HashMap, Binary Search, DFS, BFS ⭐⭐⭐⭐⭐
Graph, Stack, Heap, Iterator ⭐⭐⭐⭐
Two Pointers, DP, Tree, Union Find, Trie ⭐⭐⭐

Data Structures

Interval

Merge Intervals

```
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda interval: interval[0])
        result = [intervals[0]]
        for interval in intervals[1:]:
            if interval[0] <= result[-1][1]:
                result[-1][1] = max(result[-1][1], interval[1])
            else:
                result.append(interval)
        return result
```

Insert Interval

```
    def insert(self, intervals: List[List[int]], newInterval: List[int])
-> List[List[int]]:
        result = []
        for i, interval in enumerate(intervals):
            if newInterval[0] > interval[1]:
                result.append(interval)
            elif newInterval[0] <= interval[1] and interval[0] <=
newInterval[1]:
                newInterval = [min(interval[0], newInterval[0]),
max(interval[1], newInterval[1])]
            else:
                result.append(newInterval)
                result.extend(intervals[i:])
                newInterval = None
                break

        if newInterval:
            result.append(newInterval)

        return result
```

HashMap, HashSet

TODO:

Linked List

Reversed Linked List

```
    def reverseList(self, head: ListNode) -> ListNode:
        if head is None:
            return head

        prev, pos = None, head.next
        while head:
            pos = head.next
            head.next = prev
            prev = head
            head = pos
        return prev
```

Middle of Linked List

```
    def middleNode(self, head: ListNode) -> ListNode:
        if head is None:
            return None

        fast, slow = head, head
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next
        return slow
```

HashLinkedList (LRU)

```
class ListNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.hash = {}  # key to node
        self.head = ListNode(0, 0) # dummy head
        self.tail = ListNode(0, 0) # dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def delete_node(self, node):
        # delete from list
        node.prev.next = node.next
        node.next.prev = node.prev

    def add_to_head(self, node):
        # add to head
        node.next = self.head.next
        node.prev = self.head
        self.head.next = node
        node.next.prev = node

    def get(self, key: int) -> int:
        if key not in self.hash:
            return -1
        node = self.hash[key]
        self.delete_node(node)
        self.add_to_head(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key in self.hash:
            node = self.hash[key]
            node.value = value
            self.delete_node(node)
        else:
            node = ListNode(key, value)
            self.hash[key] = node
        self.add_to_head(node)

        # evict
        if len(self.hash) > self.capacity:
            # node to be deleted
            node = self.tail.prev
            del self.hash[node.key]
            self.delete_node(node)
```

TODO: LFU

Stack

TODO:

Queue

TODO:

Deque

TODO:

Monotonic Stack

适用范围 :一维数组上判断前后元素大小关系
栈里保存的是index

```
def largestRectangleArea(self, heights: List[int]) -> int:
    if not heights:
        return 0

    max_area = 0
    stack = []
    heights.append(-1)

    for i in range(len(heights)):
        while stack and heights[stack[-1]] > heights[i]:
            j = stack.pop()
            h = heights[j]
            l = 0 if not stack else stack[-1] + 1
            r = i - 1
            max_area = max(max_area, heights[j] * (r - l + 1))
        stack.append(i)
    return max_area
```

Monotonic Deque

适用范围
sliding window上求一定范围内的值
例子
https://leetcode.com/problems/sliding-window-maximum/submissions/

```
def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
   if nums is None or len(nums) < k:
       return []

   queue = deque([])
   result = []
   for i in range(len(nums)):
       while queue and nums[q[-1]] <= nums[i]:
           queue.pop()
       queue.append(i)
```

```
        if i >= k - 1:
            result.append(nums[queue[0]])

        if i - q[0] >= k - 1:
            queue.popleft()
    return result
```

例子2
https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k

```
def shortestSubarray(self, A: List[int], k: int) -> int:
        if not A:
            return -1

        if max(A) >= k:
            return 1

        presum = [0]
        current_sum = 0
        for num in A:
            current_sum += num
            presum.append(current_sum)

        q = deque([])
        min_length = math.inf
        for i in range(len(presum)):
            while q and presum[i] - presum[q[0]] >= k:
                min_length = min(min_length, i - q[0])
                q.popleft()
            while q and presum[q[-1]] >= presum[i]:
                q.pop()
            q.append(i)
        return min_length if min_length != math.inf else -1
```

Tree

遍历

```
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        if root is None:
            return []

        stack, output = [root, ], []
        while stack:
            root = stack.pop()
            if root is not None:
                output.append(root.val)
                if root.right is not None:
                    stack.append(root.right)
                if root.left is not None:
                    stack.append(root.left)
        return output
```

```
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []

        dummy = TreeNode()
        dummy.right = root
        stack = [dummy]
        result = []

        while stack:
            node = stack.pop()
            result.append(node.val)
            node = node.right
            while node:
                stack.append(node)
                node = node.left
        return result[1:]
```

分治

```
def divide_conquer(root):
    # 递归出口，一般处理 node == null的情况
    # 有些时候需要处理node is leaf
    if root is None:
        return ...
    left_result = divide_conquer(node.left)
    right_result = divide_conquer(node.right)
    return merge left_result and right_result to get merged result
```

变形
Flatten Binary Tree to Linked List

```
    def flatten(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        self.helper(root)

    def helper(self, root):
        # return last node after flatten
        if root is None:
            return None

        if root.left is None and root.right is None:
            return root

        left = self.helper(root.left)
        right = self.helper(root.right)

        if root.left is None:
            return right

        left.right = root.right
        root.right = root.left
        root.left = None
        return right if right else left
```

Binary Search Tree

TODO:

Heap

适用范围

1. 最大值，最小值，中位数
2. 求第k大 O(k * logn)
3. 要求O(logn)时间内进行操作

不适用

1. 查询比某个数大的最小值/小的最大值/最接近值 (TreeMap)
2. 区间查询 (Segment Tree, Binary Indexed Tree)
3. O(n)求第k大 (quick select)

- 堆的应用场景：
  - 找极值（题目中第 k 大，最大等字眼）
  - 堆排序（扫描线等需要排序的情况，和快排用法基本相同）
- 数据流的考点：
  - 什么都考，主要体现“在线”二字
  - 要考虑在线数据结构（如：并查集）
  - 要注意新数据插入后，其他数据的变化能否很快处理（数据流中位数）
- 根据调用频率选择算法或数据结构
  - 频率越高，复杂度应该越低

第k大

```
    def findKthLargest(self, nums: List[int], k: int) -> int:
        heap = []
        for num in nums:
            heappush(heap, num)
            if len(heap) > k:
                heappop(heap)
        return heappop(heap)
```

