
    range_of_elements = max_val - min_val + 1

    count = [0] * range_of_elements
    output = [0] * len(arr)

    for num in arr:
        count[num - min_val] += 1

    for i in range(1, len(count)):
        count[i] += count[i - 1]

    for num in reversed(arr):
        output[count[num - min_val] - 1] = num
        count[num - min_val] -= 1

    for i in range(len(arr)):
        arr[i] = output[i]
```

Bucket Sort: Efficient for uniformly distributed data. The time complexity depends on the distribution of elements across the buckets. Uses extra space proportional to the number of buckets and elements.

Steps:

- Create an array of empty buckets.
- Distribute the elements into buckets based on a hash function.
- Sort each bucket individually.
- Concatenate the sorted buckets.

```
def bucket_sort(arr, bucket_size=5):
    if len(arr) == 0:
        return arr

    min_val, max_val = min(arr), max(arr)
    bucket_count = (max_val - min_val) // bucket_size + 1
    buckets = [[] for _ in range(bucket_count)]

    for num in arr:
        buckets[(num - min_val) // bucket_size].append(num)

    sorted_array = []
    for bucket in buckets:
        sorted_array.extend(sorted(bucket))

    for i in range(len(arr)):
        arr[i] = sorted_array[i]
```

Algorithm	**Time Complexity (Best)**	**Time Complexity (Average)**	**Time Complexity (Worst)**	**Space Complexity**
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+k)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n+k)

Counting Sort: Efficient when the range of input values (k) is not significantly larger than the number of elements (n). Uses extra space proportional to the range of the input values.

Bucket Sort: Efficient for uniformly distributed data. The time complexity depends on the distribution of elements across the buckets. Uses extra space proportional to the number of buckets and elements.

3.6. Pigeonhole Sort

Pigeonhole sort is a non-comparison-based sorting algorithm that is efficient when the range of key values (difference between the maximum and minimum values) is not significantly larger than the number of elements to be sorted. The algorithm is named after the pigeonhole principle, which states that if n items are put into m containers, with n>m, then at least one container must contain more than one item.

Steps:

- Find the minimum and maximum values in the array.
- Calculate the range of the values.
- Create a list of empty pigeonholes (each representing a potential key value).
- Place each element into its corresponding pigeonhole.
- Concatenate the elements from the pigeonholes back into the original array.

```
def pigeonhole_sort(arr):
    # Find the minimum and maximum values in the array
    min_val = min(arr)
    max_val = max(arr)

    # Calculate the range of the values
    size = max_val - min_val + 1

    # Create empty pigeonholes
    holes = [[] for _ in range(size)]

    # Place each element in its corresponding pigeonhole
    for num in arr:
        holes[num - min_val].append(num)

    # Concatenate the elements from the pigeonholes back into the original
array
    index = 0
    for hole in holes:
        for num in hole:
            arr[index] = num
            index += 1

# Test the pigeonhole sort implementation
arr = [8, 3, 2, 7, 4, 6, 8]
pigeonhole_sort(arr)
```

Time Complexity: O(n+k), where n is the number of elements and k is the range of the input.
Space Complexity: O(n+k) for the pigeonholes.
Stability: The algorithm is stable.

3.7. Merge Sort

Time Complexity: worst O(nlogn)
Space Complexity: O(n+logn)
It uses divide and conquer strategy, firstly you find the middle position of the iterable items and divide it, then merge the two parts.

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

        # Sort first and second halves
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)

        merge(arr, left, mid, right)

def merge(arr, left, mid, right):
    # Create temporary arrays to hold the two halves to merge
    left_subarray = arr[left:mid + 1]
    right_subarray = arr[mid + 1:right + 1]

    # Initialize pointers for left_subarray, right_subarray and the merged
array
    left_index, right_index = 0, 0
    merged_index = left

    # Merge the temporary arrays back into the original array
    while left_index < len(left_subarray) and right_index <
len(right_subarray):
        if left_subarray[left_index] <= right_subarray[right_index]:
            arr[merged_index] = left_subarray[left_index]
            left_index += 1
        else:
            arr[merged_index] = right_subarray[right_index]
            right_index += 1
        merged_index += 1

    # Copy the remaining elements of left_subarray, if any
    while left_index < len(left_subarray):
        arr[merged_index] = left_subarray[left_index]
        left_index += 1
        merged_index += 1

    # Copy the remaining elements of right_subarray, if any
    while right_index < len(right_subarray):
        arr[merged_index] = right_subarray[right_index]
        right_index += 1
        merged_index += 1
```

[LC 21. Merge Two Sorted Lists]
[LC 148. Sort List]

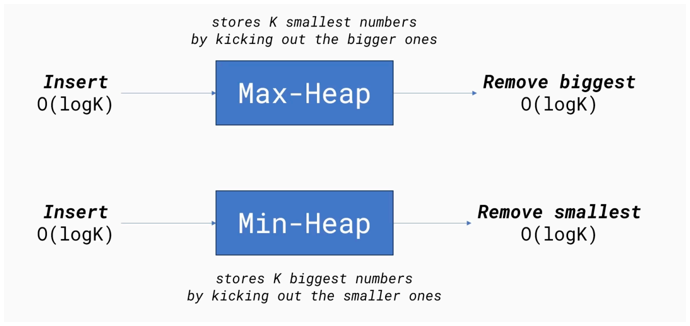
3.8. Partial Sort to Find k-th Element

Heap is good for finding the k-th largest or smallest element without sorting the entire array. The idea is to keep a heap of size k, popping out all the elements that do not meet the requirement.

Time Complexity: O(nlogk)
Space Complexity: O(k)

Keep in mind that to find the k-th largest element, you use a min heap, to find the k-th smallest element, you use a max heap.

```
def findKthLargest(self, nums, k):
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap[0]
```



3.9. Binary Search

Steps:

- Pre-processing - Sort if collection is unsorted.
- Binary Search - Using a loop or recursion to divide search space in half after each comparison.
- Post-processing - Determine viable candidates in the remaining space.

易错点
Return: 找到target的index，equal or less than target，equal or greater than target

左闭右闭, 左闭右开

Binary search in array int[] nums

左闭右开

```
# input: int[] nums, int target
int left = 0;
int right = nums.length;
while(left<right) {
    mid = left + (high-low)/2;
    if (nums[mid]==target) break;
    if (nums[mid]<target)
        low = mid + 1;
    else
        right = mid;
}

# alternative 左闭右闭
int low = 0;
int high = nums.length-1;
while(left<=right) {
    mid = left + (high-low)/2;
    if (nums[mid]==target) break;
    if (nums[mid]<target)
        low = mid + 1;
    else
        right = mid - 1;
}
# alternative
int low = 0;
int high = nums.length-1;
while(left<right) {
    mid = right - (high-low)/2;
    if (nums[mid]==target) break;
    if (nums[mid]<target)
        low = mid + 1;
    else
        right = mid;
}
Return left;
# if you want to find the index of the value that is equal or greater to
the target value, you can do
```

```
int low = 0;
int high = nums.length;
int ans = nums.length;
while(left<right) {
    mid = low + (high-low)/2;
    if (nums[mid]>=target) # change to < if you're looking for strict greater
        ans = mid
        right = mid;
    Else
        left = mid + 1;
}
return ans;
```

Binary search for the peak of V style non-monotone array:

```
int low = 0;
int high = nums.length()-1;
long ans = getCost(nums, target, nums[0]);
while(low<high) {
    int mid = low + (high-low)/2;
    long avg_cost1 = getCost(nums, target, mid);
    long avg_cost2 = getCost(nums, target, mid+1);
    ans = Math.min(avg_cost1, avg_cost2);
    if(avg_cost1>avg_cost2)
        low = mid+1;
    else
        high = mid;
}
return ans;
```

Binary Search Explore Card

Template: Check if a target value exists in the arr, if yes, return the index, otherwise -1

```
def binarySearch(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    # End Condition: left > right
    return -1
```

Template 2: Search for the first element that's greater than or equal to target

```
def binarySearch(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums) - 1 # right = len(nums) if may not exist
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < target: # if you want to search for the first
element strictly greater than target, use “nums[mid] <= target”
            left = mid + 1
        else:
            right = mid
    return left # If does not exist left = len(nums)
```

Template 3: Search for the last element that's smaller than or equal to target

```
def binarySearch(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums) - 1 # left = -1 if may not exist
    while left < right:
        mid = right - (right - left) // 2
        if nums[mid] > target: # if you want to search for the first element
strictly smaller than target, use “nums[mid] <= target”
            right = mid - 1
        else:
            left = mid
    return right # If does not exist right = -1
```

Left and right initualization is decided by if target could be out of the bounds.
Examples:
4. Median of Two Sorted Arrays

3.10. Set Cover

The set cover problem is finding the smallest range that covers at least one element from each of the given sublists. This can be solved using a sliding window (two-pointer) approach after sorting the individual elements from each sublist.

The key point the solve such questions is:

- Sort each sublist
- Sort the list based on the starting/ending point
- Use pointers to point to the head of each sublist
- Compare the ending of former range and the starting of latter range

- Type I, the sublist is a list of integers:

An example question can be: LC 632. Smallest Range Covering Elements from K Lists

```
def smallestRange(self, nums: List[List[int]]) -> List[int]:
    ans = [0, inf]
    cur_max = max([num_list[0] for num_list in nums])
    pointers = [(num_list[0], 0, i) for i, num_list in enumerate(nums)]
    pointers.sort(key = lambda x: x[0])
    while pointers:
        val, pos, loc = heappop(pointers)
        if cur_max - val < ans[1]-ans[0]:
            ans = [val, cur_max]
        if pos<len(nums[loc])-1:
            heappush(pointers, (nums[loc][pos+1], pos+1, loc))
            cur_max = max(nums[loc][pos+1], cur_max)
        else:
            break
    return ans
```

Examples:
LC 632. Smallest Range Covering Elements from K Lists

- Type II, the sublist is a pair of integers indicating a range:

An example question can be: LC 452. Minimum Number of Arrows to Burst Balloons

```
def findMinArrowShots(self, points: List[List[int]]) -> int:
    points.sort(key=lambda x:(x[1], x[0]))
    n = len(points)
    ends = points[0][1]
    ans = 0
    for i in range(n):
        if points[i][0] <= ends:
            continue
        ans += 1
        ends = points[i][1]
    ans += 1
    return ans
```

Examples:
LC 452. Minimum Number of Arrows to Burst Balloons

4. String

4.1. String Rotation

Consider a string S = "helloworld". Now, given another string T = "lloworldhe", can we figure out if T is a rotated version of S? By rotated version, we mean taking S and shifting it any number of spaces (with wrap around). For example, if S = "abc" and we shifted it to the left once, we would have "bca".

Yes, we check if T is a rotated version of S by checking if it is a **substring of S + S**. This is because S + S contains all of the rotations of S.

Let t = s + s. We can easily and efficiently check all possible rotations by **removing the first and last character of t**, then checking if s is a substring of t.

Example:
[LC 459. Repeated Substring Pattern]

4.2. Boyer-Moore (BM) Algorithm

4.3. Knuth–Morris–Pratt (KMP) Algorithm

[Knuth Morris Pratt (KMP) String Search Algorithm - tutorial with failure function in Java]

The KMP algorithm is used to match the pattern in the reference string.
There are mainly two steps:

1. Build T array from pattern, T array is used to store the maximum length of proper prefix and suffix **ending at** index i
2. Search pattern in reference string with the assistance of T array

Note:

1. T[0], have to be 0, pattern should be at lease length of 2, left and right pointer need to be initialized as 0 and 1
2. In searching, we use two pointers and while looping, remember to track if any pointer points to a position that is greater than either the pattern string or the reference string

```
def strStr(self, reference: str, pattern: str) -> int:
    def build_kmp(W: str):
        T = [0]*len(W)
        n = len(W)
        left, right = 0, 1
        while right < n:
            if W[right]==W[left]:
                left+=1
                T[right] = left
                right+=1
            elif left > 0:
                left = T[left-1]
            else:
                T[right]=0
                right+=1
        return T
    T = build_kmp(pattern)

    # This is the search part
    i, j = 0, 0
    while i<len(reference):
        while j<len(pattern) and i<len(reference) and reference[i] ==
pattern[j]:
            j+=1
            i+=1
        if j==len(pattern):
            return i-len(pattern)
        elif i==len(reference):
            return -1
        elif j>0:
            j=T[j-1]
        else:
            i+=1
    return -1
```

Examples:
LC 28. Find the Index of the First Occurrence in a String
LC 1392. Longest Happy Prefix

4.4. Rabin-Karp (RK) algorithm

Rolling Hash Function Tutorial, used by Rabin-Karp String Searching Algorithm
The Rabin-Karp (RK) algorithm is a string searching algorithm that uses hashing to find a pattern within a text efficiently. It is particularly useful for searching multiple patterns at once.

Hashing: The core idea of the Rabin-Karp algorithm is to use a hash function to convert a string (pattern and substrings of the text) into a numerical value. This allows for efficient comparison of the pattern with substrings of the text.
Rolling Hash: To efficiently compute the hash of the next substring in the text without recomputing from scratch, the Rabin-Karp algorithm uses a rolling hash function.

Hash Function:
A common choice is a **polynomial hash** function. The hash value for a string S of length m is computed as follows:

hash(S)=(S[0]⋅d^(m−1)+S[1]⋅d^(m−2)+…+S[m−1]⋅d^0)mod(q)

Rolling Hash:
The rolling hash allows efficient computation of the hash value for the next substring. If the current substring is T[i..i+m−1] and the next substring is T[i+1..i+m], the rolling hash is updated as:

hash(T[i+1..i+m])=(d · (hash(T[i..i+m−1])−T[i] · d^(m−1))+T[i+m])mod(q)

Time complexity: **Average Case**: O(n+m), **Worst Case**: O(nm)
Space complexity: O(1)

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

    # The value of h would be "pow(d, m-1) % q"
    for i in range(m-1):
        h = (h * d) % q

    # Calculate the hash value of the pattern and first window of text
    for i in range(m):
        p = (d * p + ord(pattern[i])) % q
        t = (d * t + ord(text[i])) % q

    # Slide the pattern over text one by one
    for i in range(n - m + 1):
        # Check the hash values of the current window of text and pattern
        if p == t:
            # Check for characters one by one
            if text[i:i+m] == pattern:
                print(f"Pattern found at index {i}")

        # Calculate hash value for the next window of text
        if i < n - m:
            t = (d * (t - ord(text[i]) * h) + ord(text[i + m])) % q

            # We might get a negative value of t, convert it to positive
            if t < 0:
                t += q

# Example usage
text = "GEEKS FOR GEEKS"
pattern = "GEEK"
rabin_karp(text, pattern)
```

Examples:
LC 28. Find the Index of the First Occurrence in a String

4.5. Prefix Count

Find the prefix of a string that exists in the same string array with nested loops

```
# input: String[] members
# map key is a member string, value is a list of prefix without replicas
HashMap<String, HashSet<String>> map = new HashMap<>();
for(int i=0;i<members.length;i++) {
    HashSet<String> prefixSet = new HashSet<>();
    for(int j=0;j<members.length;j++) {
        if(members[i].contains(members[j]) &&
!members[i].equals(members[j])) {
            prefixSet.add(members[j]);
        }
    }
    map.putIfAbsent(members[i], prefixSet);
}
```

Count of occurrences of strings in a string array (discount prefix) and sort them

```
# input: String[] members, String[] dialogues
# map key is a member string, value is a list of prefix without replicas

# Count string occurrences when the string id consists of numbers
HashMap<String, Integer> count = new HashMap<>();
for(String dialogue : dialogues) {
    for(String member : members) {
        if(dialogue.contains(member)) {
            # the next digit of the id of a prefix string is still a number
digit, but not for a entire id
            int index =
dialogue.charAt(dialogue.indexOf(member)+member.length()) - '0';
```

```
            # it means it’s just a prefix of another member
            if(index >= 0 && index <= 9) continue;
            count.put(member, count.getOrDefault(member, 0)+1);
        }
    }
}

