LeetCode Cheatsheet

- From https://jwl-7.github.io/leetcode-cheatsheet/

Big 0

Time/Space Complexity

Instant: O(1) Logarithmic: O(log n) Linear: O(n) Linearithmic: O(n log n) Quadratic: O(n^2) Cubic: O(n^3) Exponential: O(2^n) Factorial: O(n!)

Data Structure Operations

Data Structure

Time Complexity

Space Complexity

Average

Worst

Worst

Access

Search

Insertion

Deletion

Access

Search

Insertion

Deletion

Array

O(1)

O(n)

O(n)

O(n)

O(1)

O(n)

O(n)

O(n)

O(n)

Stack

O(n)

O(n)

O(1)

O(1)

O(n)

O(n)

O(1)

O(1)

O(n)

Queue

O(n)

O(n)

O(1)

O(1)

O(n)

O(n)

O(1)

O(1)

O(n)

Linked List

O(n)

O(n)

O(1)

O(1)

O(n)

O(n)

O(1)

O(1)

O(n)

Doubly Linked List

O(n)

O(n)

O(1)

O(1)

O(n)

O(n)

O(1)

O(1)

O(n)

Skip List

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(n)

O(n)

O(n)

O(n)

O(n log(n))

Hash Table

O(1)

O(1)

O(1)

O(1)

O(n)

O(n)

O(n)

O(n)

O(n)

Binary Search Tree

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(n)

O(n)

O(n)

O(n)

O(n)

Cartesian Tree

N/A

O(log(n))

O(log(n))

O(log(n))

N/A

O(n)

O(n)

O(n)

O(n)

B-Tree

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(n)

Red-Black Tree

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(n)

Splay Tree

N/A

O(log(n))

O(log(n))

O(log(n))

N/A

O(log(n))

O(log(n))

O(log(n))

O(n)

AVL Tree

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(n)

KD Tree

O(log(n))

O(log(n))

O(log(n))

O(log(n))

O(n)

O(n)

O(n)

O(n)

O(n)

Array Sorting Algorithms

Algorithm

Time Complexity

Space Complexity

Best

Average

Worst

Worst

Quick Sort

O(n log(n))

O(n log(n))

O(n^2)

O(log(n))

Merge Sort

O(n log(n))

O(n log(n))

O(n log(n))

O(n)

Tim Sort

O(n)

O(n log(n))

O(n log(n))

O(n)

Heap Sort

O(n log(n))

O(n log(n))

O(n log(n))

O(1)

Bubble Sort

O(n)

O(n^2)

O(n^2)

O(1)

Insertion Sort

O(n)

O(n^2)

O(n^2)

O(1)

Selection Sort

O(n^2)

O(n^2)

O(n^2)

O(1)

Tree Sort

O(n log(n))

O(n log(n))

O(n^2)

O(n)

Shell Sort

O(n log(n))

O((n log(n))^2)

O((n log(n))^2)

O(1)

Bucket Sort

O(n+k)

O(n+k)

O(n^2)

O(n)

Radix Sort

O(nk)

O(nk)

O(nk)

O(n+k)

Counting Sort

O(n+k)

O(n+k)

O(n+k)

O(k)

Cube Sort

O(n)

O(n log(n))

O(n log(n))

O(n)

Array

two pointers: one input, opposite ends

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

    while left < right:
        # TODO: logic with left and right
        if CONDITION:
            left += 1
        else:
            right -= 1

    return ans
```

two pointers: two inputs, exhaust both

```
def fn(arr1, arr2):
    i = 0
    j = 0
    ans = 0

    while i < len(arr1) and j < len(arr2):
        # TODO: logic
        if CONDITION:
            i += 1
        else:
            j += 1

    while i < len(arr1):
        # TODO: logic
        i += 1

    while j < len(arr2):
        # TODO: logic
        j += 1

    return ans
```

sliding window

```
def fn(arr):
    n = len(arr)
    window = 0
    left = 0
    ans = 0

    for right in range(n):
        # TODO: add arr[right] to window

        while WINDOW_CONDITION_BROKEN:
            # TODO: remove arr[left] from window
            left += 1

        # TODO: update ans

    return ans
```

prefix sum

```
def fn(arr):
    n = len(arr)
```

```
    prefix = [arr[0]]