堆排序 升序排建最大堆

```
def heap_sort(self, A):
        # heap sort
        self.heapify(A)

        for i in range(len(A) - 1, -1, -1):
            A[i], A[0] = A[0], A[i]
            self.sift_down(A, i, 0)

    def heapify(self, A):
        if len(A) <= 1:
            return None

        start = (len(A) - 2) // 2
        for i in range(start, -1, -1):
            self.sift_down(A, len(A), i)

    def sift_down(self, A, n, i):
        if i >= n:
            return None
        max_index = i
        if 2 * i + 1 < n and A[2 * i + 1] > A[max_index]:
            max_index = 2 * i + 1
        if 2 * i + 2 < n and A[2 * i + 2] > A[max_index]:
            max_index = 2 * i + 2
        if max_index == i:
            return None

        A[max_index], A[i] = A[i], A[max_index]
        self.sift_down(A, n, max_index)
```

data stream求中位数

```
class MedianFinder:
    def __init__(self):
        self.h1 = [] # smaller half
        self.h2 = [] # larger half

    def addNum(self, num: int) -> None:
        if len(self.h1) == 0:
            heappush(self.h1, -num)
            return None

        pivot = -self.h1[0]
        if num < pivot:
            heappush(self.h1, -num)
            if len(self.h1) > len(self.h2) + 1:
                pivot = -heappop(self.h1)
                heappush(self.h2, pivot)
        else:
            heappush(self.h2, num)
            if len(self.h2) > len(self.h1):
                pivot = heappop(self.h2)
                heappush(self.h1, -pivot)

    def findMedian(self) -> float:
        if len(self.h1) > len(self.h2):
            return -self.h1[0]
        else:
            return (self.h2[0] - self.h1[0]) / 2
```

HashHeap 带删除的堆

```
class HashHeap:
    def __init__(self, desc=False):
        self.hash = dict()
```

```python
        self.heap = []
        self.desc = desc

    @property
    def size(self):
        return len(self.heap)

    def push(self, item):
        self.heap.append(item)
        self.hash[item] = self.size - 1
        self._sift_up(self.size - 1)

    def pop(self):
        item = self.heap[0]
        self.remove(item)
        return item

    def top(self):
        return self.heap[0]

    def remove(self, item):
        if item not in self.hash:
            return

        index = self.hash[item]
        self._swap(index, self.size - 1)

        del self.hash[item]
        self.heap.pop()

        # in case of the removed item is the last item
        if index < self.size:
            self._sift_up(index)
            self._sift_down(index)

    def _smaller(self, left, right):
        return right < left if self.desc else left < right

    def _sift_up(self, index):
        while index != 0:
            parent = index // 2
            if self._smaller(self.heap[parent], self.heap[index]):
                break
            self._swap(parent, index)
            index = parent
    def _sift_down(self, index):
        if index is None:
            return
        while index * 2 < self.size:
            smallest = index
            left = index * 2
            right = index * 2 + 1

            if self._smaller(self.heap[left], self.heap[smallest]):
                smallest = left

            if right < self.size and self._smaller(self.heap[right],
self.heap[smallest]):
                smallest = right

            if smallest == index:
                break

            self._swap(index, smallest)
            index = smallest

    def _swap(self, i, j):
        elem1 = self.heap[i]
        elem2 = self.heap[j]
        self.heap[i] = elem2
        self.heap[j] = elem1
        self.hash[elem1] = j
        self.hash[elem2] = i
```

Lazy deletion

```
class Heap:
    def __init__(self):
        self.minheap = []
        self.deleted_set = set()
        # add desc for max
        # add key_set() to be able to distinguish

    def push(self, index, val):
        heappush(self.minheap, (val, index))

    def _lazy_deletion(self):
        while self.minheap and self.minheap[0][1] in self.deleted_set:
            heappop(self.minheap)
    def top(self):
        self._lazy_deletion()
        return self.minheap[0]

    def pop(self):
        self._lazy_deletion()
        heappop(self.minheap)

    def delete(self, index):
        self.deleted_set.add(index)

    def is_empty(self):
        return not bool(self.minheap)
```

TreeMap

TODO:

Segment Tree

Segment tree is a very flexible data structure, because it is used to solve numerous range query problems like finding minimum, maximum, sum, **greatest common divisor, least common denominator** in an array in logarithmic time.

```
class SegmentTreeNode:
    def __init__(self, start, end, val, left=None, right=None):
        self.start = start
        self.end = end
        self.val = val
        self.left = left
        self.right = right

class SegmentTree:
    def __init__(self, nums: List[int]):
        self.root = SegmentTree.build_tree(nums, 0, len(nums) - 1)

    @classmethod
    def build_tree(cls, nums, start, end):
        if start == end:
            return SegmentTreeNode(start, end, nums[start])

        mid = (start + end) // 2
        left = SegmentTree.build_tree(nums, start, mid)
        right = SegmentTree.build_tree(nums, mid + 1, end)
        return SegmentTreeNode(start, end, left.val + right.val, left,
right)

    def update(self, i, val):
        SegmentTree._update(self.root, i, val)

    @classmethod
    def _update(cls, root, i, val):
        if root.start == root.end and root.start == i:
            root.val = val
            return None

        mid = (root.start + root.end) // 2
        if i <= mid:
            SegmentTree._update(root.left, i, val)
        else:
            SegmentTree._update(root.right, i, val)
        root.val = root.left.val + root.right.val

    def query(self, i, j):
        return SegmentTree._query(self.root, i, j)

    @classmethod
    def _query(cls, root, i, j):
        if root.start == i and root.end == j:
            return root.val

        mid = (root.start + root.end) // 2
        if i > mid:
            return SegmentTree._query(root.right, i, j)
        elif j <= mid:
            return SegmentTree._query(root.left, i, j)
        return SegmentTree._query(root.left, i, mid) +
SegmentTree._query(root.right, mid + 1, j)
```

数组
TODO:

离散化
TODO:

lazy update
TODO:

Binary Indexed Tree

求区间和
update O(logN)

query O(logN)

```
class BinaryIndexedTree:
    def __init__(self, size):
        self.sums = [0] * (size + 1)

    def update(self, i, delta):
        while i < len(self.sums):
            self.sums[i] += delta
            i += BinaryIndexedTree.lowbit(i)

    def query(self, i):
        result = 0
        while i > 0:
            result += self.sums[i]
            i -= BinaryIndexedTree.lowbit(i)
        return result

    @staticmethod
    def lowbit(i):
        return i & (-i)
```

Trie

适用范围

需要查询包含某个前缀的字符串是否存在

字符矩阵中找字符串
时间复杂度 O(L) L为字符串长度
空间复杂度 O(n * L) n为字符串数量

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
        node.is_word = True

    def find(self, word):
        node = self.root
        for c in word:
            node = node.children.get(c)
            if node is None:
                return None
        return node

    def search(self, word):
        node = self.find(word)
        return node is not None and node.is_word

    def startsWith(self, prefix):
        return self.find(prefix) is not None
```

Union Find

适用范围
查询图的连通性问题
快速合并两个集合的问题

时间复杂度 union `O(1)`, find `O(1)`
空间复杂度 `O(n)`