# define a comparator of map entry class
class EntryCompare implements Comparator<Map.Entry<String, Integer>>
{
    public int compare(Map.Entry<String, Integer> m1, Map.Entry<String,
Integer> m2)
    {
        if (m1.getValue() == m2.getValue()) {
            return m1.getKey().compareTo(m2.getKey());
        }
        # sort integers from large to small
        return Integer.compare(m2.getValue(), m1.getValue())
    }
}

EntryCompare entryComparator = new EntryCompare();
List<Map.Entry<String, Integer>> entryList = new LinkedList<>();
for(Map.Entry<String, Integer> entrySet : count.entrySet()) {
    entryList.add(entrySet);
}
# sort the entry list from large to small with regard to count numbers and
from small to large with regard to lexical order
Collections.sort(entryList, entryComparator);

# Convert entry list to a string array with defined format
String[] ans = new String[entryList.size()];
for(int i=0;i<entryList.size();i++) {
    Map.Entry<String, Integer> entrySet = entryList.get(i);
    StringBuilder str = new StringBuilder();
    str.append(entrySet.getKey());
    str.append("=");
    str.append(entrySet.getValue());
    ans[i] = str.toString();
}
return ans;
```

```
# Alternative more efficient code for sorting
List<String> entryList = new ArrayList<>(count.keySet());
Collections.sort(entryList, (w1, w2) -> count.get(w1).equals(count.get(w2))
? w1.compareTo(w2) : count.get(w2) - count.get(w1));
return entryList;
```

5. Array/Matrix

Youtube: Kadane's Algorithm to Maximum Sum Subarray Problem
Maximum contiguous circular sum

5.1. Add up the Digits of an Integer

```
K = sum(int(b) for b in str(num))
```

5.2. Remove Elements of a value from Array

```
left = 0   # keep track of the position to put nonzero number to
for i in range(len(nums)):
    if nums[i]==target:
        continue
    nums[left] = nums[i]   # move the nonzero value to left
    left += 1
```

27. Remove Element

5.3. V Shape Array Sort (Two Pointers)

```
n = len(nums)
left = 0
right = n-1
ans = []
while left<=right:
    if abs(nums[left])>=abs(nums[right]):
        ans.append(nums[left]**2)
        left += 1
    else:
        ans.append(nums[right]**2)
        right -= 1
    return ans[::-1]
```

5.4. Maximum Subarray Problem

“Maximum Subarray Problem” for a regular array (not circular), can be solved by **Kadane’s algorithm** (an iterative dynamic programming algorithm).
The maximum subarray problem is the task of finding the largest possible sum of a contiguous subarray, within a given one-dimensional array A[1…n] of numbers.

Define two-variable:
    currSum which stores maximum sum ending here. Initialize currSum with 0
    maxSum which stores the maximum sum so far. Initialize maxSum with INT_MIN
Iterate over the array:
    add the value of the current element to currSum and check
    If currSum is less than zero, make currSum equal to zero.
    If currSum is greater than maxSum, update maxSum equals to currSum.
Return the value of maxSum.

Variant 1: allow empty subarray

```
int kadane(int a[], int n)
{
    int max_so_far = 0, max_ending_here = 0;
    int i;
    for (i = 0; i < n; i++)
    {
        if (max_ending_here + a[i] < 0)
            max_ending_here = 0;
        else:
            max_ending_here += a[i]
        max_so_far = max(max_so_far, max_ending_here);
    }
    return max_so_far;
}
```

Variant 2: Do not allow empty subarray

```
int kadane(int a[], int n)
{
    int max_so_far = a[0], max_ending_here = 0;
    int i;
    for (i = 0; i < n; i++)
    {
        if (max_ending_here < 0)
            max_ending_here = a[i];
        else:
            max_ending_here += a[i]
        max_so_far = max(max_so_far, max_ending_here);
    }
    return max_so_far;
}
```

Time complexity: O(N), Where N is the size of the array.
Space complexity: O(1)

For a **circular array**, the maximum subarray sum can be either the maximum "normal sum" which is the maximum sum of the ordinary array or a "special sum" which is the maximum sum of a prefix sum and a suffix sum of the ordinary array where the prefix and suffix don't overlap. We can calculate both the normal sum and the special sum and return the larger one.

Instead of thinking about the "special sum" as the sum of a prefix and a suffix, we can think about it as the sum of all elements, minus a subarray in the middle. In this case, we want to minimize this middle subarray's sum, which we can calculate using Kadane's algorithm as well.

“special sum” = total - minimum “normal contiguous subarray’s sum”

```
class Solution {
    public int maxSubarraySumCircular(int[] nums) {
        int curMax = 0, curMin = 0, sum = 0, maxSum = nums[0], minSum =
nums[0];
        for (int num : nums) {
            curMax = Math.max(curMax, 0) + num;
            maxSum = Math.max(maxSum, curMax);
            curMin = Math.min(curMin, 0) + num;
            minSum = Math.min(minSum, curMin);
            sum += num;
        }
        return sum == minSum ? maxSum : Math.max(maxSum, sum - minSum);
    }
}
```

5.5. 2D Matrix Rotation/Spiral

Spiral Order Traversal:

- Determine the number of sub-matrix, row and col sizes

- Loop through the current border
    - Traverse the top row from left to right.
    - Traverse the right column from top to bottom.
    - Traverse the bottom row from right to left.
    - Traverse the left column from bottom to top.
- Repeat the above steps for the inner sub-matrix.
- Edge case when the inner matrix degrade to a vector.

Tricks

- Determine one col or row that is static, they other dimension can be variant
- When switch row to col or vise versa, the variant dimension changes, the starting point for the variant dimension is the same as the last static dimension

```
def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
    n, m = len(matrix), len(matrix[0])
    ans = []
    row_size = m
    col_size = n
    i = 0
    while i < min(m,n)//2:
        for k in range(col_size-1):
            ans.append(matrix[i][i+k])
        for k in range(row_size-1):
            ans.append(matrix[i+k][n-1-i])
        for k in range(col_size-1):
            ans.append(matrix[m-1-i][n-1-i-k])
        for k in range(row_size-1):
            ans.append(matrix[m-1-i-k][i])
        row_size -= 2
        col_size -= 2
        i += 1

    if row_size==1 and col_size==1:
        ans.append(matrix[m//2][n//2])
    elif row_size>1 and col_size==1:
        for k in range(row_size):
            ans.append(matrix[i+k][n//2])
    elif col_size>1 and row_size==1:
        for k in range(col_size):
            ans.append(matrix[m//2][i+k])
    return ans
```

Examples:

LC 54. Spiral Matrix
LC 48. Rotate Image

5.6. Stack

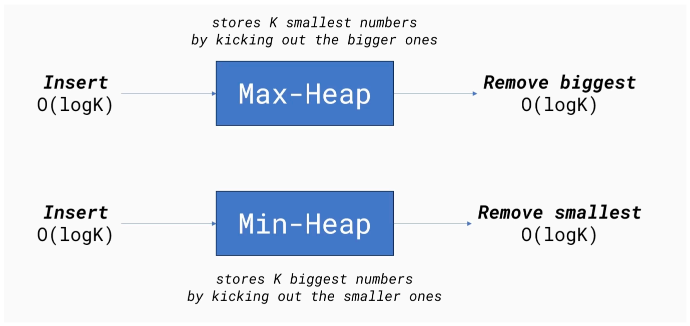
Examples:
LC 224. Basic Calculator

5.7. Heap

Heap is good for finding the k-th largest or smallest element without sorting the entire array. The idea is to keep a heap of size k, popping out all the elements that do not meet the requirement.

Time Complexity: O(nlogk)
Space Complexity: O(k)

Keep in mind that to find the k-th largest element, you use a min heap, to find the k-th smallest element, you use a max heap.


stores K smallest numbers by kicking out the bigger ones
Insert O(logK)
Max-Heap
Remove biggest O(logK)
Insert O(logK)
Min-Heap
Remove smallest O(logK)
stores K biggest numbers by kicking out the smaller ones

Problem: Given a list of values, we want to manipulate the frequencies of them step by step and always approach the value with max/min frequency.
Challenge:

1. You want to access and manipulate the frequency by value in O(1) time. (hashmap)
2. You want to update and order the pairs by frequency. (heap)
3. Everytime you change the frequency of a specific value, how to update the frequency of that value in the heap since you cannot access heap in O(1) time

Solution:
You add the pair based on the frequency in counter and push it into the heap. To have duplicate (frequency, value) pair in the heap, if you find the frequency of the tip instance in the heap is different from that in the counter, you pop it since it's outdated.

```
def mostFrequentIDs(self, nums: List[int], freq: List[int]) -> List[int]:
    count = Counter()
    h = []
    ans = []
    for num, fre in zip(nums, freq):
        count[num] += fre
        heappush(h, [-count[num], num])
        while h and h[0][0] != -count[h[0][1]]:
            heappop(h)
        ans.append(-h[0][0])
    return ans
```

Example:

- LC 347. Top K Frequent Elements
- LC 295. Find Median from Data Stream
- LC 218. The Skyline Problem
- LC 3092. Most Frequent IDs

5.8. Monotonic Queue/Stack (Deque)

用于找到当前元素左边或右边比它大或小的第一个元素，或找滑动窗口中最大值或最小值
原则：

1. 保证栈里的元素是递增或递减的，可存元素的值或下标位置
2. 新元素一定要放在栈内

栈用来存放遍历过的元素
如果要找第一个大于某元素的元素，从栈口到栈底应单调递增
如果要找第一个小于某元素的元素，从栈口到栈底应单调递减
https://leetcode.com/problems/number-of-valid-subarrays/editorial/

Monotonic stacks are used to **calculate the previous smaller (greater) element and the next smaller (greater) element in linear time complexity**. Basically given an array arr in range (0, n), return the range (i, j) in which each element is the smallest (greatest), where arr[i]<arr[element]<=arr[j].

Edge Case - Duplicate Elements. We should make sure that we don't count the contribution by an element twice. While finding the left boundary, we look for elements that are **strictly less than** the current element. To decide the right boundary, we look for the elements which **are less than or equal to** the current element.

Let’s take the monotonic increasing stack as an example:

Step

1. Push 0 into stack
2. Iterate over the array:
    a. While stack and stack[-1] < array[i]: pop from stack and update results
    b. Push i into stack

```
class Solution {
    public int monoIncStack(int[] arr) {
        Stack<Integer> stack = new Stack<>();
        long ans = 0;

        // building monotonically increasing stack
        for (int i = 0; i <= arr.length; i++) {

            // when i reaches the array length, it is an indication that
            // all the elements have been processed, and the remaining
            // elements in the stack should now be popped out.

            while (!stack.empty() && (i == arr.length || arr[stack.peek()]
>= arr[i])) {

                // Notice the sign ">=", This ensures that no contribution
                // is counted twice. rightBoundary takes equal or smaller
                // elements into account while leftBoundary takes only the
                // strictly smaller elements into account

                int mid = stack.pop();
                int leftBoundary = stack.empty() ? -1 : stack.peek();
                int rightBoundary = i;

                // Do the computation that is desired
                ans = func(ans, arr, mid, leftBoundary, rightBoundary)
            }
            stack.push(i);
        }

        return ans;
    }
}

# python version
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        stack = [0]
        n = len(temperatures)
        ans = [0]*n
        for i in range(1, n):
            while stack and temperatures[i]>temperatures[stack[-1]]:
                idx = stack.pop()
                ans[idx] = i-idx
            stack.append(i)
        return ans

# BST Version
class Solution:
    def verifyPreorder(self, preorder: List[int]) -> bool:
        min_limit = -inf
        stack = []
        for num in preorder:
            while stack and stack[-1]<num:
                min_limit = stack.pop()
            if num<=min_limit:
                return False
            stack.append(num)
        return True
```

Examples:

- LC 255. Verify Preorder Sequence in Binary Search Tree
- LC 84. Largest Rectangle in Histogram
- LC 42. Trapping Rain Water
- LC 239. Sliding Window Maximum

5.9. Greedy

Type 1: Given a **list of possible bugget**, ask for the least number of operations get to the end. Examples are jump game 2 and video stitiching.

The basic idea is to define a variable that tracks the maximum coverage of the current step, every bugget within the range of coverage is the potential next step. When we loop over the array, if the loop catches up with the coverage, we need to greedyly adopt the maximum potential next coverage, and the number of operations increase by 1.

Stitching and jump game, for a minimum count

A general greedy solution to process similar problems

```
class Solution {
    public int videoStitching(int[][] clips, int time) {
        Arrays.sort(clips, (a,b)->a[0]-b[0]);
        int start = 0, end = 0, far_can_reach = 0, ans = 0;
        while(end<time) {
            ans++;
            while(start<clips.length && clips[start][0]<=end) {
                far_can_reach = Math.max(far_can_reach, clips[start][1]);
                start++;
            }
            System.out.println(far_can_reach);
            if(end==far_can_reach) return -1;
            end = far_can_reach;
        }
        return ans;
    }
}
```

Time complexity: O(n)
Space complexity: O(1)

Type 2: Find the overlapping intervals. The input is an array of intervals, which is a list of (start, end) pairs. It usually asks to merge the overlapping intervals; to find the smallest interval set that all intervals overlap with at least one of them;

The solution idea is the first sort the list either by the start or the end, Loop over the array, find the array the start of which is later than the prior end.

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort()
        start, end = intervals[0]
        ans = []
        n = len(intervals)
        for i in range(1, n):
            if intervals[i][0]>end:
                ans.append([start, end])
                start, end = intervals[i]
            else:
                end = max(end, intervals[i][1])
        ans.append([start, end])
        return ans
```

```
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        points.sort(key=lambda x:(x[1], x[0]))
        n = len(points)
        ends = points[0][1]
        ans = 0
        for i in range(n):
            if points[i][0] <= ends:
                continue
            ans += 1
            ends = points[i][1]
        ans += 1
        return ans
```

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

Examples:

- LC 45. Jump Game II
- LC 406. Queue Reconstruction by Height
- LC 738. Monotone Increasing Digits
- LC 968. Binary Tree Cameras（子节点返回状态 ）
- LC 135. Candy (同时两边比较会顾此失彼，要两个循环分别比较左右)
- LC 1024. Video Stitching

6. Dynamic Programming

Window Sliding Technique

6.1. Toxonomization

Recall that there are two different techniques we can use to implement a dynamic programming solution; memorization and tabulation.

- **Memoization** is where we add caching to a function (that has no side effects). In dynamic programming, it is typically used on recursive functions for a top-down solution

that starts with the initial problem and then recursively calls itself to solve smaller problems.
- **Tabulation** uses a table to keep track of subproblem results and works in a bottom-up manner: solving the smallest subproblems before the large ones, in an iterative manner. Often, people use the words "tabulation" and "dynamic programming" interchangeably.

In many programming languages, iteration is faster than recursion. Therefore, we often want to convert a top-down memoization approach into a **bottom-up dynamic programming** one (some people go directly to bottom-up, but most people find it easier to come up with a recursive top-down approach first and then convert it; either way is fine).