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

    return prefix
```

efficient string building

```
def fn(strs: list[str]):
    ans = []

    for char in strs:
        ans.append(char)

    return ''.join(ans)
```

Hash Map

find number of subarrays that fit an exact criteria

```
from collections import defaultdict


def fn(arr, k):
    counts = defaultdict(int)
    counts[0] = 1
    ans = curr = 0

    for num in arr:
        # TODO: logic to change curr
        ans += counts[curr - k]
        counts[curr] += 1

    return ans
```

sliding window

```
def fn(arr):
    window = set()
    ans = 0
    left = 0

    for right, ELEMENT in enumerate(arr):
        # TODO: add arr[right] to window

        while WINDOW_CONDITION_BROKEN:
```

```
            # TODO: remove arr[left] from window
            left += 1

        # TODO: update ans

    return ans
```

Linked List

fast and slow pointer

```
def fn(head):
    slow = head
    fast = head
    ans = 0

    while fast and fast.next:
        # TODO: logic
        slow = slow.next
        fast = fast.next.next

    return ans
```

reverse linked list

```
def fn(head):
    prev = None
    curr = head

    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt

    return prev
```

Stack

monotonic increasing stack

```
def fn(arr):
    stack = []
    ans = 0

    for num in arr:
        while stack and stack[-1] > num:
```

```
            # TODO: logic
            stack.pop()
        stack.append(num)

    return ans
```

monotonic decreasing stack

```
def fn(arr):
    stack = []
    ans = 0

    for num in arr:
        while stack and stack[-1] < num:
            # TODO: logic
            stack.pop()
        stack.append(num)

    return ans
```

Binary Tree

DFS (recursive)

```
def dfs(root):
    if not root:
        return

    ans = 0

    # TODO: logic
    dfs(root.left)
    dfs(root.right)

    return ans
```

DFS (iterative)

```
def dfs(root):
    stack = [root]
    ans = 0

    while stack:
        node = stack.pop()
        # TODO: logic
        if node.left:
            stack.append(node.left)
```

```
        if node.right:
            stack.append(node.right)

    return ans
```

BFS

```
from collections import deque


def fn(root):
    que = deque([root])
    ans = 0

    while que:
        current_length = len(que)
        # TODO: logic for current level
        for _ in range(current_length):
            node = que.popleft()
            # TODO: logic
            if node.left:
                que.append(node.left)
            if node.right:
                que.append(node.right)

    return ans
```

Graph

DFS (recursive)

```
def fn(graph):
    def dfs(node):
        ans = 0
        # TODO: logic
        for neighbor in graph[node]:
            if neighbor not in seen:
                seen.add(neighbor)
                ans += dfs(neighbor)

        return ans

    seen = {START_NODE}

    return dfs(START_NODE)
```

DFS (iterative)

```python
def fn(graph):
    stack = [START_NODE]
    seen = {START_NODE}
    ans = 0

    while stack:
        node = stack.pop()
        # TODO: logic
        for neighbor in graph[node]:
            if neighbor not in seen:
                seen.add(neighbor)
                stack.append(neighbor)

    return ans
```

BFS

```python
from collections import deque


def fn(graph):
    que = deque([START_NODE])
    seen = {START_NODE}
    ans = 0

    while que:
        node = que.popleft()
        # TODO: logic
        for neighbor in graph[node]:
            if neighbor not in seen:
                seen.add(neighbor)
                que.append(neighbor)

    return ans
```

Dijkstra (shortest path)

```python
from heapq import heappop, heappush


def dijkstras(graph: list[list[tuple[int, int]]], source: int) -> list[int]:
    n = len(graph)
    distances = [float('inf')] * n
    distances[source] = 0
    heap = [(0, source)]
```

```
    while heap:
        curr_dist, node = heappop(heap)

        if curr_dist > distances[node]:
            continue

        for neighbor, weight in graph[node]:
            dist = curr_dist + weight

            if dist < distances[neighbor]:
                distances[neighbor] = dist
                heappush(heap, (dist, neighbor))

    return distances
```

Bellman-Ford (shortest path)

```
def bellman_ford(n: int, edges: list[tuple[int, int, int]], source: int) -> list[int]:
    distances = [float('inf')] * n
    distances[source] = 0

    for _ in range(n - 1):
        for u, v, w in edges:
            if distances[u] != float('inf') and distances[u] + w < distances[v]:
                distances[v] = distances[u] + w

    for u, v, w in edges:
        if distances[u] != float('inf') and distances[u] + w < distances[v]:
            return []

    return distances
```

Kahn (topological sort)