```
class UnionFind:
    def __init__(self):
        self.parent = {}
        # add rank to further optimize

    def add(self, x):
        if x in self.parent:
            return None
        self.parent[x] = x

    def find(self, x):
        root = x
        while root != self.parent[root]:
            root = self.parent[root]
        while x != root:
            x, self.parent[x] = self.parent[x], root
        return root
```

```python
    def union(self, x, y):
        root_x = self.find(x) # NOT self.parent[x]
        root_y = self.find(y) # NOT self.parent[y]
        if root_x != root_y:
            self.parent[root_x] = root_y
```

Graph

Traversal

```python
class Graph:
    def __init__(self, directed=True, nodes=[], edges=[]):
        self.directed = directed
        self.graph = {}
        for node in nodes:
            self.graph[node] = {}
        for node1, node2, weight in edges:
            self.add_edge(node1, node2, weight)

    def add_edge(self, node1, node2, weight):
        self.graph[node1][node2] = weight
        if self.directed is False:
            self.graph[node2][node1] = weight

    def dfs_recursive(self, start_nodes=[]):
        visited = set()
        result = []
        for start_node in start_nodes:
            if start_node not in visited:
                self.dfs_recursive_helper(start_node, visited, result)
        return result

    def dfs_recursive_helper(self, node, visited, result):
        visited.add(node)
        result.append(node)
        for next_node in self.graph[node]:
            if next_node not in visited:
                self.dfs_recursive_helper(next_node, visited, result)

    def dfs_iterative(self, start_nodes=[]):
        stack = list(start_nodes)
        visited = set(start_nodes)
        result = []

        while stack:
            node = stack.pop()
            result.append(node)
            for next_node in self.graph[node]:
                if next_node not in visited:
                    stack.append(next_node)
                    visited.add(next_node)
        return result

    def dfs_recursive_full(self, start_nodes=[]):
        self.time = 0
        self.start_times = defaultdict(int)
        self.end_times = defaultdict(int)

        result = []
        visited = set()

        for start_node in start_nodes:
            if start_node not in visited:
                self.dfs_recursive_full_helper(start_node, visited,
result)
        return result

    def dfs_recursive_full_helper(self, node, visited, result):
        visited.add(node)
        result.append(node)
        self.start_times[node] = self.time
        self.time += 1

        for next_node in self.graph[node]:
            if next_node not in visited:
                print(f"tree edge from {node} to {next_node}")
                self.dfs_recursive_full_helper(next_node, visited,
result)
            else:
                if self.start_times[node] < self.start_times[next_node]:
                    print(f"forward edge from {node} to {next_node}")
                else:
                    if self.end_times[node] < self.end_times[next_node]:
                        print(f"cross edge from {node} to {next_node}")
                    else:
                        print(f"backward edge from {node} to
{next_node}")
            self.end_times[node] = self.time
            self.time += 1

    def bfs(self, start_nodes):
        queue = deque(start_nodes)
        visited = set(start_nodes)
        result = []

        while queue:
            node = queue.popleft()
            result.append(node)
            for next_node in self.graph[node]:
                if next_node not in visited:
                    queue.append(next_node)
                    visited.add(next_node)
        return result
```

Connectivity

Find if there is a path between u and v (BFS, DFS)
Find Bridge (DFS - 判断每个顶点为根的子数最小能到达的编号)
Is Valid Tree (connected, and number of edges == number of nodes - 1)
Count all possible paths between u and v (backtracking)
Longest Path:
无向图：
有向图：topological sort

Detect Cycle

```
def is_cyclic(self):
        visited = set()
        recursion_stack = set()
        for node in self.graph:
            if node not in visited and self.is_cyclic_helper(node,
visited, recursion_stack):
                return True
        return False

    def is_cyclic_helper(self, node, visited, recursion_stack):
        visited.add(node)
        recursion_stack.add(node)

        for next_node in self.graph[node]:
            if next_node not in visited and
self.is_cyclic_helper(next_node, visited, recursion_stack):
                return True
            elif next_node in recursion_stack:
                return True
        recursion_stack.remove(node)
        return False
```

Topological Sorting

TODO:

Bipartite Graph

无向图 BFS https://leetcode.com/problems/is-graph-bipartite/

```
    def isBipartite(self, edges: List[List[int]]) -> bool:
        node_to_color = {}
        for node in graph:
            if node not in node_to_color:
                if self.is_bipartite_helper(node, graph, node_to_color)
is False:
                    return False
        return True

    def is_bipartite_helper(self, start_node, graph, node_to_color):
        node_to_color[start_node] = 0
        queue = deque([start_node])
        while queue:
            cur_node = queue.popleft()
            for next_node in graph[cur_node]:
                if next_node not in node_to_color:
                    node_to_color[next_node] = 1 -
node_to_color[cur_node]
                    queue.append(next_node)
                elif node_to_color[next_node] ==
node_to_color[cur_node]:
                    return False
        return True
```

无向图 DFS

```
    def isBipartite(self, graph: List[List[int]]) -> bool:
        colors = [0] * len(graph)
        for node in range(len(graph)):
            if colors[node] == 0 and self.dfs(graph, node, colors, 1) is
False:
                return False
        return True

    def dfs(self, graph, node, colors, group_id):
        """
        Returns False if a conflict is found
        node: the node to be visited
        colors: the status of node:
            0: not visited
            1: visited and in the first group
            -1: visited and in the second group
        """

        colors[node] = group_id
        for neighbor in graph[node]:
            if colors[neighbor] == group_id or (colors[neighbor] == 0
and self.dfs(graph, neighbor, colors, -group_id) is False):
                return False
        return True
```

TODO: 有向图

应用4:

Shortest Path

单源最短路径 Dijkstra O(V+ElogV)

```
    def dijkstra(self, start_node, end_node):
        distances = {node: math.inf for node in self.graph}
        distances[start_node] = 0
        visited = set()
        heap = [(0, start_node)]

        while len(visited) < len(self.graph):
            distance, node = heappop(heap)
            if node == end_node:
                return distance
            visited.add(node)
            for next_node in self.graph[node]:
                next_distance = distances[node] +
self.graph[node][next_node]
                if next_node not in visited and distances[next_node] >
next_distance:
                    heappush(heap, (next_distance, next_node))
                    distances[next_node] = next_distance
        return -1
```

TODO: Bellman Ford

如果需要打印路径，则增加一个parent dict

MST

https://leetcode.com/problems/connecting-cities-with-minimum-cost/

Kruskal: 按权值大小排序，如果同一edge两个vertex未连通，则加入结果