背包问题
打家劫舍
股票问题
子序列问题

Solve steps:
- DP cell definition
- Deduction function
- DP initialization
- Loop order (outer/inner, forward/backward)
- Print DP

6.2. Knapsack Problem

m weights napsack, most value
0-1 KP: n types, each 1 item
Complete KP: n types, each inf items
Multi-KP: n types, each different numbers of items

6.3. 0-1 Knapsack Problem

Input:
weights list[n], values list[n], knapsack capacity m
All these values should be >= 0, you can only pick an item once, fix knapsack capacity

Two loop method:

```
dp = []
dp = [[0]*(m+1) for _ in range(n+1)]
for i in range(n): # The inner and outer loop are exchangeable, since the order to update the values is from top left to bottom right
    for j in range(m+1):
        dp[i+1][j] = dp[i][j]
        if j>=weights[i]: Otherwise, there’s out of bounds error
```

```
            dp[i+1][j] = max(dp[i+1][j], dp[i][j-weights[i]]+values[i])
return dp[n][m]
```

One loop method:

```
dp = [0]*(m+1)
for i in range(n): # Can not switch inner and outer loop since the
definition of the dp is row based, otherwise overwritten.
    for j in reversed(range(weights[i], m+1)): # this has to be in reverse
order since your update of value is based on previous row and you don’t
want to overwrite it when you do forward probing. Otherwise, an item can be
picked multiple times.
        dp[j] = max(dp[j], dp[j-weights[i]]+values[i])
return dp[m]
```

There are three variants:

- Fix knapsack capacity, get max value
    - Last Stone Weight II
    - dp[i+1][j] = max(dp[i][j], dp[i][j-weights[i]]+values[i])
- Fix knapsack capacity, get whether it’s possible to reach capacity
    - Partition Equal Subset Sum
    - dp[i+1][j] = dp[i][j] or dp[i][j-weights[i]]
- Fix knapsack capacity, get how many ways to fill in the knapsack
    - Target sum
    - dp[i+1][j] = dp[i][j] + dp[i][j-weights[i]]

6.4. Complete Knapsack Problem

Input:
weights list[n], values list[n], knapsack capacity m
All these values should be >= 0, you can pick an item for inf times, fix knapsack capacity

- Pure problem

Nested loop method:

```
dp = []
dp = [[0]*(m+1) for _ in range(n+1)]
for i in range(n): # The inner and outer loop are not exchangeable
    for j in range(1, m+1):
        dp[i+1][j] = dp[i][j]
        if j>=weights[i]: Otherwise, there’s out of bounds error
```

```
            dp[i+1][j] = max(dp[i+1][j], dp[i+1][j-weights[i]]+values[i])
return dp[n][m]
```

1D array method:

```
dp = [0]*(m+1)
for i in range(n):
    for j in range(weights[i], m+1): # this has to be in increasing order
        dp[j] = max(dp[j], dp[j-weights[i]]+values[i])
return dp[m]
```

There are three variants:

- get how many combinations
  - Coin Change II
  - Inner napsack, outer items, to ensure same item being inserted consecutively
- get how many combinations with order
  - Combination Sum IV
  - Inner items, outer napsack, same item can be inserted separately
- Fix knapsack capacity, get how many ways to fill in the knapsack
  - Target sum
  - dp[i+1][j] = dp[i][j] + dp[i+1][j-weights[i]]

6.5. Two Pointers

The two pointers technique is a common algorithmic approach that involves using two pointers (or indices) to iterate through a data structure, often to find pairs or subarrays that meet certain conditions. Unlike sliding windows, the two pointers technique doesn't necessarily maintain a fixed window size but instead relies on the relative movement of the two pointers.

Template:

```
def two_pointers(arr):
    left = 0
    right = len(arr) - 1

    while left < right:
        # Perform operations with arr[left] and arr[right]

        # Example condition to move pointers
        if condition:
            left += 1
        else:
            right -= 1

    return result
```

Partition Array:

```
def partition(arr, pivot):
    left, right = 0, len(arr) - 1

    while left <= right:
        while left <= right and arr[left] < pivot:
            left += 1
        while left <= right and arr[right] >= pivot:
            right -= 1
        if left < right:
            arr[left], arr[right] = arr[right], arr[left]
            left += 1
            right -= 1

    return left
```

Intersection of two sorted arrays:

```
def intersection_of_two_sorted_arrays(arr1, arr2):
    i, j = 0, 0
    intersection = []

    while i < len(arr1) and j < len(arr2):
        if arr1[i] == arr2[j]:
            if not intersection or arr1[i] != intersection[-1]:
                intersection.append(arr1[i])
            i += 1
            j += 1
        elif arr1[i] < arr2[j]:
            i += 1
        else:
            j += 1

    return intersection
```

Examples:

- Two Sum (Sorted Array): Find two numbers such that they add up to a specific target.

- Remove Duplicates from Sorted Array: Remove duplicates in-place in a sorted array.
- Trapping Rain Water: Calculate the total water that can be trapped after raining.
- Container With Most Water: Find two lines that together with the x-axis form a container, such that the container contains the most water.
- Valid Palindrome: Check if a given string is a palindrome considering only alphanumeric characters and ignoring cases.
- Partition Array: Partition an array into two parts based on a pivot element.
- Find Intersection of Two Sorted Arrays: Find the intersection of two sorted arrays.

LC 15. 3Sum

6.6. Merge Interval

Another variant of two pointers is the merge intervals tasks. The "merge intervals" problem involves merging overlapping intervals into a single interval.

Algorithm:

- Sort the Intervals: Start by sorting the intervals based on the starting point of each interval. This helps in easily identifying overlapping intervals.
- Merge Overlapping Intervals: Iterate through the sorted intervals, and for each interval:
    - If the current interval overlaps with the previous one (i.e., the start of the current interval is less than or equal to the end of the previous interval), merge them by updating the end of the previous interval.
    - If it doesn't overlap, add the previous interval to the result list and move to the next interval.

```
def merge_intervals(intervals):
    if not intervals:
        return []

    # Sort the intervals based on the starting time
    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]

    for current in intervals[1:]:
        prev = merged[-1]
        # If the current interval overlaps with the merged interval, merge
them
        if current[0] <= prev[1]:
            prev[1] = max(prev[1], current[1])
        else:
            # If it doesn't overlap, add the current interval to the merged
list
            merged.append(current)

    return merged
```

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

To insert a new interval into the sorted intervals:

```
def insert(self, intervals: List[List[int]], newInterval: List[int]) ->
List[List[int]]:
    n = len(intervals)
    i = 0
    res = []

    # Case 1: No overlapping before merging intervals
    while i < n and intervals[i][1] < newInterval[0]:
        res.append(intervals[i])
        i += 1

    # Case 2: Overlapping and merging intervals
    while i < n and newInterval[1] >= intervals[i][0]:
        newInterval[0] = min(newInterval[0], intervals[i][0])
        newInterval[1] = max(newInterval[1], intervals[i][1])
        i += 1
    res.append(newInterval)

    # Case 3: No overlapping after merging newInterval
    while i < n:
        res.append(intervals[i])
        i += 1

    return res
```

Examples:
LC 56. Merge Intervals
LC 57. Insert Interval

6.7. Cycle/Intersection Detection

https://leetcode.com/problems/find-the-duplicate-number/editorial/

Floyd's Tortoise and Hare (Cycle Detection) Algorithm: consists of two phases and uses two pointers, usually called tortoise (slow pointer) and hare (fast pointer).

General strategy:

1. (Phase 1) Initialize slow, fast pointers as starting point
2. Iterate end through the iterable objects,
    a. fast pointer moves twice the speed of slow pointer
    b. If fast pointer meets slow pointer
        i. Mark this point as **intersection point**; break
        ii. Slow travels (F+a), fast travels F+nC+a. Thus, 2(F+a)=F+nC+a
3. (Phase 2) Initialize slow as starting point, fast pointers as intersection point (F+a)
4. Iterate end through the iterable objects,
    a. fast pointer moves the same speed of slow pointer
    b. If fast pointer meets slow pointer
        i. Mark this point as **Entrance of Cycle**; return
        ii. Slow at F, fast at nC+F, which is F

```
low = fast = root
while True:
    slow = slow.next
    fast = fast.next.next
    if slow == fast:
        break

slow = root
while slow != fast:
    slow = slow.next
    fast = fast.next
return slow (or fast)
```

Examples:
LC 202. Happy Number

6.8. Sliding Window

Intuition: reduce the use of nested loops and replace it with a **single loop,** thereby reducing the **time complexity.**

Sliding windows problems usually are able to be solved using fixed-size Deque structure which allows you to drop the outdated elements from the front and drop insignificant elements from the end. For monotonic deque, You want to ensure the deque window only has decreasing (increasing) elements. That way, the leftmost element is always the largest (smallest).

Sliding window general strategy

5. Initialize start, answer, check_status values;
6. Iterate end through the iterable objects
    a. Update check_status based on end
    b. While check_status is invalid
        i. Update check_status based start
        ii. Shrink (increment start) window
    c. Update answer based on start and end
7. Finalize and return answer

Example:
LC 3. Longest Substring Without Repeating Characters
LC 30. Substring with Concatenation of All Words
LC 76. Minimum Window Substring
LC 209. Minimum Size Subarray Sum
LC 239. Sliding Window Maximum (monotonic deque)

```
queue = []
    max_collection = []

    for i in range(len(nums)):
        while len(queue) > 0 and queue[-1] < nums[i]:
            queue.pop()
        queue.append(nums[i])

        if i+1 >= k:
            max_collection.append(queue[0])
            if queue[0] == nums[i+1-k]:
                queue.pop(0)

    return max_collection
```

Intuition: In any sliding window based problem we have two pointers. One right pointer whose job is to expand the current window and then we have the left pointer whose job is to contract a given window. At any point in time, only one of these pointers moves and the other remains fixed.

The solution is pretty intuitive. We keep expanding the window by moving the right pointer. When the window between left and right pointers meets all the desired requirements. we contract (moving the left pointer) and save the optimal window.

Examples:

- Maximum Sum Subarray of Size K: Find the maximum sum of a subarray of size K.
- Longest Substring Without Repeating Characters: Find the length of the longest substring without repeating characters.
- Minimum Window Substring: Find the minimum window in a string that contains all characters of another string.
- Subarrays with Given Sum: Find the number of subarrays with a given sum.
- Longest Subarray with Sum at Most K: Find the length of the longest subarray with sum at most K.
- Longest Subarray with Sum Exactly K: Find the length of the longest subarray with sum exactly K.

LC 76. Minimum Window Substring (HashMap)

```
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if not t or not s:
            return ""

        dict_t = Counter(t)
        char_counts = len(dict_t.keys())
        ans = inf, None, None

        l, r = 0, 0
        formed = 0
        window_counts = Counter()

        while r < len(s):
            ch_r = s[r]
            window_counts[ch_r] += 1

            if ch_r in dict_t and window_counts[ch_r] == dict_t[ch_r]:
                formed += 1

            while l<=r and formed == char_counts:
                ch_l = s[l]

                if r-l+1 < ans[0]:
                    ans = (r-l+1, l, r)

                window_counts[ch_l] -= 1
                if ch_l in dict_t and window_counts[ch_l] < dict_t[ch_l]:
                    formed -= 1
                l += 1
            r += 1
        return "" if ans[0] == inf else s[ans[1]:ans[2]+1]
```

6.9. Finite State Machine

Create one dp to store each finite state's optimal cost at time i.
at each time state, update the result of each state from the result of time i-1, save them in dp.

```
class Solution {
    public long[] minimumCosts(int[] state1_cost, int[] state2_cost, int
transfer_cost) {
        int n = state1_cost.length;
        int[] ans = new int[n];
        int[] dp_state1 = new int[n+1];
        int[] dp_state2 = new int[n+1];
        dp_state1 = 0;
        dp_state2 = expressCost;
        for(int i=1;i<=n;i++) {
            dp_state1[i] = Math.min(dp_state1[i-1]+state1_cost[i-1],
dp_state2[i-1]+state2_cost[i-1]);
            dp_state2[i] =
Math.min(dp_state1[i-1]+state1_cost[i-1]+expressCost,
dp_state2[i-1]+state2_cost[i-1]);
            ans[i-1] = Math.min(dp_state1[i], dp_state2[i]);
        }
        return ans;
    }
}
```

6.10. Bit Mask

Bit mask is usually used to solve “minimum subset cover problem ”, which means to select a minimum number of candidates that the join of them fulls all the the requirements.

What is bitmasking? Bitmasking is used to indicate the selection of subsets of all candidates. For the bit part, every candidate is encoded as a single bit, so all states of potential selections can be encoded as a group of bits, i.e. a binary number. For the mask part, we use 0/1 to represent the binary state of selecting something. In most cases, 1 stands for the valid state while 0 stands for the invalid state.

Usually for n candidates, all states are from “0” to “1<<n -1”, the initialization can be:

```
int[] dp = new int[1 << n];
Arrays.fill(dp, -1);
dp[0] = 0;
```

“Integer.bitCount(x)” counts the number of selected candidates at current state.
“(1 << n) - 1”, to get ending mask that all nodes are set to true
“(x >> i) & 1” to get i-th bit in state x.
“x | (1 << i);” to mark i-th candidate as selected in current state x.
“x ^ (1 << i)” tp flip the bit at position i
“x = target_x & ~baseline_x” The set target_x \ baseline_x denotes the set difference, containing the target state but not in the baseline state.
“(x & (x - 1) == 0)”, Brian Kernighan's method to check if a mask x has only one bit set to 1. This is because, in those cases, the mask is in the form of 100...000, and mask - 1 in binary is 0111...111. The AND of these two numbers will be 0 as there are no positions where both bits are set to 1.

6.11. Game Theory

Game theory problems are usually in the setting that player A and B take turns to do some action inorder to maximize their scores.They can be solved using dynamic programming (DP) or memoization to optimize recursive solutions. The key idea is to simulate all possible moves and use the results of subproblems to find the optimal strategy.

The big idea is,

- We keep the record of the maximum score of current situation
- For any choice of current player, if we select any of it, we need to maximize current gain but minimize the remainning gain which is the score of the next player. Basically, use current option score - next remainning score.

We take LC 486. Predict the Winner as an example:

Top-down solution:

```
def predictTheWinner(self, nums: List[int]) -> bool:
    @cache
    def dp(left, right):
        if left == right:
            return nums[left]
        return max(nums[left] - dp(left + 1, right), nums[right] - dp(left, 
right - 1))
    return dp(0, len(nums)-1)>=0
```

Top-down solution with player identifier:

```
def predictTheWinner(self, nums: List[int]) -> bool:
    @cache
    def dp(flag, left, right):
        if left > right:
            return 0
        if flag > 0:
            return max(nums[left] + dp(-flag, left+1, right), nums[right] + 
dp(-flag, left, right-1))
        else:
            return min(-nums[left] + dp(-flag, left+1, right), -nums[right] 
+ dp(-flag, left, right-1))
    return dp(1, 0, len(nums)-1)>=0
```

Bottom-up solution:

```
# We fill the diagonal first and than trying to fill the next diagonal 
towards the top right.
def predictTheWinner(self, nums: List[int]) -> bool:
    n = len(nums)
    dp = [[0] * n for _ in range(n)]
    for i in range(n):
        dp[i][i] = nums[i]

    for diff in range(1, n):
        for left in range(n - diff):
            right = left + diff
            dp[left][right] = max(nums[left] - dp[left + 1][right], 
nums[right] - dp[left][right - 1])

    return dp[0][n - 1] >= 0
```

Time complexity: O(n^2)
Space complexity: O(n^2)

The Game theory questions
LC 486. Predict the Winner
LC 877 Stone Game
LC 1908. Game of Nim

6.12. Simulation

https://leetcode.com/problems/champagne-tower/editorial/

7. Tree

7.1. Resources

Introduction to Trie
Backtracking Algorithms

7.2. Binary Tree Traversal

Level Order transversal: Use BFS style of probing, we need a queue to save the nodes in each level.

```
def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        q = [root]
        ans = []
        while q:
            layer = []
            size = len(q)
            for _ in range(size):
                cur = q.pop(0)
                if not cur:
                    continue
                layer.append(cur.val)
                q.append(cur.left)
                q.append(cur.right)
            if layer:
                ans.append(layer)
        return ans
```

Recursive tranversal:
Here we give an example of inorder tranversal, for preorder and postorder, you need to swap the order of those lines of code.

```
def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    ans = []
    def dfs(node):
        if not node:
            return
        dfs(node.left)
        ans.append(node.val)
        dfs(node.right)
    dfs(root)
    return ans
```

Iterative tranversal of a tree need the assist of stack.