```
from collections import defaultdict, deque


def kahn_topological_sort(graph: dict[int, list[int]]) -> list[int]:
    result = []
    indegree = defaultdict(int)

    for vertices in graph.values():
        for v in vertices:
            indegree[v] += 1
```

```
    que = deque([node for node in graph if not indegree[node]])

    while que:
        node = que.popleft()
        result.append(node)

        for neighbor in graph[node]:
            indegree[neighbor] -= 1

            if not indegree[neighbor]:
                que.append(neighbor)

    return result if len(result) == len(graph) else []
```

Kruskal (mst)

```
def kruskal_mst(n: int, edges: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    mst = []
    uf = UnionFind(n)
    edges.sort()

    for w, u, v in edges:
        if not uf.connected(u, v):
            uf.union(u, v)
            mst.append((w, u, v))

    return mst
```

Prim (mst)

```
from heapq import heappop


def prim_mst(n: int, edges: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    mst = []
    uf = UnionFind(n)
    edges.sort()

    while edges:
        w, u, v = heappop(edges)

        if not uf.connected(u, v):
            uf.union(u, v)
            mst.append((w, u, v))

    return mst
```

Floyd-Warshall (shortest path)

```python
def floyd_warshall(n: int, edges: list[tuple[int, int, int]]) -> list[list[int]]:
    V = len(graph)
    dist = [row[:] for row in graph]

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    for i in range(V):
        if dist[i][i] < 0:
            return None # Or raise an exception

    return dist
```

Heap

find top k elements

```python
from heapq import heappop, heappush


def fn(arr, k):
    heap = []

    for num in arr:
        # TODO: logic to push onto heap according to problem's criteria
        heappush(heap, (CRITERIA, num))

        if len(heap) > k:
            heappop(heap)

    return [num for num in heap]
```

Binary Search

binary search

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

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

```
        if arr[mid] == target:
            # TODO: logic
            return
        if arr[mid] > target:
            right = mid - 1
        else:
            left = mid + 1

    return left
```

duplicate elements, left-most insertion point

```
def fn(arr, target):
    left = 0
    right = len(arr)

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

        if arr[mid] >= target:
            right = mid
        else:
            left = mid + 1

    return left
```

duplicate elements, right-most insertion point

```
def fn(arr, target):
    left = 0
    right = len(arr)

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

        if arr[mid] > target:
            right = mid
        else:
            left = mid + 1

    return left
```

greedy (minimum)

```
def fn(arr):
    def check(x):
```

```
        return BOOLEAN

    left = MINIMUM_POSSIBLE_ANSWER
    right = MAXIMUM_POSSIBLE_ANSWER

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

        if check(mid):
            right = mid - 1
        else:
            left = mid + 1

    return left
```

greedy (maximum)

```
def fn(arr):
    def check(x):
        return BOOLEAN

    left = MINIMUM_POSSIBLE_ANSWER
    right = MAXIMUM_POSSIBLE_ANSWER

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

        if check(mid):
            left = mid + 1
        else:
            right = mid - 1

    return right
```

Backtracking

backtracking

```
def backtrack(curr, OTHER_ARGUMENTS...):
    if (BASE_CASE):
        # TODO: modify answer
        return

    ans = 0

    for (ITERATE_OVER_INPUT):
        # TODO: modify current state
```

```
        ans += backtrack(curr, OTHER_ARGUMENTS...)
        # TODO: undo modification of current state

    return ans
```

Dynamic Programming

DP top-down

```
def fn(arr):
    @cache
    def dp(STATE):
        if BASE_CASE:
            return 0
        return RECURRENCE_RELATION(STATE)

    return dp(STATE_FOR_WHOLE_INPUT)
```

DP bottom-up

```
def fn(arr):
    if BASE_CASE:
        return 0

    dp = [BASE_CASE] * (STATE_FOR_WHOLE_INPUT + 1)

    for STATE in range(SMALLEST_SUBPROBLEM, STATE_FOR_WHOLE_INPUT + 1):
        if BASE_CASE:
            dp[STATE] = BASE_CASE
        else:
            dp[STATE] = RECURRENCE_RELATION(STATE)

    return dp[STATE_FOR_WHOLE_INPUT]
```

Kadane (max-sum subarray)