```
    def minimumCost(self, n: int, connections: List[List[int]]) -> int:
        connections.sort(key=lambda connection: connection[2])

        uf = UnionFind(n)
        edge_num, edge_sum = 0, 0
        for x, y, weight in connections:
            if uf.union(x, y):
                edge_num += 1
                edge_sum += weight

        return edge_sum if edge_num == n - 1 else -1
```

Prim: O(ElogE) - Python没有HashHeap, 如果有HashHeap可以到O(ElogV)
visited set初始化为1个vertex, 不断加入和visited set中任意顶点距离最近的且还不在visited 中的顶点，每次加入时选择最小权重的那个

```
def minimumCost(self, n: int, connections: List[List[int]]) -> int:
        if n <= 1:
            return 0

        graph = defaultdict(dict)
        for u, v, weight in connections:
            if v in graph[u]:
                graph[u][v] = min(weight, graph[u][v])
            else:
                graph[u][v] = weight
            if u in graph[v]:
                graph[v][u] = min(weight, graph[u][v])
            else:
                graph[v][u] = weight

        visited = set()
        distances = [(0, n)]

        cost_sum = 0
        while len(visited) < n and len(distances) > 0:
            cost, node = heappop(distances)
            if node in visited:
                continue
            cost_sum += cost
            visited.add(node)
            for neighbor in graph[node]:
                heappush(distances, (graph[node][neighbor], neighbor))
        return cost_sum if len(visited) == n else -1
```

Find ALL shortest path
记录parent[i] as a set
无权图 BFS
加权图 SPFA

Iterator

1. flatten data into a list
2. use stack

```
define function iterativeDepthFirstSearch(nestedList):
    result = []

    stack = a new Stack
    push all items in nestedList onto stack, in reverse order

    while stack is not empty:
        nestedInteger = pop top of stack
        if nestedInteger.isInteger():
            append nestedInteger.getInteger() to result
        else:
            list = nestedInteger.getList()
            push all items in list onto stack, in reverse order

    return result
```

3. generator (usually with peeked value)

例子

```
class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):
        self.data = []
        self.index = 0
        self.flatten(nestedList)

    def flatten(self, nested_list):
        for element in nested_list:
            if element.isInteger():
                self.data.append(element.getInteger())
            else:
                self.flatten(element.getList())

    def next(self) -> int:
        val = self.data[self.index]
```

```python
        self.index += 1
        return val

    def hasNext(self) -> bool:
        return self.index < len(self.data)
```

Stack 1:

```python
class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):
        self.stack = []
        for element in reversed(nestedList):
            self.stack.append(element)

    def pop_integer(self):
        while self.stack and self.stack[-1].isInteger() is False:
            temp_list = self.stack.pop()
            for element in reversed(temp_list.getList()):
                self.stack.append(element)

    def next(self) -> int:
        self.pop_integer()
        return self.stack.pop().getInteger()

    def hasNext(self) -> bool:
        self.pop_integer()
        return self.stack
```

Stack 2:

```python
class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):
        self.stack = [[nestedList, 0]]

    def pop_integer(self):
        while self.stack:
            current_list = self.stack[-1][0]
            current_index = self.stack[-1][1]
            if current_index == len(current_list):
                self.stack.pop()
                continue

            if current_list[current_index].isInteger():
                break

            next_list = current_list[current_index].getList()
            self.stack[-1][1] += 1
            self.stack.append([next_list, 0])

    def next(self) -> int:
        self.pop_integer()
        current_list, current_index = self.stack[-1]
        self.stack[-1][1] += 1
        return current_list[current_index].getInteger()

    def hasNext(self) -> bool:
        self.pop_integer()
        return self.stack
```

3. generator + peeked value

```
class NestedIterator:

    def __init__(self, nestedList: [NestedInteger]):
        # Get a generator object from the generator function, passing in
        # nestedList as the parameter.
        self._generator = self._int_generator(nestedList)
        # All values are placed here before being returned.
        self._peeked = None

    # This is the generator function. It can be used to create generator
    # objects.
    def _int_generator(self, nested_list) -> "Generator[int]":
        # This code is the same as Approach 1. It's a recursive DFS.
        for nested in nested_list:
            if nested.isInteger():
                yield nested.getInteger()
            else:
                # We always use "yield from" on recursive generator
calls.
                yield from self._int_generator(nested.getList())
        # Will automatically raise a StopIteration.

    def next(self) -> int:
        # Check there are integers left, and if so, then this will
        # also put one into self._peeked.
        if not self.hasNext(): return None
        # Return the value of self._peeked, also clearing it.
        next_integer, self._peeked = self._peeked, None
        return next_integer

    def hasNext(self) -> bool:
        if self._peeked is not None: return True
        try: # Get another integer out of the generator.
            self._peeked = next(self._generator)
            return True
        except: # The generator is finished so raised StopIteration.
            return False
```

Reservoir Sampling

Algorithm

Two Pointers

Sliding Window

同向双指针使用条件：1. 关键词：sliding window, subarray/substring 2. O(n)

以i为主指针，求最短的时候需要反过来X

```
def shortest_subarray(array, X):
    i, j = 0, 0
    min_length = math.inf

    for i in range(len(array)):
        while j < len(array) and not satisfy X:
            # update
            j += 1
        if satisfy X:
            min_length = min(min_length, j - i)
        # update
    return min_length if min_length != math.inf else -1
```

例子

```
def minimumSize(self, nums, s):
        j = 0
        sum_subarray = 0
        result = math.inf # min length
        for i in range(len(nums)):
            while j < len(nums) and sum_subarray < s:  # not X
                sum_subarray += nums[j]
                j += 1
            if sum_subarray >= s:
                result = min(result, j - i)
```

```
           sum_subarray -= nums[i]
       return result if result != math.inf else -1
```

求最长的X

```
def longest_subarray(array):
    i, j = 0, 0
    max_length = 0

    for i in range(len(array)):
        while j < len(array) and satisfy X:
            # update
            j += 1
        if satisfy X:
            max_length = max(min_length, j - i)
        # update
    return min_length if min_length != math.inf else -1
```

例子

```
def lengthOfLongestSubstringKDistinct(self, s, k):
        if k == 0:
            return 0

        i, j = 0, 0
        counter = collections.defaultdict(int)
        max_length = 0
        for i in range(len(s)):
            while j < len(s) and (len(counter) < k or s[j] in counter):
                counter[s[j]] += 1
                j += 1

            max_length = max(max_length, j - i) # 也可以把这个写在循环里面 max_length = max(max_length, j - i + 1)
            counter[s[i]] -= 1
            if counter[s[i]] == 0:
                del counter[s[i]]
        return max_length
```

以j为主指针也可以

```
   def minimumSize(self, nums, s):
       i, j = 0, 0
       sum_subarray = 0
       min_length = math.inf

       for j in range(len(nums)):
           sum_subarray += nums[j]
           while i <= j and sum_subarray >= s:
               min_length = min(min_length, j - i + 1)
               sum_subarray -= nums[i]
               i += 1
       return min_length if min_length != math.inf else -1
```