- Preorder: stack initialized with root, push order (mid, right, left), pop and add value before children push
- Postorder: stack initialized with root, push order (mid, left, right), pop and add value before children push, reverse the order of final answer
- Inorder: stack initialized with empty, use a cur pointer point to root

```
Preorder tranversal:
def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    stack = [root]
    ans = []
    while stack:
        cur = stack.pop()
        if cur:
            ans.append(cur.val)
        else:
            continue
        stack.append(cur.right)
        stack.append(cur.left)
    return ans

Postorder tranversal:
def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    stack = [root]
    ans = []
    while stack:
        cur = stack.pop()
        if cur:
            ans.append(cur.val)
        else:
            continue
        stack.append(cur.left)
        stack.append(cur.right)
    return ans[::-1]
```

```
Inorder tranversal:
def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    stack = []
    ans = []
    cur = root
    while cur or stack:
        while cur:
            stack.append(cur)
            cur = cur.left
        cur = stack.pop()
        ans.append(cur.val)
        cur = cur.right
    return ans
```

Examples:
LC 98. Validate Binary Search Tree

7.3. Binary Tree Construction

You're given two traversals of the same BT, and you're asked to reconstruct the tree.
The key point is, based on the order property of the traversals, locate the root node, split traversals into chunks and recursively construct subtree.

Approach:

1. Use either the preorder or postorder trasversal as an anchor list. (easy finding root)
2. Use the other traversal as a reference list, a map of value to index should be learnt
3. In the recursive loop:
    a. If left index is greater than the right index for the reference list, exit
    b. Construct the root from the anchor list
    c. Locate the position of root in the reference list
    d. Split the reference list

    e. Split the anchor list (because of the length equality)
    f. Recursively process the matched substring

Here's an example or building a tree from inorder and postorder traversal

```
def buildTree(inorder: List[int], postorder: List[int]) ->
Optional[TreeNode]:
    postIndex = len(postorder)-1
    inorder_map = {}
    for i, val in enumerate(inorder):
        inorder_map[val] = i

    def dfs(in_left, in_right):
        if in_left>in_right:
            return None
        nonlocal postIndex

        rootval = postorder[postIndex]
        root = TreeNode(rootval)
        postIndex -= 1

        index = inorder_map[rootval]

        root.right = dfs(index+1, in_right)
        root.left = dfs(in_left, index-1)

        return root

    return dfs(0, len(inorder)-1)
```

Examples:
LC 105. Construct Binary Tree from Preorder and Inorder Traversal
LC 106. Construct Binary Tree from Inorder and Postorder Traversal
LC 889. Construct Binary Tree from Preorder and Postorder Traversal

7.4. Binary Search Tree

Key points:

- Inorder traversal of BST is an array sorted in the ascending order. (According to this rule, we can easily construct an inorder transversal of BST. If you're given another traversal, either preorder or postorder, you can use the two traversals together to construct a BST.)
- The left subtree are all smaller than root, the right subtree are all greater than root. This can be used to simplify transversal

- The right most node of the left subtree is the node that's smaller than and closest to root, the left most node of the right subtree is the node that's greater than and closest to root.

Recursive Search:

```
def searchBST(self, root: Optional[TreeNode], val: int) ->
Optional[TreeNode]:
    if not root:
        return None
    if val == root.val:
        return root
    elif val>root.val:
        return self.searchBST(root.right, val)
    else:
        return self.searchBST(root.left, val)
```

Iterative Search:

```
def searchBST(self, root: Optional[TreeNode], val: int) ->
Optional[TreeNode]:
    while root:
        if val>root.val:
            root = root.right
        elif val<root.val:
            root = root.left
        elif val==root.val:
            return root
        else:
            return None
```

Insert:

```
def insertIntoBST(self, root: Optional[TreeNode], val: int) ->
Optional[TreeNode]:
    if root is None:
        return TreeNode(val)
    if val>root.val:
        root.right = self.insertIntoBST(root.right, val)
    elif val<root.val:
        root.left = self.insertIntoBST(root.left, val)
    return root
```

Delete:

```
def deleteNode(self, root: Optional[TreeNode], key: int) ->
Optional[TreeNode]:
    if root is None:
        return root
    if root.val == key:
        if root.right:
            node = root.right
            while node.left:
                node = node.left
            node.left = root.left
            return root.right
        elif root.left:
            return root.left
        else:
            return None
    elif root.val < key:
        root.right = self.deleteNode(root.right, key)
    else:
        root.left = self.deleteNode(root.left, key)
    return root
```

Binary search tree heavily dependent on inorder transversal, for example, check if a binary tree is a valid binary search tree.

```
# Recursive method
def isValidBST(self, root: TreeNode) -> bool:

    def inorder(root):
        if not root:
            return True
        if not inorder(root.left):
            return False
        if root.val <= self.prev:
            return False
        self.prev = root.val
        return inorder(root.right)
    self.prev = -math.inf
    return inorder(root)

# Iterative method
def isValidBST(self, root: Optional[TreeNode]) -> bool:
    stack, prev = [], -inf

    while stack or root:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()

        if root.val <= prev:
            return False
        prev = root.val
        root = root.right

    return True
```

Examples:
LC 1008. Construct Binary Search Tree from Preorder Traversal

7.5. Trie

Trie is used to solve the prefix problems, which is to build a dictionary of the tree structures to store all target words. **It makes finding words sharing the same prefix faster**.

1. Nary tree structure, where each Trie node represents a string (a prefix), children nodes represent different successive characters.
2. Prefix tree, the root node is associated with the empty string, and all the descendants of a node have a common prefix of the string associated with that node.
3. The corresponding relationship between characters and children nodes can be represented using an array (fast, easy access, waste of space) or map (flexible, save space)
4. TreeNode consists of, (value), children TreeNodes, and an end-word flag.
5. Basic operations of TreeNode, constructor, contains-key, get, put, set-end, is-end.
6. Basic operations of Trie constructor (empty TreeNode), insert, search
7. In questions, firstly, define TreeNode, then construct Trie, and lastly search in batches.

Array representation:

```
class TrieNode {
    // change this value to adapt to different cases
    public static final N = 26;
    public TrieNode[] children = new TrieNode[N];

    // you might need some extra values according to different cases
};

/** Usage:
 *  Initialization: TrieNode root = new TrieNode();
 *  Return a specific child node with char c: root.children[c - 'a']
 */
```

Map representation:

```
class TrieNode {
    public Map<Character, TrieNode> children = new HashMap<>();

    // you might need some extra values according to different cases
};

/** Usage:
 *  Initialization: TrieNode root = new TrieNode();
 *  Return a specific child node with char c: root.children.get(c)
 */
```

Implement a Trie:

```
class TrieNode{
    private TrieNode[] links;
    private final int R = 26;
    private boolean isEnd;
    public TrieNode() {
        links = new TrieNode[R];
    }
    public boolean containsKey(char ch) {
        return links[ch -'a'] != null;
    }
    public TrieNode get(char ch) {
        return links[ch -'a'];
    }
    public void put(char ch, TrieNode node) {
        links[ch -'a'] = node;
    }
    public void setEnd() {
        isEnd = true;
    }
    public boolean isEnd() {
        return isEnd;
    }
}

class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char currentChar = word.charAt(i);
            if (!node.containsKey(currentChar)) {
                node.put(currentChar, new TrieNode());
            }
            node = node.get(currentChar);
        }
        node.setEnd();
    }

    public boolean search(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
           char curLetter = word.charAt(i);
           if (node.containsKey(curLetter)) {
               node = node.get(curLetter);
           } else {
               return false;
           }
        }
        return node.isEnd();
    }

    public boolean startsWith(String prefix) {
    # Time Complexity: O(m)
    # Space Complexity: O(1)
        TrieNode node = root;
        for (int i = 0; i < prefix.length(); i++) {
            char curLetter = prefix.charAt(i);
            if (node.containsKey(curLetter)) {
                node = node.get(curLetter);
            } else {
                return false;
            }
        }
        return true;
    }
}

class Trie:

    def __init__(self):
        self.N = 26
        self.root = TrieNode(self.N)

    def insert(self, word: str) -> None:
        node = self.root
        for ch in word:
            idx = ord(ch) - ord('a')
            if not node.field[idx]:
                node.field[idx] = TrieNode(self.N)
            node = node.field[idx]
        node.isEnd = True

    def search(self, word: str) -> bool:
        node = self.root
        for ch in word:
            idx = ord(ch) - ord('a')
            if node.field[idx]:
                node = node.field[idx]
            else:
                return False
        return node.isEnd

    def startsWith(self, prefix: str) -> bool:
        node = self.root
        for ch in prefix:
            idx = ord(ch) - ord('a')
            if node.field[idx]:
                node = node.field[idx]
            else:
                return False
        return True
```

Time Complexity: O(M*N*K), M words, N length, K characters
Space Complexity: O(M*N*K), M words, N length, K characters

Simplest Trie implementation is a simple dictionary.

```
trie = {}
for i, word in enumerate(words):
    node = trie
    for ch in word:
        if ch not in node:
            node[ch] = {}
            node = node[ch]
    node['$'] = i
```

Example questions:
LC 208. Implement Trie (Prefix Tree)
LC 212. Word Search II
LC 425. Word Squares

7.6. DFS

For a tree, we have the following traversal methods using DFS:

- **Preorder**: visit each node before its children.
- **Postorder**: visit each node after its children.
- **Inorder** (for binary trees only): visit left subtree, node, right subtree.

There are generally two methods: recursive and iterative:
Recursive pseudo code (preorder):

```
procedure preorder(treeNode v)
{
  visit(v);
  for each child u of v
    preorder(u);
}
```

The iterative DFS is similar to the iterative BFS but differs from it in following ways:

- It uses a **stack** instead of a queue.
- The DFS should mark discovered only after popping the vertex, not before pushing it.
- It uses a reverse iterator instead to produce the same results as recursive DFS.

Iterative pseudo code (preorder):

```
procedure preorder(treeNode v)
{
    Stack node_stack;
    node_stack.offer(v)
    while node_stack:
        v = node_stack.pop()
        visit(v);
        for each child u of v
            node_stack.push(u);
}
```

Recursive rooted tree java code (postorder):

```
class Solution {
    public List<Integer> postorder(Node root) {
        List<Integer> ans = new ArrayList<>();
        if (root == null)
            return ans;
        dfs(root, ans);
        return ans;
    }
    public void dfs(Node root, List<Integer> ans) {
        if(root==null)
            return;
        for(Node child : root.children) {
            dfs(child, ans);
        }
        ans.add(root.val);
    }
}
```

Iterative rooted-tree java code (postorder):

```
class Solution {
    public List<Integer> postorder(Node root) {
        LinkedList<Node> stack = new LinkedList<>();
        LinkedList<Integer> ans = new LinkedList<>();
        if (root == null) {
            return ans;
        }
        stack.add(root);
        while (!stack.isEmpty()) {
            Node node = stack.pollLast();
            ans.addFirst(node.val);
            for (Node item : node.children) {
                if (item != null) {
                    stack.add(item);
                }
            }
        }
        return ans;
    }
}
```

If the tree is non-rooted, then instead of children list of each node, we're provided with the neighbors of each node. We can start from any random node and consider it as the root, the children of nodes are the neighbors excluding the parent of the node.

Recursive non-rooted tree java code (postorder):
Input: graph (adjacency list)
Procedure: post_order_dfs(0, -1);

```
public void post_order_dfs(int node, int parent) {
    for(int neiborgh : graph.get(node)) {
        // neiborghs that is not the parent node are children of non-rooted
tree
        if(neiborgh!=parent) {
            post_order_dfs(neiborgh, node);
            visit(node);
        }
    }
}
```

Recursive non-rooted tree DFS java code (postorder):

```
class Solution:
  def minTime(self, n: int, edges: List[List[int]], hasApple: List[bool]) -> int:
    adj = [[] for i in range(n)]
    for edge in edges:
      adj[edge[0]].append(edge[1])
      adj[edge[1]].append(edge[0])
    return self.dfs(0, -1, adj, hasApple)
```

```
def dfs(self, node, parent, adj, hasApple):
    totalTime = 0
    childTime = 0
    for child in adj[node]:
        if child==parent:
            continue
        childTime = self.dfs(child, node, adj, hasApple)
        if childTime>0 or hasApple[child]:
            totalTime += childTime + 2
    return totalTime
```

7.7. BackTrack

回溯总是和递归结合在一起，形成一个n叉树，本质是嵌套for循环
常见回溯能解决的问题类型有：

- 组合问题
- 切割问题
- 子集问题
- 子序列问题
- 排列问题
- 棋盘问题

三步走：
确定参数和返回值，确定终止条件，单层递归逻辑
关于返回值，搜索所有可行解，用void，搜索单个可行解，用bool
组合template

```
def backtrack(start, end, cur_solution, cur_stat, termination_rule):
    If termination_rule:
        collect solution
        return
    for node in (start, end):
        process node to cur_solution
        cur_stat += node
        bracktrack(start+1, end, cur_solution, cur_stat, termination_rule)
        cur_stat -= node
        remove node from cur_solution
```

组合和无放回排列的区别在于，排列要循环所有不在当前solution内的元素，无放回组合要从当前元素的下一个元素开始循环。
剪枝操作有两个：

1. 循环的结束条件，如果树的深度已经不足以得到可行解，则在循环边界剪枝
2. 当前解的值，如果已经超出目标值的范围，则在终止条件前剪枝

3. 去重，只选择重复值中的最后一个，
   if i>start and candidates[i]==candidates[i-1]: continue
   也可以用layer wised 的 used数组来去重

Note:

- 每一步在同一个集合中取选项，使用startIndex，在不同集合中取选项，使用index
- 有放回和无放回的区别在于，无放回遍历从当前元素开始，有放回只遍历当前之后元素。
- 树层去重，只选择重复值中的最后一个， if i>start and candidates[i]==candidates[i-1]: continue

分割问题template

```
def partition(self, s: str) -> List[List[str]]:
    def check_切割条件(s):
        for i in range(len(s)//2):
            if ...:
                return False
        return True

    def backtrack(start, sol):
        if start == len(s):
            ans.append(sol[:])
            return
        cur = []
        for i in range(start, len(s)):
            cur.append(s[i])
            if check_切割条件(cur):
                sol.append("".join(cur))
                backtrack(i+1, sol)
                sol.pop()
        return
    ans = []
    backtrack(0, [])
    return ans
```

startIndex 告诉我们下一层切割的起始位置，用于判断是在当前切割中增加元素，还是重新开启一个新的切割。

子集问题：
返回list of list，三层嵌套
和组合问题的区别：子集问题，每个节点都有需要的结果，所以每个节点都要收集结果，每层递归都要收集结果，而非只在叶子节点取结果。

子集template:

```
def subsets(self, nums: List[int]) -> List[List[int]]:
    def backtrack(start, cur):
        ans.append(cur[:])
        for i in range(start, len(nums)):
            cur.append(nums[i])
            backtrack(i+1, cur)
            cur.pop()
        return
    ans = []
    backtrack(0, [])
    return ans
```

子序列问题template:

- 每个元素只有取或不取两种情况，在节点收集结果而非叶子
- 在每一层上定义一个set，use this set to remove duplicates

```
def findSubsequences(self, nums: List[int]) -> List[List[int]]:
    def backtrack(start, cur):
        if len(cur)>=2:
            ans.append(cur[:])
        used = set()
        for i in range(start, len(nums)):
            if (nums[i] in used) or (len(cur)>0 and nums[i]<cur[-1]):
                continue
            used.add(nums[i])
            cur.append(nums[i])
            backtrack(i+1, cur)
            cur.pop()
            # one used set for every layer, do not backtrack it
    ans = []
    backtrack(0, [])
    return ans
```

排列问题template:

- 循环中每一层元素，不是从startindex开始取，而是对于所有选项，选择还未在当前解中选择的去取，也就是树枝去重，需要用一个可回溯的set(全局变量)检测，维持所有保存在当前解中的元素
- 只在叶子结点收集结果
- 也可以用Answer set来去重

```
def permute(self, nums: List[int]) -> List[List[int]]:
    used = [False]*len(nums)
    def backtrack(cur):
        if len(cur)==len(nums):
            ans.append(cur[:])
        for i in range(len(nums)):
            if used[i]:
               continue
            cur.append(nums[i])
            used.add(i)
            backtrack(cur)
            used.remove(i)
            cur.pop()
        return
    ans = []
    backtrack([])
    return ans
```

Examples:
[LC 51. N-Queens]
[LC 52. N-Queens II]
[LC 37. Sudoku Solver]
[LC 79. Word Search]

Backtrack is a special case in DFS, by terminating a failed path and switching to other possible searching paths. It's trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.