```
def kadane(arr: list[int]) -> int:
    curr_sub = max_sub = arr[0]

    for num in arr[1:]:
        curr_sub = max(curr_sub + num, num)
        max_sub = max(max_sub, curr_sub)

    return max_sub
```

Bit Manipulation

test kth-bit

```
def test_kth_bit(num: int, k: int) -> bool:
    return num & (1 << k) != 0
```

set kth bit

```
def set_kth_bit(num: int, k: int) -> int:
    return num | (1 << k)
```

clear kth bit

```
def clear_kth_bit(num: int, k: int) -> int:
    return num & ~(1 << k)
```

toggle kth bit

```
def toggle_kth_bit(num: int, k: int) -> int:
    return num ^ (1 << k)
```

get rightmost bit

```
def get_rightmost_set_bit(num: int) -> int:
    return num & -num
```

count set bits

```
def count_set_bits(num: int) -> int:
    return bin(num).count('1')
```

multiply by 2^k

```
def multiply_by_power_of_two(num: int, k: int) -> int:
    return num << k
```

divide by 2^k

```
def divide_by_power_of_two(num: int, k: int) -> int:
    return num >> k
```

check if number is power of 2

```
def is_power_of_two(num: int) -> bool:
    return (num & (num - 1)) == 0
```

swap two variables

```python
def swap_variables(num1: int, num2: int) -> tuple:
    num1 ^= num2
    num2 ^= num1
    num1 ^= num2
    return num1, num2
```

Matrix

create / copy

```python
def fn(matrix: list[list[int]]):
    r = len(matrix)
    c = len(matrix[0])

    create_matrix = [[0 for _ in range(c)] for _ in range(r)]
    copy_matrix = [row[:] for row in matrix]
```

diagonals / anti-diagonals

```python
def fn(matrix: list[list[int]]):
    r = len(matrix)
    c = len(matrix[0])

    main_diagonal = [matrix[i][i] for i in range(min(r, c))]
    anti_diagonal = [matrix[i][~i] for i in range(min(r, c))]
```

rotate / transpose

```python
def fn(matrix: list[list[int]]):
    r = len(matrix)
    c = len(matrix[0])

    transpose_tuple = zip(*matrix)
    transpose = [list(row) for row in transpose_tuple]
    rotate_left = transpose[::-1]
    rotate_right = [row[::-1] for row in transpose]
```

Data Structures

array

```python
from typing import Any


class Array:
    def __init__(self, size: int) -> None:
```

```
        self.size = size
        self.data = [None] * size

    def __getitem__(self, index: int) -> Any:
        return self.data[index]

    def __setitem__(self, index: int, value: Any) -> None:
        self.data[index] = value

    def __len__(self) -> int:
        return len(self.data)

    def __repr__(self) -> str:
        return repr(self.data)
```

hash map

```
from typing import Any


class HashMap:
    def __init__(self) -> None:
        self.size = 100000
        self.bucket = [None] * self.size

    def _hash(self, key: int) -> int:
        return hash(key) % self.size

    def __setitem__(self, key: int, value: Any) -> None:
        self.bucket[self._hash(key)] = value

    def __getitem__(self, key: int) -> Any:
        return self.bucket[self._hash(key)]

    def __delitem__(self, key: int) -> None:
        self.bucket[self._hash(key)] = None
```

linked list

```
from typing import Any


class ListNode:
    def __init__(self, data: Any) -> None:
        self.data = data
        self.next = None
```

```
    def __repr__(self) -> str:
        return f'[{self.data}]'


class LinkedList:
    def __init__(self) -> None:
        self.head = None

    def append(self, data: Any) -> None:
        if not self.head:
            self.head = ListNode(data)
            return

        curr = self.head

        while curr.next:
            curr = curr.next

        curr.next = ListNode(data)

    def delete(self, data: Any) -> None:
        if not self.head:
            return

        if self.head.data == data:
            self.head = self.head.next
            return

        prev = None
        curr = self.head

        while curr:
            if curr.data == data:
                prev.next = curr.next
                return

            prev = curr
            curr = curr.next

    def reverse(self) -> None:
        prev = None
        curr = self.head

        while curr:
            nxt = curr.next
```

```
            curr.next = prev
            prev = curr
            curr = nxt

        self.head = prev

    def __repr__(self) -> str:
        if not self.head:
            return 'None'