```
def lengthOfLongestSubstringKDistinct(self, s, k):
        i, j = 0, 0
        max_length = 0
        counter = collections.defaultdict(int)

        for j in range(len(s)):
            counter[s[j]] += 1
            while i < j and len(counter) > k:
                counter[s[i]] -= 1
                if counter[s[i]] == 0:
                    del counter[s[i]]
                i += 1
            if len(counter) <= k:
                max_length = max(max_length, j - i + 1)
        return max_length
```

Partition

适用范围
原地操作，只可以交换，不能适用额外空间(80%)

```
    def partitionArray(self, nums, k):
        if not nums:
            return 0
        left, right = 0, len(nums) - 1
        while left <= right:
            while left <= right and nums[left] < k:
                left += 1
            while left <= right and nums[right] >= k:
                right -= 1
            if left <= right:
                nums[left], nums[right] = nums[right], nums[left]
                left += 1
                right -= 1
        return left
```

为什么用left <= right而不是left < right：确保在两者相等的时候再进一次循环，从而避免额外处理

快速排序
nums[left] < pivot, nums[right] > pivot 确保每次分配平均

```
    def sortArray(self, nums: List[int]) -> List[int]:
        self.quick_sort(nums, 0, len(nums) - 1)
        return nums

    def quick_sort(self, nums, start, end):
        if start >= end:
            return None

        pivot = nums[(start + end) // 2]
        left, right = start, end

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

        self.quick_sort(nums, start, right)
        self.quick_sort(nums, left, end)
```

kth largest

```
    def findKthLargest(self, nums: List[int], k: int) -> int:
        return self.quick_select(nums, 0, len(nums) - 1, k - 1)

    def quick_select(self, nums, start, end, k):
        if start >= end:
            return nums[k]

        left, right = start, end
        pivot = nums[(start + end) // 2]

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

        if k >= left:
            return self.quick_select(nums, left, end, k)
        elif k <= right:
            return self.quick_select(nums, start, right, k)
        else:
            return nums[k]
```
Sort Colors (3 pointer)

left 的左边都是0
right的右边都是2
index 遍历其余

```
    def sortColors(self, nums: List[int]) -> None:
        if not nums:
            return

        left, index, right = 0, 0, len(nums) - 1

        while index <= right:
            if nums[index] == 0:
                nums[left], nums[index] = nums[index], nums[left]
                left += 1
                index += 1
            elif nums[index] == 1:
                index += 1
            else:
                nums[index], nums[right] = nums[right], nums[index]
                right -= 1
```

Merge

Match

https://leetcode.com/problems/heaters/

```
    def findRadius(self, houses: List[int], heaters: List[int]) -> int:
        houses = sorted(houses)
        heaters = sorted(heaters)

        i, j = 0, 0
        radius = 0
        while i < len(houses) and j < len(heaters):
            cur_radius = abs(houses[i] - heaters[j])
            next_radius = abs(houses[i] - heaters[j + 1]) if j <
len(heaters) - 1 else math.inf
            if cur_radius < next_radius:
                radius = max(radius, cur_radius)
                i += 1
            else:
                j += 1
        return radius
```

BFS

搜索问题，能用BFS就不用DFS

1. 拓扑排序
2. 连通性
3. 分层遍历
4. 简单图最短路径
5. 给定一个变换规则，从初始到终止需要的步数

拓扑排序

```
    def canFinish(self, numCourses, prerequisites):
        graph, indegree = self.build_graph(numCourses, prerequisites)
        queue = collections.deque()
        for i in range(numCourses):
            if indegree[i] == 0:
                queue.append(i)

        result = []
        while queue:
            course = queue.popleft()
            result.append(course)
            for next_course in graph[course]:
                indegree[next_course] -= 1
                if indegree[next_course] == 0:
                    queue.append(next_course)

        return len(result) == numCourses

    def build_graph(self, numCourses, prerequisites):
        graph = {i: set() for i in range(numCourses)}
        indegree = {i: 0 for i in range(numCourses)}
        for edge in prerequisites:
            if edge[0] not in graph[edge[1]]: # 防止重复边
                graph[edge[1]].add(edge[0])
                indegree[edge[0]] += 1
        return graph, indegree
```

双向BFS

```
    def extend_queue(self, queue, distance, other_distance):
        # return -1 if not found; otherwise return the distance
        while queue:
            state = queue.popleft()
            next_states = self.get_next_states(state)
            for next_state in next_states:
                if next_state in other_distance:
                    return distance[state] + other_distance[next_state]
+ 1
                if next_state in distance:
                    continue
                queue.append(next_state)
                distance[next_state] = distance[state] + 1
        return -1
```

例子：https://www.lintcode.com/problem/1565/
解法1：两重BFS，第一层BFS寻找下一个点；第二层BFS求层数

```
    def modernLudo(self, length, connections):
        graph = self.build_graph(length, connections)
        queue = collections.deque([1])
        distance = {1: 0}

        while queue:
            i = queue.popleft()
            for next_node in range(i + 1, min(i + 7, length + 1)):
                connected_nodes = self.get_connected_nodes(next_node,
graph, distance)
                for connected_node in connected_nodes:
                    queue.append(connected_node)
                    distance[connected_node] = distance[i] + 1

        return distance[length]

    def get_connected_nodes(self, i, graph, distance):
        queue = collections.deque([i])
        visited = set()
        if i not in distance: # 这里比较tricky
            visited.add(i)

        while queue:
            node = queue.popleft()
            if node in distance:
                continue

            for connected_node in graph[node]:
                if connected_node in distance or connected_node in
visited:
                    continue
                queue.append(connected_node)
                visited.add(connected_node)
        return visited

    def build_graph(self, length, connections):
        graph = {i: set() for i in range(1, length + 1)}
        for a, b in connections:
            graph[a].add(b)
        return graph
```

解法2: 两个队列/数组 BFS

```
def modernLudo(self, length, connections):
        graph = self.build_graph(length, connections)
        queue = [1]
        distance = {1 : 0}

        while queue:
            for node in queue:
                for next_node in graph[node]:
                    if next_node not in distance:
                        queue.append(next_node)
                        distance[next_node] = distance[node]

            next_queue = []
            for node in queue:
                for next_node in range(node + 1, min(node + 7, length +
1)):
                    if next_node not in distance:
                        next_queue.append(next_node)
                        distance[next_node] = distance[node] + 1
            queue = next_queue

        return distance[length]
```

解法3: SPFA Shortest Path Faster Algorithm (queue)
SPFA的复杂度大约是O(kE),k是每个点的平均进队次数(一般的，k是一个常数，在稀疏图中小于2)。

```
    def modernLudo(self, length, connections):
        graph = self.build_graph(length, connections)
        queue = collections.deque([1])
        distance = {i: math.inf for i in range(1, length + 1)}
        distance[1] = 0  # 注意初始化

        while queue:
            node = queue.popleft()
            for connected_node in graph[node]:
                if distance[connected_node] > distance[node]:
                    queue.append(connected_node)
                    distance[connected_node] = distance[node]
            for next_node in range(node + 1, min(node + 7, length + 1)):
                if distance[next_node] > distance[node] + 1:
                    queue.append(next_node)
                    distance[next_node] = distance[node] + 1

        return distance[length]
```

解法4 SPFA + deque
解法3中 append改成appendleft 但是会快不少

解法5 SPFA + heap