It can be used to solve the following questions:

- Decision Problem – In this, we search for a feasible solution.
- Optimization Problem – In this, we search for the best solution.
- Enumeration Problem – In this, we find all feasible solutions.

The function can be broken down into the following four steps:

- Step 1). Check if we reach the bottom case of the recursion (EXIT THE RECURSION).
- Step 2). Check if the current state is invalid, out of boundary, do not match searching target requirement.
- Step 3). If the current step is valid, mark the current state as visited, or change the state to target.
- Step 4). Start the exploration of backtracking with the strategy of DFS. Iterate through all the children in the next level. If True, break the loop.
- Step 5). At the end of the exploration, revert the state back to its original state.

- Step 6). Return the result of the exploration.

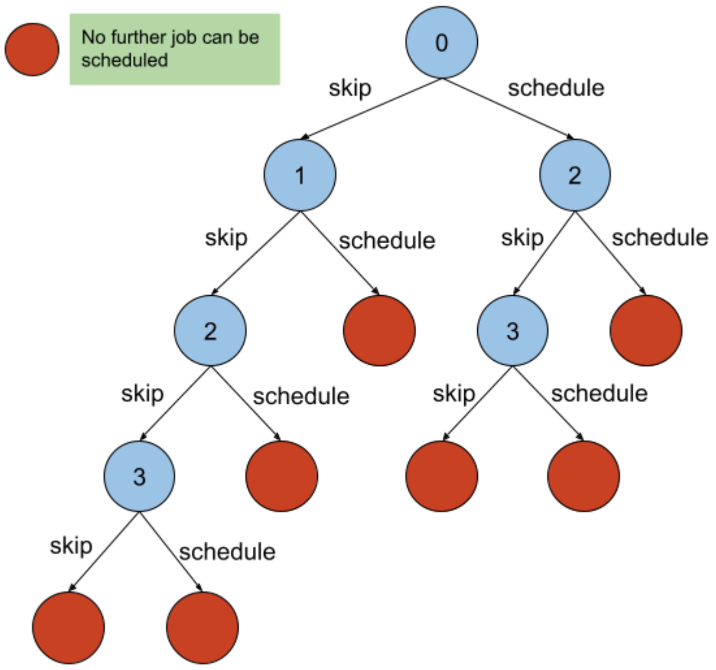
```
class Solution {
  private char[][] board;
  private int ROWS;
  private int COLS;

  public boolean exist(char[][] board, String word) {
    this.board = board;
    this.ROWS = board.length;
    this.COLS = board[0].length;

    for (int row = 0; row < this.ROWS; ++row)
      for (int col = 0; col < this.COLS; ++col)
        if (this.backtrack(row, col, word, 0))
          return true;
    return false;
  }

  protected boolean backtrack(int row, int col, String word, int index) {
    if (index >= word.length())
      return true;

    if (row < 0 || row == this.ROWS || col < 0 || col == this.COLS
        || this.board[row][col] != word.charAt(index))
      return false;

    boolean ret = false;
    // mark the path before the next exploration
    this.board[row][col] = '#';

    int[] rowOffsets = {0, 1, 0, -1};
    int[] colOffsets = {1, 0, -1, 0};
    for (int d = 0; d < 4; ++d) {
      ret = this.backtrack(row + rowOffsets[d], col + colOffsets[d], word,
index + 1);
      if (ret)
        break;
    }

    this.board[row][col] = word.charAt(index);
    return ret;
    }
}
```

7.8. DFS Memorization

Memorization is used to look up the calculated result in **constant time** by storing the subprblem's results and **avoid** recalculating repeated subproblems.

This recursive approach will have repeated subproblems; this can be observed in the figure below. Notice, the subtree with root 2 is repeated signifying that we must **solve this subproblem more than once**.
To address this issue, the first time we calculate maxProfit for a certain position, we will store the value in an array; this value represents the maximum profit we can get from the jobs at indices from position to the end of the array. The next time we need to calculate maxProfit for this position, we can look up the result in constant time.

Job	0	1	2	3
startTime	1	2	3	3
endTime	3	4	5	6



```
class Solution {
    // maximum number of memories are 50000
    int[] memo = new int[50001];

    // jobs:  a list of items for searching and scheduling, this is a
varying sized list
    // n: the termination flag indicating the end of all jobs
    // position: the position indicator of which level of the tree the
current searching is at
    private int findMaxProfit(List<List<Integer>> jobs, int n, int
position) {
        // terminate because of reaching the end of search list
        if (position == n) {
            return 0;
        }

        // return result directly if it is calculated
        if (memo[position] != -1) {
            return memo[position];
        }

        // nextIndex is the index of next non-conflicting job
        int nextIndex = findNextJob(jobs);

        // find the maximum profit of our two options: skipping or
scheduling the current job
        int maxProfit = Math.max(findMaxProfit(jobs, n, position + 1),
                        jobs.get(position).get(2) + findMaxProfit(jobs, n,
nextIndex));

        // return maximum profit and also store it for future reference
(memoization)
        return memo[position] = maxProfit;
    }

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit)
{
        // marking all values to -1 so that we can differentiate  if we
have already calculated the answer or not

        Arrays.fill(memo, -1);
        List<List<Integer>> jobs = new ArrayList<>();
        jobs = createJobs(startTime, endTime);
        int n = startTime.length;

        return findMaxProfit(jobs, n, 0);
    }
}
```

One issue with the previous method is the recursive calls incurred stack space. This can be avoided by applying the same approach in an **iterative** manner starting from the base case (in **reverse** logical order) which is generally faster than the top-down recursive approach.

```
class Solution {
    // maximum number of jobs are 50000
    int memo[] = new int[50001];

    private int findMaxProfit(List<List<Integer>> jobs, int[] startTime) {
        int length = startTime.length;

        for (int position = length - 1; position >= 0; position--) {
            // it is the profit made by scheduling the current job
            int currProfit = 0;

            // nextIndex is the index of next non-conflicting job
            int nextIndex = findNextJob(startTime,
jobs.get(position).get(1));

            // Fetch the recursive next result, if there is the next job is
not out of bound add it's profit else just consider the current job profit
            if (nextIndex != length) {
                currProfit = jobs.get(position).get(2) + memo[nextIndex];
            } else {
                currProfit = jobs.get(position).get(2);
            }

            // terminate case and storing the maximum profit we can achieve
by scheduling jobs from index position to the end of array
            if (position == length - 1) {
                memo[position] = currProfit;
            } else {
                memo[position] = Math.max(currProfit, memo[position + 1]);
            }
        }

        return memo[0];
    }

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit)
{
        List<List<Integer>> jobs = new ArrayList<>();
        jobs = createJobs(startTime, endTime);
        // no need to reinitialize the memo with -1

        return findMaxProfit(jobs, startTime);
    }
}
```

Example problems:
LC 1235. Maximum Profit in Job Scheduling
LC 95. Unique Binary Search Trees II

7.9. BFS

7.10. BFS Memorization

7.11. Binary Indexed Tree

7.12. Segment Tree

Segment Tree Data Structure - Min Max Queries - Java source code
Segment Tree Range Minimum Query
Lazy Propagation Segment Tree
A segment tree is a binary tree used for storing intervals or segments. It allows querying which of the stored segments contain a given point and is efficient for answering range queries and updates. Segment trees are particularly useful for problems where you need to perform multiple range queries and updates on an array.

Basic Segment Tree Operations
- Build: Construct the segment tree from a given array. O(n)
- Query: Query a range to get information (e.g., sum, minimum, maximum, sum). O(log n)
- Update: Update an element or a range of elements. O(log n)

Keynotes:
- Segment Tree is a data structure that facilitates fast **range queries**, such as finding the minimum, maximum, or sum across a range of numbers.

- It is constructed by computing the desired operation (e.g., max) between sequential pairs of elements, and repeating the process at higher levels until the entire range is covered.
- The implementation uses an additional array **twice** the size of the input array, where the bottom half stores the input elements, and the top half stores the **computed values** at different levels.
- Querying the max/min value in a given range involves walking up the tree, checking only the necessary nodes that cover the range.
- The time complexity for construction and queries is O(n) and O(log n), respectively, where n is the size of the input array.

For example, you spent one year counting the customer flow if the entire street. Now if you want to know the accumulated number of customers in a time range for example from Jan to March, you may use segment tree and only collect the data that's related.

If you have the numbers simply stored in an array, you want to achieve a max range query.

- A naive solution: Iterate over every element in the array in the desired range. Query time O(n)
- A speed-up solution: Create a two dimensional lookup table and pre-compute the max between every possible date range looking up a value, Construct time O(n^2), Space O(n^2), query time O(1)
- Segment Tree solution: Construct time & Space O(n), query time O(log(n))

Query given query range R and node range r:

- If R completely overlaps r: return precomputed value of r
- If R partially overlaps r: propagate the children of r
- If R does not overlap r: return the counterpart of r (max query return a minimum value, min query return a maximum value, sum query return a 0)

Code template for range sum:

- Note that, for every normal range, the left should be divisible by 2, the right should not be divisible by 2.

```
class SegmentTree:

    def __init__(self, nums: List[int]):
        self.n = len(nums)
        self.tree = [None] * (self.n * 2)
        for i in range(self.n):
            self.tree[i+self.n] = nums[i]
        for i in range(self.n-1, 0, -1):
            self.tree[i] = self.tree[i * 2] + self.tree[i * 2 + 1]

    def update(self, index: int, val: int) -> None:
        pos = index + self.n
        self.tree[pos] = val
        while pos > 1:
            left = pos
            right = pos
            if pos % 2 == 0:
                right = pos + 1
            else:
                left = pos - 1
            self.tree[pos // 2] = self.tree[left] + self.tree[right]
            pos //= 2

    def sumRange(self, left: int, right: int) -> int:
        left += self.n
        right += self.n
        ans = 0
        while left <= right:
            if left % 2 == 1:
                ans += self.tree[left]
                left += 1
            if right % 2 == 0:
                ans += self.tree[right]
                right -= 1
            left //= 2
            right //= 2
        return ans
```

Lazy propagation:
This is an optimization technique on segment tree when there are a lot of updates, **it miminize the number of nodes to be updated**.

Without lazy propagation, we go all the way towards the leaf nodes.
When lazy propagation is in place, we need to **keep a lazy tree** (a copy of segment tree with default values) which stores the unapplied operations.

Whenever we reached a node in update or query, we need to check whether the past updates are applied (if the value in the lazy tree is default value). If there’s a copmlete overlap, we update the intermediate node, store the updates in its child nodes in the lazy tree, and return the updated value. (we don’t propagate to the leaves.)

```
class SegmentTree:

    def __init__(self, nums: List[int]):
        self.n = len(nums)
        self.tree = [0] * (2 * self.n)
        self.lazy = [0] * (2 * self.n)
        self._build(nums)

    def _build(self, nums: List[int]):
        for i in range(self.n):
            self.tree[self.n + i] = nums[i]
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

    def _apply(self, pos: int, val: int, length: int):
        # updates the node and marks it for lazy propagation.
        self.tree[pos] += val * length
        if pos < self.n:
            self.lazy[pos] += val

    def _push(self, pos: int):
        # ensures that updates are propagated down the tree before any
query or further updates.
        for s in range(self.n.bit_length(), 0, -1):
            i = pos >> s
            if self.lazy[i] != 0:
                self._apply(i * 2, self.lazy[i], (i * 2 + 1) - i * 2)
                self._apply(i * 2 + 1, self.lazy[i], (i * 2 + 2) - (i * 2 +
1))
                self.lazy[i] = 0

    def updateRange(self, left: int, right: int, val: int):
        left += self.n
        right += self.n
        l0, r0 = left, right
        length = 1
        while left <= right:
            if left % 2 == 1:
                self._apply(left, val, length)
                left += 1
            if right % 2 == 0:
                self._apply(right, val, length)
                right -= 1
            left //= 2
            right //= 2
            length *= 2
        self._push(l0)
        self._push(r0)

    def sumRange(self, left: int, right: int) -> int:
        left += self.n
        right += self.n
        self._push(left)
        self._push(right)
        ans = 0
        while left <= right:
            if left % 2 == 1:
                ans += self.tree[left]
                left += 1
            if right % 2 == 0:
                ans += self.tree[right]
                right -= 1
            left //= 2
            right //= 2
        return ans
```
Examples:
[LC 307. Range Sum Query - Mutable]()

7.13. Combination

Backtracking can be used to find all the combinations of a list of elements. For example, given a list nums of n elements, we want to find all the combinations of k of them. The backtrack function usually takes 4 inputs: start (the current location of the pointer on the list), temp_list (the current collection of selected elements), nums (the list of all elements to choose from), k (the ending criteria indicates for example the total number of elements should be selected).
Note:

- The difference from permutation is, the loop starts from starts which means all the seen elements are not considered again to ensure no ordering issue.
- k may mean the remaining number of elements so every time to call backtrack we deduct k by 1

```
def find_combinations(nums, k):
    result = []
    def backtrack_combinations(temp_list, nums, start, k):
        # Base case: if the combination is of length k, add it to the
result
        if len(temp_list) == k:
            result.append(temp_list[:])
            return
        # Prune the cases if the remaining elements to choose from is not
enough for completing k
        if n-start+1 < (k-len(temp_list)):
                return

        for i in range(start, len(nums)):
            # Include nums[i] in the current combination
            temp_list.append(nums[i])
            # Recur with the next element and incremented combination
length
            backtrack_combinations(temp_list, nums, i + 1, k)
            # Backtrack: remove the last element to try another combination
            temp_list.pop()

    backtrack_combinations([], nums, 0, k)
    return result
```

Time Complexity: O(C(n, k) * k)
Space Complexity: O(C(n, k) * k)

7.14. Permutation

To generate all possible permutations of k elements from a list of n elements, we can use backtracking. This problem is a variation of the combination problem but involves ordering the elements within each subset, making it a permutation problem. The backtrack function takes 3 inputs, temp_list (the current collection of selected elements), nums (the list of all elements to choose from), k (the ending criteria indicates for example the total number of elements should be selected).

Note:

- The difference from combination is, the loop starts always from the beginning which means we allow first seen elements to appear later.
- We need to keep a used set to make sure the used elements not using again.

```
def find_k_permutations(nums, k):
    result = []
    used = [False] * len(nums)  # To track used elements

    def backtrack_permutations(temp_list, nums, k):
        # Base case: if the permutation is of length k, add it to the
result
        if len(temp_list) == k:
            result.append(temp_list[:])
            return

        for i in range(len(nums)):
            if used[i]:
                continue

            # Include nums[i] in the current permutation
            temp_list.append(nums[i])
            used[i] = True

            # Recur with the next element
            backtrack_permutations(temp_list, nums, k)

            # Backtrack: remove the last element and mark it as unused
            temp_list.pop()
            used[i] = False

    backtrack_permutations([], nums, k)
    return result
```

Time Complexity: O(A(n, k) * k)
Space Complexity: O(A(n, k) * k)

8. Graph

8.1. Resources

8.2. Graph Representations

Edges representation:

```
edges = []
for row in range(n):
    for col in range(row+1, n):
        if connected[row][col] == 1:
            edges.append([row, col])
```

Construct graph from edges

```
graph = defaultdict(list)

for (u,v) in edges:
    graph[u].append(v)
```

8.3. Connected Components

Method1: DFS algorithm with visited

Idea: To run DFS starting from a particular vertex, it will continue to visit the vertices depth-wise until there are no more adjacent vertices left to visit. Each time we finish exploring a connected component, we can find another vertex that has not been visited yet, and start a new DFS from there. The number of times we start a new DFS will be the number of connected components.