        nodes = []
        node = self.head

        while node:
            nodes.append(repr(node))
            node = node.next

        return ' -> '.join(nodes) + ' -> None'
```

doubly linked list

```
from typing import Any


class ListNode:
    def __init__(self, data: Any) -> None:
        self.data = data
        self.prev = None
        self.next = None

    def __repr__(self) -> str:
        return f'[{self.data}]'


class DoublyLinkedList:
    def __init__(self) -> None:
        self.head = None

    def append(self, data: Any) -> None:
        if not self.head:
            self.head = ListNode(data)
            return

        curr = self.head

        while curr.next:
```

```
            curr = curr.next

        new_node = ListNode(data)
        curr.next = new_node
        new_node.prev = curr

    def delete(self, data: Any) -> None:
        if not self.head:
            return

        if self.head.data == data:
            self.head = self.head.next
            if self.head:
                self.head.prev = None
            return

        curr = self.head
        while curr:
            if curr.data == data:
                prev_node = curr.prev
                prev_node.next = curr.next
                if curr.next:
                    curr.next.prev = prev_node
                return
            curr = curr.next

    def reverse(self) -> None:
        curr = self.head
        prev = None

        while curr:
            nxt = curr.next
            curr.next = prev
            curr.prev = nxt
            prev = curr
            curr = nxt

        self.head = prev

    def __repr__(self) -> str:
        if not self.head:
            return 'None'

        nodes = []
        curr = self.head
```

```
        while curr:
            nodes.append(repr(curr))
            curr = curr.next

        return ' <-> '.join(nodes) + ' <-> None'
```

binary tree

```
from typing import Any


class TreeNode:
    def __init__(self, data: Any) -> None:
        self.data = data
        self.left = None
        self.right = None


class BinaryTree:
    def __init__(self) -> None:
        self.root = None

    def insert(self, data: Any) -> None:
        if not self.root:
            self.root = TreeNode(data)
        else:
            self.insert_node(self.root, data)

    def insert_node(self, node: TreeNode | None, data: Any) -> TreeNode:
        if not node:
            return TreeNode(data)

        if not node.left:
            node.left = TreeNode(data)
        elif not node.right:
            node.right = TreeNode(data)
        else:
            node.left = self.insert_node(node.left, data)

        return node

    def __repr__(self) -> str:
        return 'Empty tree' if not self.root else self._print_tree(self.root, '', True)

    def _print_tree(self, node: TreeNode | None, prefix: str, is_left: bool) -> str:
        if node is None:
```

```
            return ''

        result = ''
        result += self._print_tree(node.right, prefix + ('    ' if is_left else '    '), Fal
        result += prefix + ('   ' if is_left else '   ') + str(node.data) + '\n'
        result += self._print_tree(node.left, prefix + ('    ' if is_left else '    '), True

        return result
```

binary search tree

```
from typing import Any


class TreeNode:
    def __init__(self, data: Any) -> None:
        self.data = data
        self.left = None
        self.right = None


class BinarySearchTree:
    def __init__(self) -> None:
        self.root = None

    def insert(self, data: Any) -> None:
        if not self.root:
            self.root = TreeNode(data)
        else:
            self.insert_node(self.root, data)

    def insert_node(self, node: TreeNode | None, data: Any) -> None:
        if data < node.data:
            if not node.left:
                node.left = TreeNode(data)
            else:
                self.insert_node(node.left, data)
        else:
            if not node.right:
                node.right = TreeNode(data)
            else:
                self.insert_node(node.right, data)

    def __repr__(self) -> str:
        return 'Empty tree' if not self.root else self._print_tree(self.root, '', True)
```