```
def modernLudo(self, length, connections):
        from heapq import heappush, heappop
        graph = self.build_graph(length, connections)

        distance = {i: math.inf for i in range(1, length + 1)}
        distance[1] = 0
        heap = []
        heappush(heap, (0, 1))   # distance, index

        while heap:
            cur_distance, node = heappop(heap)
            for connected_node in graph[node]:
                if distance[connected_node] > cur_distance:
                    heappush(heap, (cur_distance, connected_node))
                    distance[connected_node] = cur_distance
            for next_node in range(node + 1, min(node + 7, length + 1)):
                if distance[next_node] > cur_distance + 1:
                    heappush(heap, (cur_distance + 1, next_node))
                    distance[next_node] = cur_distance + 1

        return distance[length]
```

解法6 Dijkstra
seen 代表计算过距离的节点
对于不在seen中的 如果更接近就更新

```
    def modernLudo(self, length, connections):
        from heapq import heappush, heappop
        graph = self.build_graph(length, connections)

        distance = {i: math.inf for i in range(1, length + 1)}
        distance[1] = 0
        seen = set()
        heap = []
        heappush(heap, (0, 1))

        while len(seen) < length:
            cur_distance, cur_node = heappop(heap)
            seen.add(cur_node)
            for connected_node in graph[cur_node]:
                if connected_node not in seen and 
distance[connected_node] > distance[cur_node]:
                    heappush(heap, (distance[cur_node], connected_node))
                    distance[connected_node] = distance[cur_node]

            for next_node in range(cur_node + 1, min(cur_node + 7, 
length + 1)):
                if next_node not in seen and distance[next_node] > 
distance[cur_node] + 1:
                    heappush(heap, (distance[cur_node] + 1, next_node))
                    distance[next_node] = distance[cur_node] + 1

        return distance[length]
```

解法7 DP
坐标型

```
    def modernLudo(self, length, connections):
        graph = self.build_graph(length, connections)
        dp = [math.inf] * (length + 1)
        dp[1] = 0

        for i in range(2, length + 1):
            for j in graph[i]:
                dp[i] = min(dp[i], dp[j])
            for j in range(max(1, i - 6), i):
                dp[i] = min(dp[i], dp[j] + 1)
        return dp[length]
```

解法8 memoization

```
    def modernLudo(self, length, connections):
        graph = self.build_graph(length, connections)
        return self.get_distance(length, graph, {})

    def get_distance(self, pos, graph, cache):
        if pos <= 0:
            return math.inf
        if pos == 1:
            return 0
        if pos in cache:
            return cache[pos]

        distance = math.inf
        for connected_node in graph[pos]:
            distance = min(distance, self.get_distance(connected_node, 
graph, cache))
        for last_node in range(pos - 6, pos):
            distance = min(distance, self.get_distance(last_node, graph, 
cache) + 1)
        cache[pos] = distance
        return distance
```

DFS / Backtracking

适用范围：

1. 树的分治和遍历
2. 求所有方案
3. 求排列组合

复杂度 调用的次数*单次处理复杂度
TODO:

可行性剪枝：word square - 利用prefix先把不可能的prune
重复性剪枝：word search III - 先搜A后搜B和先搜B后搜A相同 prune

最优性剪枝（顺序）：sliding puzzle

Binary Search

适用条件：

1. 排序数据
2. < O(n)
3. 找到数组中某个位置，左半边满足某个条件，右半边不满足
4. 找到一个最大/最小的值使得某个条件满足

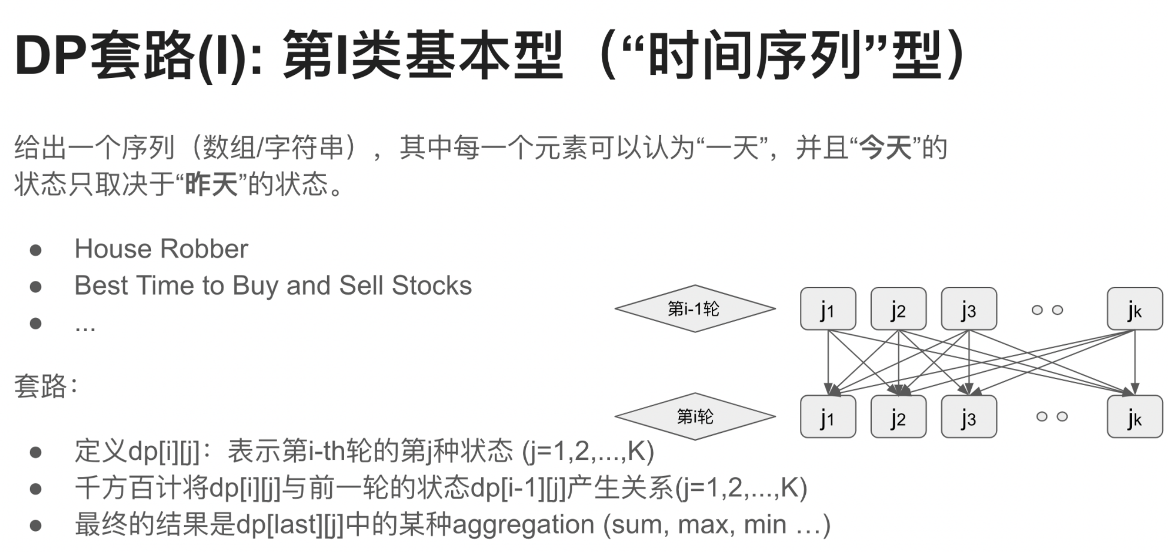
TODO:

Divide and Conquer

TODO:

Dynamic Programming

序列型1: 只和i-1有关



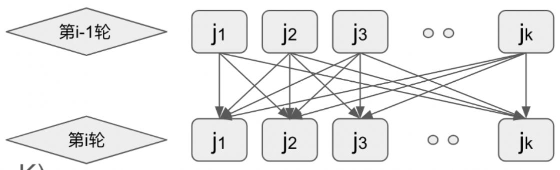
DP套路(I): 第I类基本型（“时间序列”型）

给出一个序列（数组/字符串），其中每一个元素可以认为“一天”，并且“**今天**”的状态只取决于“**昨天**”的状态。

- House Robber
- Best Time to Buy and Sell Stocks
- ...

套路：

- 定义dp[i][j]：表示第i-th轮的第j种状态 (j=1,2,...,K)
- 千方百计将dp[i][j]与前一轮的状态dp[i-1][j]产生关系(j=1,2,...,K)
- 最终的结果是dp[last][j]中的某种aggregation (sum, max, min ...)



例：Buy and Sell Stock III