Keynotes:

- Visited flag should be marked at the beginning of each DFS as an exit rule
- Check visited flag before go into any subtree DFS

Algorithm:

- Step 1). Create an adjacency list such that adj[v] contains all the adjacent vertices of vertex v. Initialize a hashmap or array, visited, to track the visited vertices.
- Step 2). Define a counter variable and initialize it to zero.
- Step 3). Iterate over each vertex in edges, and if the vertex is not already visited, start a DFS from it. Add every vertex visited during the DFS to visited.
- Step 4). Every time a new DFS starts, increment the counter variable by one.
- Step 5). At the end, the counter variable will contain the number of connected components in the undirected graph.

```
class Solution {

     private void dfs(List<Integer>[] adjList, int[] visited, int
startNode) {
        visited[startNode] = 1;

        for (int i = 0; i < adjList[startNode].size(); i++) {
            if (visited[adjList[startNode].get(i)] == 0) {
                dfs(adjList, visited, adjList[startNode].get(i));
            }
        }
    }

    public int countComponents(int n, int[][] edges) {
        int components = 0;
        int[] visited = new int[n];

        List<Integer>[] adjList = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            adjList[i] = new ArrayList<Integer>();
        }

        for (int i = 0; i < edges.length; i++) {
            adjList[edges[i][0]].add(edges[i][1]);
            adjList[edges[i][1]].add(edges[i][0]);
        }

        for (int i = 0; i < n; i++) {
            if (visited[i] == 0) {
                components++;
                dfs(adjList, visited, i);
            }
        }
        return components;
    }
}
```

Time complexity O(E+V), space complexity O(E+V)

Method2: Union and find

Examples:
[Number of Connected Components in an Undirected Graph]

8.4. Union Find

Union-Find Algorithm can be used

1. Check whether an undirected graph contains a cycle or not
2. Group connected components in undirected graph

Note that the input to union find algorithm to run is pairs of edges.

```
int[] parents = new int[n];
for(int i=0;i<n;i++){
    parents[i]=i;
}
private int find(int x) {
    if(parents[x]!=x)
        return find(parents[x])
    return x;
}
private void union(int x, int y){
    parents[find(y)]=find(x);
}
```

Time complexity O(n), space complexity O(n)
These methods can be improved to O(logN) using Union by Rank or Height.

```
class UnionFind:
    def __init__(self, size: int) -> None:
        self.group = [0] * size
        self.rank = [0] * size
        self.components = size

        for i in range(size):
            self.group[i] = i

    def find(self, node: int) -> int:
        if self.group[node] != node:
            self.group[node] = self.find(self.group[node]) # compression
        return self.group[node]

    def join(self, node1: int, node2: int) -> bool:
        group1 = self.find(node1)
        group2 = self.find(node2)

        # node1 and node2 already belong to same group.
        if group1 == group2:
            return False # It means there exist a cycle.

        self.components -= 1
        if self.rank[group1] > self.rank[group2]:
            self.group[group2] = group1
        elif self.rank[group1] < self.rank[group2]:
            self.group[group1] = group2
        else:
            self.group[group2] = group1 # union by rank
            self.rank[group1] += 1

        return True

    def check_unique(self):
        splits = set()
        for i in range(self.size):
            group = self.find(i)
            if group not in splits:
                splits.add(group)
        return splits

    def connected(self, node1, node2):
        return self.find(node1) == self.find(node2)

def numOfConnected(self, isConnected: List[List[int]]) -> int:
    n = len(isConnected)
    uf = UnionFind(n)
    edges = []
    for row in range(n):
            for col in range(row+1, n): # Cannot do row+1 as it may cause
error in self.group if you use len(set(self.group)) as components
                if isConnected[row][col] == 1:
                    edges.append((row, col))
    return uf.components
```

Path Compression: This technique is used during the find operation to make the tree flatter, speeding up future operations. When you call find on an element, you make all nodes on the path from that element to the root point directly to the root.

Union by Rank (or Size): This technique is used during the union operation to ensure that the smaller tree (in terms of rank or size) is always added under the root of the larger tree, keeping the overall tree shallow.

Time complexity: Constructor O(N), Find, Union, Connected O(α(n)), per operation, where α(n) is the inverse Ackermann function.
Space complexity: O(α(n))

There are two variants of Union Find:
- Quick Find O(1) + Union O(N)
- Find O(N) + Quick Union O(N)
- Union by rank: Find Olog(N), Union Olog(N)

8.5. Minimum Spanning Tree

Resources:
https://leetcode.com/problems/min-cost-to-connect-all-points/solution/
https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm
https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

A spanning tree is a subset of a graph that includes all the graph's vertices and some of the edges of the original graph, intending to have no cycles. A spanning tree is not necessarily unique.

Method1: Prim's Algorithm (BFS)
- Step 1: Determine an arbitrary vertex as the starting vertex of the MST. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE.
- Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
- Step 3: Find edges connecting any tree vertex with the fringe vertices.
- Step 4: Find the minimum among these edges.
- Step 5: Add the chosen edge to the MST if it does not form any cycle.
- Step 6: Return the MST and exit

Note:
1. We start with node 0 by adding a virtual edge of cost 0 from node 0 to node 0. This makes the code implementation cleaner.
2. Keep track of the nodes that's in the MST for every step

Time complexity: O(V^2), O(ElogV) using heap
Space complexity: O(V)

Not using heap:

```
class Solution:
    def MST(self, vertices: List[List[int]]) -> int:
        n = len(points)
        mst_cost = 0
        edges_used = 0

        # Track nodes which are visited.
        in_mst = [False] * n

        min_dist = [math.inf] * n
        min_dist[0] = 0

        while edges_used < n:
            curr_min_edge = math.inf
            curr_node = -1

            # Pick least weight node which is not in MST.
            for node in range(n):
                if not in_mst[node] and curr_min_edge > min_dist[node]:
                    curr_min_edge = min_dist[node]
                    curr_node = node

            mst_cost += curr_min_edge
            edges_used += 1
            in_mst[curr_node] = True

            # Update adjacent nodes of current node.
            for next_node in range(n):
                weight = abs(points[curr_node][0] - points[next_node][0])
+\
                         abs(points[curr_node][1] - points[next_node][1])

                if not in_mst[next_node] and min_dist[next_node] > weight:
                    min_dist[next_node] = weight

        return mst_cost
```

Using heap:

```
def prim_mst(graph, start_node=0):
    n = len(graph)
    min_heap = [(0, start_node)]  # (weight, vertex)
    visited = [False] * n
    mst_cost = 0
    edges_in_mst = []

    while min_heap and len(edges_in_mst)<n:
        weight, u = heappop(min_heap)

        if visited[u]:
            continue

        # Include this edge in MST
        visited[u] = True
        mst_cost += weight
        edges_in_mst.append((weight, u))

        # Add all edges from u to the heap
        for v, cost in graph[u]:
            if not visited[v]:
               heappush(min_heap, (cost, v))

    # Return the total cost of the MST and the edges included in the MST
    return mst_cost, edges_in_mst
```

The graph is {src_node : [tgt_node, edge_cost] … }

Method2: Kruskal's Algorithm (Union Find)

- Step 1: Sort all edges in increasing order of their edge weights.
- Step 2: Pick the smallest edge.
- Step 3: Check if the new edge creates a cycle or loop in a spanning tree (Union Find).
- Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.
- Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.
- Step 6: Return the MST and exit

Time complexity: O(ElogE)
Space complexity: O(V)

```
class UnionFind:
    def __init__(self, size: int) -> None:
        self.group = [0] * size
        self.rank = [0] * size

        for i in range(size):
            self.group[i] = i

    def find(self, node: int) -> int:
        if self.group[node] != node:
            self.group[node] = self.find(self.group[node])
        return self.group[node]

    def join(self, node1: int, node2: int) -> bool:
        group1 = self.find(node1)
        group2 = self.find(node2)

        # node1 and node2 already belong to same group.
        if group1 == group2:
            return False

        if self.rank[group1] > self.rank[group2]:
            self.group[group2] = group1
        elif self.rank[group1] < self.rank[group2]:
            self.group[group1] = group2
        else:
            self.group[group1] = group2
            self.rank[group2] += 1

        return True

class Solution:
    def MST(self, vertices: List[List[int]]) -> int:
        n = len(vertices)
        all_edges = []

        # Storing all edges of our complete graph and their weights.
        for curr_node in range(n):
            for next_node in range(curr_node + 1, n):
                weight = abs(points[curr_node][0] - points[next_node][0])
+\
                              abs(points[curr_node][1] -
points[next_node][1])
                all_edges.append((weight, curr_node, next_node))
        # Sort all edges in increasing order.
        all_edges.sort()

        uf = UnionFind(n)
        mst_cost = 0
        edges_used = 0

        for weight, node1, node2 in all_edges:
            if uf.join(node1, node2):
                mst_cost += weight
                edges_used += 1
                if edges_used == n - 1:
                    break
        return mst_cost
```

Prim's Algorithm vs Kruskal's Algorithm
Prim's algorithm is good for densely connected graph Kruskals's algorithm is good for sparsely connected graph

8.6. DFS

In Graph theory, the depth-first search algorithm (abbreviated as DFS) is mainly used to:

- Traverse all vertices in a connected "graph";
- Traverse all paths between any two vertices in a "graph".

Visit All Vertices:
The most important difference from DFS on trees is, it has to maintain a visited set to avoid visiting the same node multiple times.

Recursive DFS:

- Start at a given node (the root).
- Mark the current node as visited.
- For each adjacent node (neighbor) that has not been visited, recursively apply DFS.

```
def dfs_recursive(graph, start, visited):
    visited.add(start)
    process(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)
dfs_recursive(graph, 0, set())
```

Iterative DFS:

- Created a stack of nodes and visited arrays.
- Start at a given node (the root).
- Push the start node onto a stack.
- While the stack is not empty:
    - Pop a node from the stack.
    - If the node has not been visited, mark it as visited.
    - Push all its unvisited neighbors onto the stack.

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            process(node)
            # Add unvisited neighbors to the stack
            for neighbor in reversed(graph[node]):  # Reverse to maintain
order similar to recursive
                if neighbor not in visited:
                    stack.append(neighbor)
```

Time Complexity: O(V + E). Here, VV represents the number of vertices, and EE represents the number of edges. We need to check every vertex and traverse through every edge in the graph.

Space Complexity: O(V^2). Either the manually created stack or the recursive call stack can store up to V vertices. Or O(V) if check visited before pushing into the recursive stack.

Worst case: A complete graph is a graph where every vertex is connected to every other vertex.

Visit All Edges Between A Pair of Vertices:
We need the assist of a recursive stack which saves the vertices visited during this traversal. Recursive stack is implemented by backtracking.

```
def dfs_all_paths(graph, start, end, curr_path, recursive_stack,
all_paths):
    curr_path.append(start)
    recursive_stack[start] = True
    if start == end:
        all_paths.append(curr_path[:])
    else:
        for neighbor in graph[start]:
            # Check to avoid cycles
            if not recursive_stack[neighbor]:
                dfs_all_paths(graph, neighbor, end, curr_path,
recursive_stack, all_paths)
    curr_path.pop()
    recursive_stack[start] = False
    return all_paths
```

Time complexity: O((V-2)!)
Space complexity: O(V^3)

8.7. BFS

In Graph theory, the primary use cases of the "breadth-first search" ("BFS") algorithm are:

- Traversing all vertices in the connected "graph";
- Finding the shortest path between two vertices in a graph where all edges have equal and positive weights.

The most important difference from BFS on trees is, it has to maintain a visited set to avoid visiting the same node multiple times. As long as we use BFS, the first time we add target vertex, that is the shortest path from start to target.

Iterative method:

- Start at a given node (the root).
- Enqueue the start node.
- While the queue is not empty:
    - Dequeue a node from the front of the queue.
    - If the node has not been visited, mark it as visited.
    - Enqueue all its unvisited neighbors.

```
def bfs_iterative(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            process(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
```

Recursive method: BFS can be implemented recursively using a helper function to manage the queue. This implementation is less common and can be more complex due to the need to pass the queue and visited set through recursive calls.

- Start at a given node (the root).
- Enqueue the start node and call the recursive function.
- In the recursive function:
    - If the queue is empty, skip
    - Dequeue a node from the front of the queue.
    - If the node has not been visited, mark it as visited.
    - Enqueue all its unvisited neighbors.
    - Recursively call the function.

```
def bfs_recursive(graph, queue, visited):
    if not queue:
        return
    node = queue.popleft()
    if node not in visited:
        visited.add(node)
        process(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)
    bfs_recursive(graph, queue, visited)

# Wrapper function to start the recursive BFS
def bfs_recursive_start(graph, start):
    visited = set()
    queue = deque([start])
    bfs_recursive(graph, queue, visited)
```

Time Complexity: O(V + E). Here, V represents the number of vertices, and E represents the number of edges. We need to check every vertex and traverse through every edge in the graph. The time complexity is the same as it was for the DFS approach.

Space Complexity: O(V). Generally, we will check if a vertex has been visited before adding it to the queue, so the queue will use at most O(V) space. Keeping track of which vertices have been visited will also require O(V) space.

8.8. Dijkstra's Algorithm

Dijkstra’s algorithm is used on weighted graph, computing the shortest path from a single vertex to all the other vertices.

Note: All edges should be positive

```
class Solution:
    def Dijkstra(self, n, edges: List[List[int]]) -> List[int]:
        # graph is an adjacency matrix with key to be a vertex, value to be a
(target vertex, edge weight) pair

        dist = [inf]*n
        visited = [False] * n
        dist[0] = 0 # optimal distance is pre-added, before adding to queue
        Q = [(0, 0)]  # The min heap saves (edge weight, vertex) pairs

        While Q:
            c, u = heappop(Q) # c is the current distance from start to u
            if visited[u]:
                continue
            visited[u] = True
            for neighbor in graph[u]: # For undirected version, you need keep
a visited set in order to avoid visiting u when u has already visited
                v = neighbor[0]
                if visited[v]:
                    continue
                alt = c + neightbor[1]
                if alt < dist[v]:
                   dist[v] = alt # pre-added distance
                   heappush(Q, (alt, v))
        return dist
```

Time complexity with min-heap O(V+ElogV), without min-heap O(V^2), the worst case is a complete graph where each vertex is connected to many other vertices

Space complexity is O(V)

If we want to know the shortest path to a certain target and return the trace, we can use a prev array to save the node that's prior to the current node on the path from source to target

```
class Solution:
    def dijkstra(self, n, edges: List[List[int]], target) -> List[int]:
        # graph is an adjacency matrix with key to be a vertex, value to be a
(target vertex, edge weight) pair

        dist = [inf]*n
        dist[0] = 0
        prev = [-1]*n # Initialize a prev array to save the node prior to
vertices
        Q = [(0, 0)]  # The min heap saves (edge weight, vertex) pairs

        While Q:
            c, u = heappop(Q)
            if u == target: # break the loop if the target is found
                break
            if visited[u]:
                continue
            for neighbor in graph[u]:
                v = neighbor[0]
                alt = c + neightbor[1]
                if alt < dist[v]:
                   dist[v] = alt
                   prev[v] = u # update the prev node of v
                   heappush(Q, (alt, v))
        # Find the trace from source to target
        trace = []
        u = target
        if prev[u] != -1 or u==0 # default 0 is source, check if vertex is
reachable
            while u:
                trace.append(u)
                u = prev[u]
        return trace[::-1]
```

If we want to get the minimum distance of k steps,

- we cannot use heap since it could process later steps first than previous ones
- we cannot use visited set, since this won't guarantee optimality

```
def max_k_dijkstra(self, n: int, edges: List[List[int]], src: int, dst:
int, k: int) -> int:
    graph = defaultdict(set)
    for u, v, w in edges:
        graph[u].add((v, w))

    dist = [inf]*n
    dist[src] = 0
    Q = deque([(0, src)])

    while Q:
        if k<0:
            break
        k -= 1
        size = len(Q)
        for _ in range(size):
            c, u = Q.popleft()
            for v, w in graph[u]:
                alt = c + w
                if alt < dist[v]:
                    dist[v] = alt
                    Q.append((alt, v))
    return -1 if dist[dst] == inf else dist[dst]
```

8.9. A* Algorithm

A star algorithm is used on weighted graph, computing starting from a single vertex to find a path to the given single goal node having the smallest cost. The implementation is the same with BFS, but replace the deque with heap, and

A* combines the strengths of Dijkstra's Algorithm and Greedy Best-First-Search by using a heuristic to guide its search. The algorithm uses the following function to determine the order of node exploration:
f(n) = g(n)+h(n) # estimated cost through n to goal, cost to node n, estimated cost from n to goal

Priority Queue: Uses a priority queue to explore nodes based on their f(n).
Open List: Stores nodes to be explored.
Closed Set: Keeps track of nodes already explored.
Heuristic Function: Provides an estimate of the distance from any node to the goal.
Graph Representation: Adjacency list, each node points to its neighbors with edge weights.