```
    def _print_tree(self, node: TreeNode | None, prefix: str, is_left: bool) -> str:
        if node is None:
            return ''

        result = ''
        result += self._print_tree(node.right, prefix + ('    ' if is_left else '    '), Fal
        result += prefix + ('   ' if is_left else '   ') + str(node.data) + '\n'
        result += self._print_tree(node.left, prefix + ('    ' if is_left else '    '), True

        return result
```

graph

```
class Graph:
    def __init__(self) -> None:
        self.graph = {}

    def add_vertex(self, vertex: str) -> None:
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, a: str, b: str) -> None:
        self.add_vertex(a)
        self.add_vertex(b)
        self.graph[a].append(b)
        self.graph[b].append(a)

    def get_neighbors(self, vertex: str) -> list[str]:
        return self.graph.get(vertex, [])

    def __repr__(self) -> str:
        output = ''

        for vertex, neighbors in self.graph.items():
            output += f'{vertex} - {' - '.join(neighbors)}\n'

        return output
```

union find

```
class UnionFind:
    def __init__(self, n: int) -> None:
        self.root = list(range(n))

    def find(self, a: int) -> int:
        return a if a == self.root[a] else self.find(self.root[a])
```

```
    def union(self, a: int, b: int) -> None:
        self.root[self.find(a)] = self.find(b)

    def connected(self, a: int, b: int) -> bool:
        return self.find(a) == self.find(b)

    def __repr__(self) -> str:
        n = len(self.root)
        lines = []
        components = {}

        for i in range(n):
            root = self.find(i)

            if root not in components:
                components[root] = []

            components[root].append(i)

        for component in components.values():
            lines.append(' - '.join(f'({node})' for node in component))

        return '\n'.join(lines)
```

union find optimized

```
class UnionFind:
    def __init__(self, n: int) -> None:
        self.root = list(range(n))
        self.rank = [1] * n

    def find(self, a: int) -> int:
        return a if a == self.root[a] else self.find(self.root[a])

    def union(self, a: int, b: int) -> None:
        root_a = self.find(a)
        root_b = self.find(b)

        if root_a != root_b:
            if self.rank[root_a] < self.rank[root_b]:
                self.root[root_a] = root_b
            elif self.rank[root_a] > self.rank[root_b]:
                self.root[root_b] = root_a
            else:
                self.root[root_b] = root_a
```

```
                self.rank[root_a] += 1

    def connected(self, a: int, b: int) -> bool:
        return self.find(a) == self.find(b)

    def __repr__(self) -> str:
        n = len(self.root)
        lines = []
        components = {}

        for i in range(n):
            root = self.find(i)

            if root not in components:
                components[root] = []

            components[root].append(i)

        for component in components.values():
            lines.append(' - '.join(f'({node})' for node in component))

        return '\n'.join(lines)
```

trie

```
class TrieNode:
    def __init__(self) -> None:
        self.children = {}
        self.is_word = False


class Trie:
    def __init__(self) -> None:
        self.root = TrieNode()

    def build(self, words: list[str]) -> None:
        for word in words:
            self.insert(word)

    def insert(self, word: str) -> None:
        node = self.root

        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
```

```
        node.is_word = True

    def search(self, word: str) -> bool:
        node = self.root

        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]

        return node.is_word

    def starts_with(self, prefix: str) -> bool:
        node = self.root

        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]

        return True

    def collect_words(self, node: TrieNode, prefix: str) -> list[str]:
        words = []

        if node.is_word:
            words.append(prefix)

        for char, child_node in node.children.items():
            words.extend(self.collect_words(child_node, prefix + char))

        return words

    def __repr__(self) -> str:
        return 'Trie:\n' + self._print_trie(self.root)

    def _print_trie(self, node: TrieNode | None, level: int = 0, prefix: str = '') -> str:
        output = ''
        prefix_str = '    ' * level + prefix

        if not node:
            return output

        if node.is_word:
            output += prefix_str + '   ' + '(*)' + '\n'
```

```
        for i, (char, child_node) in enumerate(node.children.items()):
            is_last = i == len(node.children) - 1
            marker = '└── ' if is_last else '├── '
            output += prefix_str + marker + char + '\n'
            output += self._print_trie(child_node, level + 1, '    ' if not is_last else '│   ')

        return output
```

Min Heap

```
class MinHeap:
    def __init__(self):
        # The heap is stored internally as a list
        self.heap = []

    def _get_parent_index(self, index):
        # Parent of index k is at (k-1)//2
        return (index - 1) // 2

    def _get_left_child_index(self, index):
        # Left child of index k is at 2*k + 1
        return 2 * index + 1

    def _get_right_child_index(self, index):
        # Right child of index k is at 2*k + 2
        return 2 * index + 2