```
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        dp = [[0, -math.inf, -math.inf, -math.inf], [0, 0, 0, 0]]

        for i in range(1, n):
            dp[i % 2][0] = max(dp[(i - 1) % 2][0] + prices[i] - prices[i
- 1], 0)
            dp[i % 2][1] = max(dp[(i - 1) % 2][0] + prices[i] - prices[i
- 1], dp[(i - 1) % 2][1])
            dp[i % 2][2] = max(dp[(i - 1) % 2][2] + prices[i] - prices[i
- 1], dp[(i - 1) % 2][1])
            dp[i % 2][3] = max(dp[(i - 1) % 2][2] + prices[i] - prices[i
- 1], dp[(i - 1) % 2][3])
```

```
        return max(0, dp[(n - 1) % 2][1], dp[(n - 1) % 2][3])
```

序列1.5: 和固定位数i-1, i-2, .., i-k有关

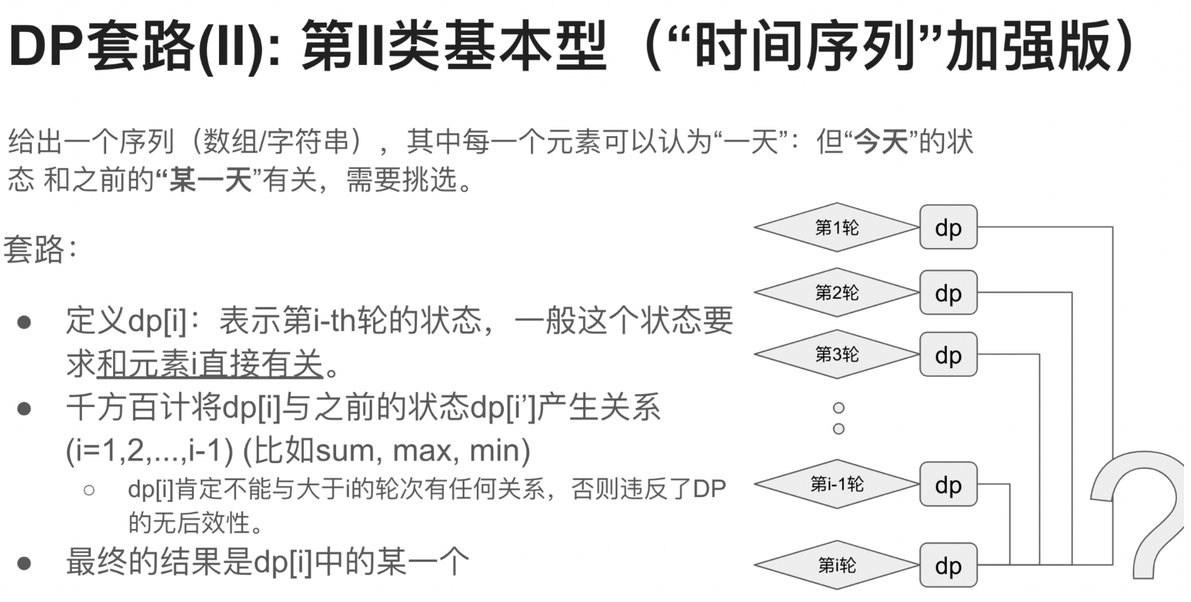
例1: Decoding Ways
例2: Regex Matching / Wildcard Matching

```
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)

        dp = [[False] * (n + 1) for _ in range(m + 1)]
        dp[0][0] = True

        for i in range(m + 1):
            for j in range(1, n + 1):
                if i > 0 and (s[i - 1] == p[j - 1] or p[j - 1] == "."):
                    dp[i][j] = dp[i - 1][j - 1]
                elif p[j - 1] == "*":
                    dp[i][j] = dp[i][j - 2]
                    if s[i - 1] == p[j - 2] or p[j - 2] == ".":
                        dp[i][j] = dp[i][j] or dp[i - 1][j]
        return dp[m][n]
```

序列型2: 和可变位数i-1, i-2, …, i - k有关



DP套路(II): 第II类基本型（“时间序列”加强版）

给出一个序列（数组/字符串），其中每一个元素可以认为“一天”：但“**今天**”的状态 和之前的“**某一天**”有关，需要挑选。

套路:

- 定义dp[i]：表示第i-th轮的状态，一般这个状态要求和元素i直接有关。
- 千方百计将dp[i]与之前的状态dp[i']产生关系 (i=1,2,...,i-1) (比如sum, max, min)
  - dp[i]肯定不能与大于i的轮次有任何关系，否则违反了DP的无后效性。
- 最终的结果是dp[i]中的某一个

第1轮
dp
第2轮
dp
第3轮
dp
第i-1轮
dp
第i轮
dp
?

最长递增子序列

```
    def lengthOfLIS(self, nums: List[int]) -> int:
        if not nums:
            return 0
        dp = [1] * len(nums)
        for i in range(1, len(nums)):
            j = i - 1
            while j >= 0:
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)
                j -= 1
        return max(dp)
```

O(NlogN)解法

```
def lengthOfLIS(self, nums: List[int]) -> int:
        dp = [] # dp[i] = smallest number with (i + 1)length LIS
        for num in nums:
            self.insert(dp, num)
        return len(dp)

    def insert(self, array, target):
        if not array or target > array[-1]:
            array.append(target)
            return None

        start, end = 0, len(array) - 1
        while start + 1 < end:
            mid = (start + end) // 2
            if target > array[mid]:
                start = mid
            elif target < array[mid]:
                end = mid
            else:
                return None

        if target < array[start]:
            array[start] = target
        elif target < array[end] and target > array[start]:
            array[end] = target
```

另一个例子https://leetcode.com/problems/filling-bookcase-shelves/

```
    def minHeightShelves(self, books: List[List[int]], shelf_width: int)
-> int:
        # dp[i] min_height with first i books

        dp = [math.inf] * (len(books) + 1)
        dp[0] = 0
        for i in range(1, len(books) + 1):
            j = i
            current_sum = 0
            current_max = 0
            while j > 0:
                current_sum += books[j - 1][0]
                current_max = max(current_max, books[j - 1][1])
                if current_sum <= shelf_width:
                    dp[i] = min(dp[i], current_max + dp[j - 1])
                    j -= 1
                else:
                    break
        return dp[len(books)]
```

双序列型

DP套路(III): 双序列型

给出两个序列s和t（数组/字符串），让你对它们搞事情。

- Longest Common Subsequences
- Shortest Common Supersequence
- Edit distances
- ...

套路：

- 定义dp[i][j]：表示针对s[1:i]和t[1:j]的子问题的求解。
- 千方百计将dp[i][j]往之前的状态去转移：dp[i-1][j], dp[i][j-1], dp[i-1][j-1]
- 最终的结果是dp[m][n]

最长公共子序列