```
def a_star(graph, start, goal, heuristic):
    open_list = []
    heappush(open_list, (0 + heuristic(start, goal), 0, start, [])) # keep
record of f(n), g(n), current node, path
    closed_set = set()

    while open_list:
        _, cost, current, path = heappop(open_list)

        if current in closed_set:
            continue

        path = path + [current] # keep recording the path

        if current == goal: # check if goal is reached
            return path

        closed_set.add(current)

        for neighbor, weight in graph[current]:
            if neighbor not in closed_set:
                total_cost = cost + weight
                heappush(open_list, (total_cost + heuristic(neighbor,
goal), total_cost, neighbor, path))

    return None  # No path found

# Heuristic function example (Manhattan distance for grids)
def heuristic(node, goal):
    x1, y1 = node
    x2, y2 = goal
    return abs(x1 - x2) + abs(y1 - y2)
```

Time complexity: O(b^d)

Space complexity: O(b^d)

8.10. Bellman-Ford’s Algorithm

Bellman-Ford’s algorithm is used on weighted graph, computing the shortest path from a single vertex to all the other vertices.

Note:

- It can handle edges with negative weights, although it’s slower
- It can be used to detect a negative cycle

Idea:
The Bellman-Ford algorithm’s primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on “Principle of Relaxation“.

It states that for the graph having N vertices, all the edges should be relaxed N-1 times to compute the single source shortest path.
In the worst-case scenario, a shortest path between two vertices can have at most N-1 edges, where N is the number of vertices.

```
class Solution:
    def bellman_ford(self, n, edges: List[List[int]]) -> List[int]:
        dist = [inf]*n
        dist[0] = 0

        for i in range(1, n): # This has to be N-1 times iteration
            for edge in edges:
                u, v, w = edge # edge definition
                if dist[u] != inf and dist[v] > dist[u] + w:
                    dist[v] = dist[u] + w
        return dist
```

Time complexity: O(VE)
Space complexity: O(V)

This algorithm can only work when all vertices are reachable from the source vertex 0. If not, we have to go over all vertices with a distance of infinity one by one.

To detect a negative cycle: we need to do another loop and check if distances won’t change

```
class Solution:
    def bellman_ford_detect_negative_cycle(self, n, edges: List[List[int]])
-> Boolean:
        dist = [inf]*n
        dist[0] = 0

        for i in range(1, n): # This has to be N-1 times iteration
            for edge in edges:
                u, v, w = edge # edge definition
                if dist[u] != inf and dist[v] > dist[u] + w:
                    dist[v] = dist[u] + w

        # Detect negative cycle by the last cycle
        for e in edges:
            u, v, w = edge
            if dist[u] != inf and dist[v] > dist[u] + w:
                return True
        return False
```

If we want to get the minimum distance of k steps,

- we have to use use a copy of dist array to update the distance step by step

```
def max_k_bellman_ford(self, n: int, edges: List[List[int]], src: int, dst:
int, k: int) -> int:
    graph = defaultdict(set)
    for u, v, w in edges:
        graph[u].add((v, w))

    dist = [inf]*n
    dist[src] = 0

    for i in range(k+1):
        tmp_dist = [c for c in dist]
        for u, v, w in edges:
            if dist[u] == inf:
                continue
            alt = dist[u] + w
            if alt < tmp_dist[v]:
                tmp_dist[v] = alt
        dist = tmp_dist
    return -1 if dist[dst] == inf else dist[dst]
```

8.11. Floyd-Warshall Algorithm

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles) between all pairs of vertices.

The idea of the algorithm:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.

- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.

```
class Solution:
    def floyd_warshall(self, n, edges: List[List[int]]) -> List[List[int]]:
        dist = [[inf]*n for _ in range(n)]
        for i in range(n):
            dist[i][i] = 0
        for edge in edges:
            u, v, w = edge
            dist[u][v] = w

        for k in range(n):
            for i in range(n):
                for j in range(n):
                    dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j])
        return dist
```

Time complexity: O(V^3)
Space complexity: O(V^2)

The algorithm can also be used to detect negative cycles:
The idea is to find a node that the shortest path to itself is smaller than 0

```
class Solution:
    def floyd_warshall_detect_negative_cycle(self, n, edges: List[List[int]])
-> Boolean:
        dist = [[inf]*n for _ in range(n)]
        for i in range(n):
            dist[i][i] = 0
        for edge in edges:
            u, v, w = edge
            dist[u][v] = w

        for k in range(n):
            for i in range(n):
                for j in range(n):
                    dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j])
        # An extra transversal on the vertices to check negative connection
        for i in range(n):
            if dist[i][i]<0:
                return True
        return False
```

8.12. Bipartite Graph

In graph theory, a “cut” is a partition of vertices in a “graph” into two disjoint subsets. A crossing edge is an edge that connects a vertex in one set with a vertex in the other set.

The cut property: For any cut C of the graph, if the weight of an edge E in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MSTs of the graph.

A bipartite graph is a special type of graph that can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to a vertex in V. In other words, no edge exists between vertices within the same set.

To check if a given graph is bipartite, you can use either BFS or DFS to try and color the graph using two colors. If you can successfully color the graph without conflicts, the graph is bipartite.

```
def is_bipartite_dfs(graph):
    n = len(graph)
    colors = [-1]*n

    def dfs(node, c):
        colors[node] = c # color node once visiting
        ans = True
        for neighbor in graph[node]:
            if colors[neighbor]!=-1: # check color before visiting
                if colors[neighbor] == colors[node]:
                    return False
                continue
            ans &= dfs(neighbor, 1-c)
        return ans

    return dfs(i, 0)


def is_bipartite_bfs(graph):
    color = {}
    for start in graph:
        if start not in color:
            queue = deque([start])
            color[start] = 0  # Start coloring with 0
            while queue:
                node = queue.popleft()
                for neighbor in graph[node]:
                    if neighbor not in color:
                        color[neighbor] = 1 - color[node]  # Alternate 
color

                        queue.append(neighbor)
                    elif color[neighbor] == color[node]:
                        return False
    return True
```

Properties of Bipartite Graphs:

- **Two Sets of Vertice**s: Vertices can be divided into two sets U and V.
- **No Odd-Length Cycles**: A graph is bipartite if and only if it does not contain any odd-length cycles.
- **Two-Colorable**: Bipartite graphs can be colored using two colors such that no two adjacent vertices share the same color.

Maximum Bipartite Matching: (also know as **Minimum Vertex Cover**) is the largest set of edges such that no two edges share a common vertex.
We use the **Hopcroft-Karp** algorithm to solve it.

```
class BipartiteGraph:
    def __init__(self, u_size, v_size):
        self.u_size = u_size
        self.v_size = v_size
        self.edges = [[] for _ in range(u_size + v_size + 1)]
        self.pair_u = [-1] * (u_size + 1)
        self.pair_v = [-1] * (v_size + 1)
        self.dist = [-1] * (u_size + 1)

    def add_edge(self, u, v):
        self.edges[u].append(v + self.u_size)
        self.edges[v + self.u_size].append(u)

    def bfs(self):
        queue = []
        for u in range(1, self.u_size + 1):
            if self.pair_u[u] == -1:
                self.dist[u] = 0
                queue.append(u)
            else:
                self.dist[u] = float('inf')
        self.dist[0] = float('inf')
        for u in queue:
            if self.dist[u] < self.dist[0]:
                for v in self.edges[u]:
                    if self.dist[self.pair_v[v - self.u_size]] ==
float('inf'):
                        self.dist[self.pair_v[v - self.u_size]] =
self.dist[u] + 1
                        queue.append(self.pair_v[v - self.u_size])
        return self.dist[0] != float('inf')

    def dfs(self, u):
        if u != 0:
            for v in self.edges[u]:
                if self.dist[self.pair_v[v - self.u_size]] == self.dist[u]
+ 1:
                    if self.dfs(self.pair_v[v - self.u_size]):
                        self.pair_v[v - self.u_size] = u
                        self.pair_u[u] = v - self.u_size
                        return True
            self.dist[u] = float('inf')
            return False
        return True

    def hopcroft_karp(self):
        matching = 0
        while self.bfs():
            for u in range(1, self.u_size + 1):
                if self.pair_u[u] == -1:
                    if self.dfs(u):
                        matching += 1
        return matching
```

Time complexity: O(sqrt(V)· E)

8.13. Detect Cycle in an undirected graph

https://www.geeksforgeeks.org/detect-cycle-undirected-graph/

DFS (with parent):

We need to keep recording the parent node of the current node, since if a visited node is a parent node, then, there's a cycle.

- If an adjacent vertex is already visited and is not the parent of the current vertex, a cycle is detected.

```
def dfs_cycle_detection(graph, vertex, visited, parent):
    visited[vertex] = True
    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            if dfs_cycle_detection(graph, neighbor, visited, vertex):
                return True
        elif neighbor != parent:
            return True
    return False

def has_cycle_dfs(graph):
    visited = {v: False for v in graph}
    for vertex in graph:
        if not visited[vertex]:
            if dfs_cycle_detection(graph, vertex, visited, -1):
                return True
    return False
```

Union-Find Method (Disjoint Set Union - DSU)

Using the Union-Find data structure, you can detect cycles by merging sets of vertices and checking if two vertices belong to the same set.

- If the roots are the same, a cycle is detected.

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return False
        return True

def has_cycle_union_find(edges, n):
    uf = UnionFind(n)
    for u, v in edges:
        if uf.union(u, v):
            return True
    return False
```

8.14. Detect Cycle in a directed graph

Detect Cycle in a Directed Graph

DFS (backtracking with recursion stack):
DFS for cycle detection is based on the idea that there is a cycle in a graph only if there is a **back edge** (i.e., a node points to one of its ancestors) present in the graph.

To detect a back edge, we need to keep track of the **nodes visited till now** and the **nodes that are in the current recursion stack** (i.e., the current path that we are visiting). Recursive stack is implemented by backtracking. (This recursive stack visited set is actually a seen set, which means this vertex is seen during this traversing path but not all neighbor edges of this vertex are explored.)

- If during recursion, we reach a node that is already in the recursion stack, there is a cycle present in the graph.

Note: If the graph is disconnected then get the DFS forest and check for a cycle in individual trees by checking back edges.

Time Complexity: O(V + E)
Space Complexity: O(V)

```
def dfs_cycle_detection(graph, vertex, visited, rec_stack):
    visited[vertex] = True
    rec_stack[vertex] = True

    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            if dfs_cycle_detection(graph, neighbor, visited, rec_stack):
                return True
        elif rec_stack[neighbor]:
            return True

    rec_stack[vertex] = False
    return False

def has_cycle_dfs(graph):
    visited = {v: False for v in graph}
    rec_stack = {v: False for v in graph}

    for vertex in graph:
        if not visited[vertex]:
            if dfs_cycle_detection(graph, vertex, visited, rec_stack):
                return True
    return False
```

Kahn's Algorithm (Topological Sort):
A little change to the original topological sort is:

- If the number of vertices in the topological order is less than the number of vertices in the graph, a cycle exists.

Time Complexity: O(V + E)
Space Complexity: O(V)

```
def has_cycle_kahn(graph):
    in_degree = {u: 0 for u in graph}

    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    queue = deque([u for u in in_degree if in_degree[u] == 0])
    count = 0

    while queue:
        vertex = queue.popleft()
        count += 1

        for neighbor in graph[vertex]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return count != len(graph)
```

Examples:
LC 207. Course Schedule

8.15. Topological Sort (Kahn's algorithm)

topological sort is used for:

- find a global order for all nodes in a DAG (Directed Acyclic Graph) with regarding to their dependencies.
- Detect circles in directed graph

Kahn's algorithm for topological sorting is an iterative approach that uses the concept of in-degrees (number of incoming edges) of vertices. It maintains a list of vertices with zero in-degree and processes them iteratively. It is the same as BFS from zero in-dgree vertices and update the in-degrees in each iteration, we don't need a visited set here.

- Calculate the in-degrees of all vertices.
- Enqueue all vertices with zero in-degree.
- While the queue is not empty:
    - Dequeue a vertex, add it to the topological order.
    - Decrease the in-degree of all its neighbors by 1.
    - If any neighbor's in-degree becomes zero, enqueue it.
- If all vertices are processed, the topological order is complete. If not, the graph contains a cycle.

```
def topological_sort_kahn(graph):
    in_degree = {u: 0 for u in graph}

    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1 # directed edge

    queue = deque([u for u in in_degree if in_degree[u] == 0])
    topological_order = []

    while queue:
        vertex = queue.popleft()
        topological_order.append(vertex)

        for neighbor in graph[vertex]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    if len(topological_order) == len(graph):
        return topological_order
    else:
        return []  # The graph has a cycle and thus no topological order
exists
```

Time complexity: O(V+E), Space Complexity: O(V+E)
Example:
Leetcode 207. Course Schedule
Leetcode 3203. Find Minimum Diameter After Merging Two Trees

Directed Topological Sort

```
class Solution {
    public void DirectedTopologicalSort(int n, int[][] edges) {
        Map<Integer, List<Integer>> adj = new HashMap<>();
        // Has to keep track of all in-degrees
        int[] indegree = new int[n];

        // Adjacent matrix only store the outgoing edges
        for (int[] edge : edges) {
            adj.computeIfAbsent(edge[0], k->new
ArrayList<Integer>()).add(edge[1]);
            indegree[edge[1]]++;
        }
        Queue<Integer> q = new LinkedList<>();

        // Push all the nodes with indegree zero in the queue.
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                q.offer(i);
            }
        }

        int nodesSeen = 0;
        while (!q.isEmpty()) {
            int size = q.size();
            for(int j=0;j<size;j++) {
                int node = q.poll();
                nodesSeen++;

                if (!adj.containsKey(node)) {
                    continue;
                }

                for (int neighbor : adj.get(node)) {
                    indegree[neighbor]--;
                    // When indegree is 0, add to queue
                    if (indegree[neighbor] == 0) {
                        q.offer(neighbor);
                    }
                }
            }
        }
        return;
    }
}
```

Undirected Topological Sort (inverse BFS)

```
class Solution {
    public void UndirectedTopologicalSort(int n, int[][] edges) {
        Map<Integer, List<Integer>> adj = new HashMap<>();
        int[] degree = new int[n];
```

```
          // Adjacent matrix store all edges
         for(int[] edge : edges) { # add bi directions
             adj.computeIfAbsent(edge[0], k->new
ArrayList<>()).add(edge[1]);
             degree[edge[0]]++;
             adj.computeIfAbsent(edge[1], k->new
ArrayList<>()).add(edge[0]);
             degree[edge[1]]++;
         }

         Queue<Integer> q = new LinkedList<>();
         for(int i=0; i<degree.length; i++) {
             if(degree[i] == 1) # only one edge connected
                 q.add(i);
         }

        Int nodeSeen = 0
         while(!q.isEmpty()) {
             int size = q.size();

             for(int i = 0; i < size; i++) {
                 int node = q.poll();
                 degree[node]--;

                 for(int neighbour : adj.get(node)) {
                     degree[neighbour]--;

               // degree to be 1 is the same as in degree to be 0
                     if(degree[neighbour] == 1) {
                         q.add(neighbour);
                     }
                 }
             }
         }
         return;
     }
}
```

Minimum height of undirected tree:

```
def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
```

```
if n<=2: return [i for i in range(n)]
adj = [set() for _ in range(n)]
for u, v in edges:
    adj[u].add(v)
    adj[v].add(u)

leaves = deque([u for u in range(n) if len(adj[u])==1])

remaining = n
while remaining>2:
    new_leaves = []
    remaining -= len(leaves)
    for u in leaves:
        v = adj[u].pop()
        adj[v].remove(u)
        if len(adj[v]) == 1:
            new_leaves.append(v)
    leaves = new_leaves
return leaves
```

9. Eulerian Path (Hierholzer's Algorithm)

https://leetcode.com/problems/reconstruct-itinerary/editorial/

An Eulerian path (or Euler path) in a graph is a path that visits every edge exactly once. If such a path exists and starts and ends at the same vertex, it is called an Eulerian circuit or cycle.