    def _swap(self, i, j):
        # Helper function to swap two elements in the list
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def _heapify_up(self, index):
        """
        Moves the element up the tree until the min-heap property is satisfied.
        Used after insertion.
        """
        parent_index = self._get_parent_index(index)
        # While we are not at the root and the current element is smaller than its parent
        while index > 0 and self.heap[index] < self.heap[parent_index]:
            self._swap(index, parent_index)
            index = parent_index
            parent_index = self._get_parent_index(index)

    def _heapify_down(self, index):
        """
```

```
        Moves the element down the tree until the min-heap property is satisfied.
        Used after removal of the root.
        """
        min_index = index
        while True:
            left_child_index = self._get_left_child_index(index)
            right_child_index = self._get_right_child_index(index)
            heap_size = len(self.heap)

            # Check left child
            if (left_child_index < heap_size and
                self.heap[left_child_index] < self.heap[min_index]):
                min_index = left_child_index

            # Check right child
            if (right_child_index < heap_size and
                self.heap[right_child_index] < self.heap[min_index]):
                min_index = right_child_index

            # If the current node is the smallest among its children/itself, we stop
            if min_index == index:
                break

            # Otherwise, swap and continue heapifying down
            self._swap(index, min_index)
            index = min_index

    def insert(self, value):
        """Adds a new element to the heap."""
        self.heap.append(value)
        # Maintain heap property by moving the new element up
        self._heapify_up(len(self.heap) - 1)

    def extract_min(self):
        """Removes and returns the smallest element (root) from the heap."""
        if self.is_empty():
            return None
        if len(self.heap) == 1:
            return self.heap.pop()

        # Swap the root with the last element
        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        # Maintain heap property by moving the new root down
        self._heapify_down(0)
        return root
```

```
    def peek(self):
        """Returns the smallest element without removing it."""
        if self.is_empty():
            return None
        return self.heap[0]

    def is_empty(self):
        """Checks if the heap is empty."""
        return len(self.heap) == 0

    def size(self):
        """Returns the number of elements in the heap."""
        return len(self.heap)

    def __str__(self):
        return str(self.heap)
```

Sorting Algorithms

bubble sort

- **Should never used**

```
def bubble_sort(arr: list) -> None:
    n = len(arr)

    for i in range(n):
        swapped = False

        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        if not swapped:
            break
```

selection sort

```
def selection_sort(arr: list) -> None:
    n = len(arr)

    for i in range(n):
        min_i = i

        for j in range(i + 1, n):
```

```
            if arr[j] < arr[min_i]:
                min_i = j

        if min_i != i:
            arr[i], arr[min_i] = arr[min_i], arr[i]
```

insertion sort

```
def insertion_sort(arr: list) -> None:
    n = len(arr)

    for i in range(1, n):
        key = arr[i]

        while i > 0 and key < arr[i - 1]:
            arr[i] = arr[i - 1]
            i -= 1

        arr[i] = key
```

shell sort

```
def shell_sort(arr: list) -> None:
    n = len(arr)
    gaps = [701, 301, 132, 57, 23, 10, 4, 1]

    for gap in gaps:
        for i in range(gap, n):
            tmp = arr[i]
            j = i

            while j >= gap and arr[j - gap] > tmp:
                arr[j] = arr[j - gap]
                j -= gap

            if j != i:
                arr[j] = tmp
```

merge sort

- Should used for LeetCode because it runs faster than quick sort

```
def merge_sort(arr: list) -> list:
    n = len(arr)

    if n <= 1:
        return arr
```

```
    mid = n // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left: list, right: list) -> list:
    output = []

    while left and right:
        min_num = left.pop(0) if left[0] <= right[0] else right.pop(0)
        output.append(min_num)

    output.extend(left)
    output.extend(right)

    return output
```

quick sort

```
def quick_sort(arr: list) -> list:
    n = len(arr)

    if n <= 1:
        return arr

    pivot = arr[n // 2]
    left = [x for x in arr if x < pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + [pivot] + quick_sort(right)
```

timsort