- Eulerian Circuit: An undirected graph has an Eulerian circuit if and only if every vertex has an even degree and all vertices with nonzero degree are connected.
- Eulerian Path: An undirected graph has an Eulerian path if and only if exactly zero or two vertices have an odd degree and all vertices with nonzero degree are connected.

Hierholzer's algorithm (Finding an Eulerian Path) works as follows:

- Start at any vertex with an odd degree. If no such vertex exists, start at any vertex.
- Follow edges one by one. Remove each edge after it is visited. If you reach a vertex with no unvisited edges, backtrack to the previous vertex.
- Continue until all edges have been visited. The path you have followed is the Eulerian path.

Keynotes:

- Use a recursive stack to keep track of the current path.
- Traverse the graph, removing edges as they are visited.

- When reaching a vertex with no unvisited edges, backtrack using the stack.

```
def find_eulerian_path(graph):
    # Count the degree of each vertex
    degree = defaultdict(int)
    for u in graph:
        for v in graph[u]:
            degree[u] += 1
            degree[v] += 1

    # Find the start vertex (one with odd degree or any vertex if none)
    start = None
    odd_degree_vertices = 0
    for vertex in degree:
        if degree[vertex] % 2 == 1:
            odd_degree_vertices += 1
            start = vertex

    if odd_degree_vertices not in [0, 2]:
        return None  # No Eulerian Path or Circuit exists

    if start is None:
        start = next(iter(graph))  # Start from any vertex if all have even
degree

    # Hierholzer's Algorithm to find Eulerian path
    def hierholzer(v):
        path = []
        stack = [v]
        while stack:
            u = stack[-1]
            if graph[u]:
                next_vertex = graph[u].pop()
                graph[next_vertex].remove(u)
                stack.append(next_vertex)
            else:
                path.append(stack.pop())
        return path

    # Find the Eulerian path starting from the start vertex
    path = hierholzer(start)
    return path[::-1]
```

Time complexity: O(E+V)
Space complexity: O(E+V)

10. Hamiltonian Path (Hierholzer's Algorithm)

A Hamiltonian path in a graph is a path that visits each vertex exactly once. If such a path exists and starts and ends at the same vertex, it is called a Hamiltonian cycle. Unlike Eulerian paths, which are concerned with edges, Hamiltonian paths focus on vertices.

Keynote:

- NP-complete problem

Backtracking algorithm to find a hamiltonian path

```
def is_valid(vertex, pos, path, graph):
    # Check if this vertex is an adjacent vertex of the previously added
vertex.
    if vertex not in graph[path[pos - 1]]:
        return False

    # Check if the vertex has already been included in the path.
    if vertex in path:
        return False

    return True

def hamiltonian_path_util(graph, path, pos):
    # Base case: If all vertices are included in the path
    if pos == len(graph):
        return True

    # Try different vertices as the next candidate in the Hamiltonian Path.
    for vertex in graph:
        if is_valid(vertex, pos, path, graph):
            path[pos] = vertex
            if hamiltonian_path_util(graph, path, pos + 1):
                return True
            path[pos] = -1

    return False
```

```python
def find_hamiltonian_path(graph):
    path = [-1] * len(graph)

    # Let the first vertex in the path be the first vertex of the graph.
    # This is arbitrary and can be any vertex.
    start_vertex = next(iter(graph))
    path[0] = start_vertex

    if not hamiltonian_path_util(graph, path, 1):
        return None

    return path
```

Time complexity: O(N!) in the worst case, where N is the number of vertices. This is because the algorithm tries all possible permutations of vertices.
Space complexity: O(N) for the path array and recursion stack.

11. General

11.1. Divide and Conquer

The divide and conquer algorithm is a paradigm for solving complex problems by breaking them down into smaller subproblems, solving each subproblem independently, and then combining their solutions to solve the original problem. This approach is particularly effective for problems that can be recursively divided into similar subproblems.

The frequent questions are summarization or sorting

Key Steps in Divide and Conquer

- Divide: Break the problem into smaller, non-overlapping subproblems of the same type.
- Conquer: Solve each subproblem recursively. If the subproblem size is small enough, solve it directly.
- Combine: Combine the solutions of the subproblems to form the solution to the original problem.

```python
def divide_and_conquer(problem):
    # Base case: if the problem is small enough, solve it directly
    if is_small_enough(problem):
        return direct_solution(problem)

    # Divide the problem into smaller subproblems
    subproblems = divide(problem)

    # Conquer each subproblem recursively
    sub_solutions = []
    for subproblem in subproblems:
        sub_solution = divide_and_conquer(subproblem)
        sub_solutions.append(sub_solution)

    # Combine the solutions of the subproblems to form the solution of the
original problem
    return combine(sub_solutions)
```

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

Examples:
[LC 148. Sort List]

11.2. Streaming

11.3. Hash Function

11.4. Number Theory

The simplest lambda expression contains a single parameter and an expression:

11.5. Probability & Statistics

Problems involving probabilities on platforms like LeetCode often test your understanding of statistical concepts, probability distributions, and sometimes require implementing simulations.

- Random Pick with Weight:

    - Problem: You are given an array of positive integers w where w[i] describes the weight of index i (0-indexed). Write a function pickIndex which randomly picks an index in proportion to its weight.
    - Use prefix and binary search

```python
class Solution:
    def __init__(self, w: List[int]):
        self.prefix_sums = []
        current_sum = 0
        for weight in w:
            current_sum += weight
            self.prefix_sums.append(current_sum)
        self.total_sum = current_sum

    def pickIndex(self) -> int:
        target = random.random() * self.total_sum
        # Binary search for the target zone in the prefix sums
        low, high = 0, len(self.prefix_sums) - 1
        while low < high:
            mid = (low + high) // 2
            if target > self.prefix_sums[mid]:
                low = mid + 1
            else:
                high = mid
        return low
```

- Shuffle an Array:
    - Problem: Implement the Fisher-Yates algorithm to shuffle an array.
    - Iterate over the array and swap each element with a randomly chosen element that comes after it (including itself).

```python
class Solution:
    def __init__(self, nums: List[int]):
        self.original = nums[:]
        self.array = nums[:]

    def reset(self) -> List[int]:
        self.array = self.original[:]
        return self.array

    def shuffle(self) -> List[int]:
        for i in range(len(self.array)):
            swap_idx = random.randrange(i, len(self.array))
            self.array[i], self.array[swap_idx] = self.array[swap_idx],
self.array[i]
        return self.array
```

- Reservoir Sampling:
    - Problem: Given a stream of unknown length, randomly select k elements.
    - Initially fill the reservoir array.
    - For each new element in the stream, randomly decide whether to include it in the reservoir (and if so, replace an existing element).

```
class ReservoirSampling:
    def __init__(self, k: int):
        self.k = k
        self.reservoir = []
        self.n = 0

    def process_element(self, element):
        self.n += 1
        if len(self.reservoir) < self.k:
            self.reservoir.append(element)
        else:
            s = random.randint(0, self.n - 1)
            if s < self.k:
                self.reservoir[s] = element

    def get_sample(self):
        return self.reservoir
```

Probability comoputing problems are usually solved by accumulating the multiplication of probabilities of individual events. This type of problem are combine with DP or graph to get the maximum of minimum probabilities of achieving some goal.

11.6. Voting

https://leetcode.com/problems/majority-element-ii/solution/
https://leetcode.com/problems/majority-element/solution/

Boyer-Moore Voting Algorithm:
A majority voting algorithm that takes O(n) time and O(1) space to identify the majority elements that appear greater than n/m in an array of length n.

Keynote:
The algorithm only ensures that the appearance of each of the selected majority elements is greater than the appearance of minor elements combined.

Algorithm:

- Initialize m candidates and their countings as (None, 0) pairs
- Iterate over the array:
    - If the current element is equal to one of the potential candidates, the count for that candidate is increased while leaving the count of the other candidate as it is.
    - If the counter reaches zero, the candidate associated with that counter will be replaced with the next element if the next element is not equal to the other candidate as well.
    - All counters are decremented only when the current element is different from all candidates.
- Iterate over the array to verify if the selected majority elements appear greater than n/m times

```
class Solution:
    def majorityElement(self, nums: List[int]) -> List[int]:
        if not nums:
            return []

        count1, count2, candidate1, candidate2 = 0, 0, None, None
        for num in nums:
            if candidate1 == num:
                count1 += 1
            elif candidate2 == num:
                count2 += 1
            elif count1 == 0:
                candidate1 = num
                count1 += 1
            elif count2 == 0:
                candidate2 = num
                count2 += 1
            else:
                count1 -= 1
                count2 -= 1
        result = []
        for c in [candidate1, candidate2]:
            if nums.count(c) > len(nums)//3:
                result.append(c)
        return result
```

11.7. Reservoir Sampling

Reservoir Sampling:
In order to do random sampling over a population of unknown size with constant space, the answer is reservoir sampling. The reservoir sampling algorithm is intended to sample k elements from a population of unknown size, ensuring that each element has an equal probability to be chosen.

We summarize the main idea of the algorithm as follows:
Initially, we fill up an array of reservoir R[] with the heading elements from the pool of samples S[]. At the end of the algorithm, the reservoir will contain the final elements we sample from the pool.
We then iterate through the rest of the elements in the pool. For each element, we need to decide if we want to include it in the reservoir or not. If so, we will replace an existing element in the reservoir with the current element.

```
# S has items to sample, R will contain the result
def ReservoirSample(S[1..n], R[1..k])
  # fill the reservoir array
  for i := 1 to k
      R[i] := S[i]

  # replace elements with gradually decreasing probability
  for i := k+1 to n
    # randomInteger(a, b) generates a uniform integer
    #   from the inclusive range {a, ..., b} *)
    j := randomInteger(1, i)
    if j <= k
        R[j] := S[i]
```

Given the above algorithm, it is guaranteed that at any moment, for each element scanned so far, it has an equal chance to be selected into the reservoir.

If k happens to be one, which means to draw one element from a varied length list, we have a tutorial.

```
int scope = 1, chosenValue = 0;
ListNode curr = this.head;
while (curr != null) {
    // decide whether to include the element in reservoir
    if (Math.random() < 1.0 / scope)
```

```
        chosenValue = curr.val;
    // move on to the next node
    scope += 1;
    curr = curr.next;
}
return chosenValue;
```

11.8. Bitwise Operation

To retrieve the right-most bit in an integer n: n & 1
To retrieve the left bits except the right-most one in an integer n: n >> 1
To flips the least-significant 1-bit in n to 0:   n &= (n - 1)
XOR operation: a^0 = a; a^a = 0; a^b^a = a^a^b = 0^b = b
XOR is module 2 addition: a^b = (a+b)%2

Tricks:

```
# Check if a number is even or odd:

def is_even(n):
    return (n & 1) == 0

def is_odd(n):
    return (n & 1) == 1

# Swap two numbers without a temporary variable:

def swap(a, b):
    a = a ^ b
    b = a ^ b
    a = a ^ b
    return a, b

# Clear the lowest set bit (least significant bit with value one):

def clear_lowest_set_bit(n):
    return n & (n - 1)

# Isolate the lowest set bit (least significant bit with value one):

def isolate_lowest_set_bit(n):
    return n & -n
```

```
n = 12  # 1100 in binary
print(clear_lowest_set_bit(n))  # Output: 8 (1000 in binary)

# Count the number of set bits (Hamming weight):

def count_set_bits(n):
    count = 0
    while n:
        n &= (n - 1)
        count += 1
    return count

# Check if a number is a power of two:

def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0

# Find the next power of two greater than or equal to n:

def next_power_of_two(n):
    if n == 0:
        return 1
    n -= 1
    n |= n >> 1
    n |= n >> 2
    n |= n >> 4
    n |= n >> 8
    n |= n >> 16
    n += 1
    return n

# Reverse the bits of a number

def reverse_bits(n):
    result = 0
    for _ in range(32):  # Assuming 32-bit integer
        result = (result << 1) | (n & 1)
        n >>= 1
    return result

# Find the only non-repeating element in an array where every element
```

```python
repeats twice:

def find_single_element(arr):
    result = 0
    for num in arr:
        result ^= num
    return result

# Get the binary representation of a number as a string:

def binary_representation(n):
    return bin(n)[2:] if n >= 0 else "-" + bin(n)[3:]

# Find the highest set bit:

def highest_set_bit(n):
    if n == 0:
        return 0
    msb = 0
    while n > 1:
        n >>= 1
        msb += 1
    return 1 << msb

# Find the lowest unset bit:

def lowest_unset_bit(n):
    return ~n & (n + 1)

# Toggle all bits:

def toggle_bits(n):
    num_bits = n.bit_length()
    return n ^ ((1 << num_bits) - 1)

n = 18  # Binary: 10010
print(toggle_bits(n))  # Output: 13 (Binary: 01101)

# Multiply by 7 using bitwise operations:

def multiply_by_seven(n):
    return (n << 3) - n
```

```
# Divide by 2 using bitwise operations (floor division):
def divide_by_two(n):
    return n >> 1

# Detect if two integers have opposite signs:

def have_opposite_signs(x, y):
    return (x ^ y) < 0

# Add two numbers without using arithmetic operators:

def add(a, b):
    while b != 0:
        carry = a & b
        a = a ^ b
        b = carry << 1
    return a

# Subtract two numbers without using arithmetic operators:

def subtract(a, b):
    while b != 0:
        borrow = (~a) & b
        a = a ^ b
        b = borrow << 1
    return a

# Count trailing zeros in an integer:

def count_trailing_zeros(n):
    if n == 0:
        return 32  # Assuming 32-bit integer
    count = 0
    while (n & 1) == 0:
        n >>= 1
        count += 1
    return count

# Count leading zeros in an integer:

def count_leading_zeros(n):
    if n == 0:
        return 32  # Assuming 32-bit integer
    count = 0
    while (n >> (31 - count)) & 1 == 0:
        count += 1
    return count

# Find the next higher number with the same number of 1 bits:

def next_higher_with_same_ones(n):
    c = n
    c0 = c1 = 0
    while ((c & 1) == 0) and (c != 0):
        c0 += 1
        c >>= 1
    while (c & 1) == 1:
        c1 += 1
        c >>= 1
    if c0 + c1 == 31 or c0 + c1 == 0:
        return -1
    pos = c0 + c1
    n |= (1 << pos)
    n &= ~((1 << pos) - 1)
    n |= (1 << (c1 - 1)) - 1
    return n

# Find the previous lower number with the same number of 1 bits:

def next_lower_with_same_ones(n):
    temp = n
    c0 = c1 = 0
    while (temp & 1) == 1:
        c1 += 1
        temp >>= 1
    if temp == 0:
        return -1
    while ((temp & 1) == 0) and (temp != 0):
        c0 += 1
        temp >>= 1
    p = c0 + c1
    n &= ((~0) << (p + 1))
    mask = (1 << (c1 + 1)) - 1
    n |= mask << (c0 - 1)
    return n

# Calculate the XOR from 1 to n:
def xor_from_1_to_n(n):
    if n % 4 == 0:
        return n
    elif n % 4 == 1:
        return 1
    elif n % 4 == 2:
        return n + 1
    else:
        return 0

# Determine if a number is a power of four:

def is_power_of_four(n):
    return n > 0 and (n & (n - 1)) == 0 and (n & 0xAAAAAAAA) == 0

# Rotate bits to the left:

def rotate_left(n, d, bit_width=32):
    return (n << d % bit_width) | (n >> (bit_width - d % bit_width))
```

LC 190. Reverse Bits