```
def tim_sort(arr: list) -> list:
    n = len(arr)
    min_run = 32

    for start in range(0, n, min_run):
        end = min(start + min_run - 1, n - 1)
        insertion_sort(arr, start, end)

    size = min_run
```

```
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), (n - 1))
            arr[left : right + 1] = merge(arr[left : mid + 1], arr[mid + 1 : right + 1])
        size *= 2

    return arr

def insertion_sort(arr: list, left: int, right: int) -> None:
    for i in range(left + 1, right + 1):
        key = arr[i]

        while i > 0 and key < arr[i - 1]:
            arr[i] = arr[i - 1]
            i -= 1

        arr[i] = key

def merge_sort(arr: list) -> list:
    n = len(arr)

    if n <= 1:
        return arr

    mid = n // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left: list, right: list) -> list:
    output = []

    while left and right:
        min_num = left.pop(0) if left[0] <= right[0] else right.pop(0)
        output.append(min_num)

    output.extend(left)
    output.extend(right)

    return output
```

heap sort

```python
def heap_sort(arr: list) -> list:
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

    return arr

def heapify(arr: list, n: int, i: int) -> None:
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

counting sort

```python
def counting_sort(arr: list) -> list:
    max_num = max(arr)
    min_num = min(arr)
    count_range = max_num - min_num + 1
    count = [0] * count_range
    output = [0] * len(arr)

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

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

    for num in arr[::-1]:
        output[count[num - min_num] - 1] = num
        count[num - min_num] -= 1

    return output
```

bucket sort

```
def bucket_sort(arr: list) -> list:
    num_buckets = 10
    min_num = min(arr)
    max_num = max(arr)
    bucket_size = (max_num - min_num) / num_buckets
    buckets = [[] for _ in range(num_buckets)]

    for num in arr:
        index = min(int((num - min_num) / bucket_size), num_buckets - 1)
        buckets[index].append(num)

    return [num for bucket in buckets for num in sorted(bucket)]
```

radix sort

```
def radix_sort(arr: list) -> None:
    max_val = max(arr)
    exp = 1

    while max_val // exp > 0:
        counting_sort(arr, exp)
        exp *= 10

def counting_sort(arr: list, exp: int) -> None:
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        idx = arr[i] // exp
        count[idx % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1

    while i >= 0:
        idx = arr[i] // exp
        output[count[idx % 10] - 1] = arr[i]
        count[idx % 10] -= 1
        i -= 1

    for i in range(n):
```

```
        arr[i] = output[i]
```

cubesort

```
def cube_sort(arr: list, processors: int) -> None:
    n = len(arr)
    subarrays = []
    subarray_size = n // processors

    for i in range(processors):
        subarray = arr[i * subarray_size : (i + 1) * subarray_size]
        subarrays.append(subarray)

    for subarray in subarrays:
        subarray.sort()

    for dimension in range(processors.bit_length() - 1):
        for i in range(processors):
            partner = i ^ (1 << dimension)
            if i < partner:
                merged = subarrays[i] + subarrays[partner]
            else:
                merged = subarrays[partner] + subarrays[i]
            merged.sort()
            subarrays[i] = merged[:subarray_size]
            subarrays[partner] = merged[subarray_size:]

    arr[:] = [num for subarray in subarrays for num in subarray]
```

bogo sort

```
import random


def bogo_sort(arr: list) -> None:
    target = sorted(arr)

    while arr != target:
        random.shuffle(arr)
```

pancake sort

```
def pancake_sort(arr: list) -> None:
    n = len(arr)

    for size in reversed(range(2, n + 1)):
        max_idx = find_max_index(arr, size)
```

```
        if max_idx != size - 1:
            flip(arr, max_idx)
            flip(arr, size - 1)

def flip(arr: list, i: int) -> None:
    left = 0

    while left < i:
        arr[left], arr[i] = arr[i], arr[left]
        left += 1
        i -= 1

def find_max_index(arr: list, n: int) -> int:
    max_idx = 0

    for i in range(n):
        if arr[i] > arr[max_idx]:
            max_idx = i

    return max_idx
```

sleep sort

```
def pancake_sort(arr: list) -> None:
    n = len(arr)

    for size in reversed(range(2, n + 1)):
        max_idx = find_max_index(arr, size)

        if max_idx != size - 1:
            flip(arr, max_idx)
            flip(arr, size - 1)

def flip(arr: list, i: int) -> None:
    left = 0

    while left < i:
        arr[left], arr[i] = arr[i], arr[left]
        left += 1
        i -= 1

def find_max_index(arr: list, n: int) -> int:
    max_idx = 0

    for i in range(n):
```

```
    if arr[i] > arr[max_idx]:
        max_idx = i

return max_idx
```