```
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp[i][j] = 1 + dp[i - 1][j - 1]
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
        return dp[m][n]
```

最短公共supersequence

```
    def shortestCommonSupersequence(self, str1: str, str2: str) -> str:
        # 0: first, 1: second: 2: both, -1: unknown

        m, n = len(str1), len(str2)
        dp = [[math.inf] * (n + 1) for _ in range(m + 1)]
        prev = [[-1] * (n + 1) for _ in range(m + 1)]

        for i in range(m + 1):
            dp[i][0] = i
            prev[i][0] = 0
        for j in range(n + 1):
            dp[0][j] = j
            prev[0][j] = 1
        prev[0][0] = 2

        dp[0][0] = 0
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1
                if str1[i - 1] == str2[j - 1] and dp[i][j] > dp[i - 1][j
 - 1] + 1:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    prev[i][j] = 2
                elif dp[i][j] == dp[i - 1][j] + 1:
                    prev[i][j] = 0
                else:
                    prev[i][j] = 1

        path = []
        while m > 0 or n > 0:
            if prev[m][n] == 0:
                path.append(str1[m - 1])
                m -= 1
            elif prev[m][n] == 1:
                path.append(str2[n - 1])
                n -= 1
            else:
                path.append(str1[m - 1])
                m -= 1
                n -= 1

        return "".join(reversed(path))
```

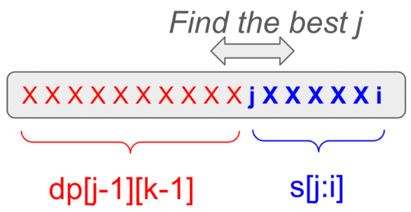
N个区间

DP套路(IV): 第I类区间型DP

给出一个序列，明确要求**分割成K个连续区间**，要你计算这些区间的某个最优性质。

套路：

- 状态定义：dp[i][k]表示针对s[1:i]分成k个区间，此时能够得到的最优解
- 搜寻**最后一个区间的起始位置j**，将dp[i][k]分割成dp[j-1][k-1]和s[j:i]两部分。
- 最终的结果是dp[N][K]



(分成k个子串)

```
 def palindromePartition(self, string: str, total_splits: int) -> int:
        # dp[i][k] = the minimal number of characters to change with
        # first i characters (1 based) partitioned into k splits

        n = len(string)
        num_changes = [[0] * n for _ in range(n)]
        for i in range(n - 1):
            num_changes[i][i + 1] = (0 if string[i] == string[i + 1]
else 1)

        for k in range(2, n):
            for i in range(n - k):
                num_changes[i][i + k] = num_changes[i + 1][i + k - 1] +
(0 if string[i] == string[i + k] else 1)

        dp = [[math.inf] * (total_splits + 1) for _ in range(len(string)
+ 1)]
        dp[0][0] = 0

        for i in range(1, len(string) + 1):
            for k in range(1, min(i, total_splits) + 1):
                for j in range(i, k - 1, -1):
                    dp[i][k] = min(dp[i][k], dp[j - 1][k - 1] +
num_changes[j - 1][i - 1])
        return dp[len(string)][total_splits]
```

区间跟长度相关

DP套路(V): 第II类区间型DP

只给出一个序列S（数组/字符串），求一个针对这个序列的最优解。

适用条件：这个最优解对于序列的index而言，没有“无后效性”。即无法设计dp[i]使得dp[i]仅依赖于dp[j] (j<i). 但是大区间的最优解，可以依赖小区间的最优解。

套路：

- 定义dp[i][j]：表示针对s[i:j]的子问题的求解。
- 千方百计将大区间的dp[i][j]往小区间的dp[i’][j’]转移。
    - 第一层循环是区间大小；第二层循环是起始点。
- 最终的结果是dp[1][N]

https://leetcode.com/problems/palindrome-removal/

```
    def minimumMoves(self, arr: List[int]) -> int:
        n = len(arr)
        dp = [[math.inf] * n for _ in range(n)]

        for i in range(n):
            dp[i][i] = 1

        for i in range(n - 1):
            dp[i][i + 1] = (1 if arr[i] == arr[i + 1] else 2)

        for length in range(3, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                dp[i][j] = min(dp[i][j - 1] + 1, dp[i][j - 2] + dp[j -
1][j])
                if arr[i] == arr[j]:
                    dp[i][j] = min(dp[i][j], dp[i + 1][j - 1])
                for k in range(i + 1, j):
                    if arr[k] == arr[j]:
                        dp[i][j] = min(dp[i][j], dp[i][k - 1] + dp[k +
1][j - 1])
        return dp[0][n - 1]
```

Burst Ballon

```
    def maxCoins(self, nums: List[int]) -> int:
        # reframe problem as before
        nums = [1] + nums + [1]
        n = len(nums)

        # dp will store the results of our calls
        dp = [[0] * n for _ in range(n)]

        for right in range(1, n + 1):
            for left in range(right - 1):
                dp[left][right] = max(nums[left] * nums[i] * nums[right]
+ dp[left][i] + dp[i][right] for i in range(left + 1, right))
        return dp[0][n-1]
```

背包型

DP套路(VI): 背包入门

题型抽象：给出N件物品，**每个物品可用可不用**（或者有若干个不同的用法）。要求以某个有上限C的代价来实现最大收益。（有时候反过来，要求以某个有下限的收益来实现最小代价。）

套路：

- 定义dp[i][c]：表示考虑只从前i件物品的子集里选择、代价为c的最大收益。
    - c = 1,2,...,C
- 千方百计将dp[i][c]往dp[i-1][c']转移：即考虑如何使用物品i，对代价/收益的影响
    - 第一层循环是物品编号i；
    - **第二层循环是遍历"代价"的所有可能值。**
- 最终的结果是 max {dp[N][c]}, for c=1,2,...,C

```
    def backPackII(self, W, weights, values):
        n = len(weights)

        dp = [[0] * (W + 1) for _ in range(n + 1)]
        for i in range(1, n + 1):
            for j in range(1, W + 1):
                if j >= weights[i - 1]:
                    dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i
- 1]] + values[i - 1])
                else:
                    dp[i][j] = dp[i - 1][j]
        return max(dp[n])
```

状态压缩

对于比较复杂的“状态”，DP经常会用到“状态压缩”的技巧。

比如：有些情况下如果想设计“状态”代表一个01向量（不超过32位），我们可以用一个整形的bit位来表示。

[1,0,1,1,0,0,1] => b1011001 => 89

Sweep Line

The Skyline Problem

1. Sweeper Line + HashMap

```
from heapq import heappush, heappop
class MaxHeap:
    def __init__(self):
        self.data = []
        self.deleted_indices = set()

    def push(self, index, value):
        heappush(self.data, (-value, index))

    def pop(self):
        self._lazy_delete()
        return -heappop(self.data)[0]

    def top(self):
        self._lazy_delete()
        return -self.data[0][0]

    def delete(self, index):
        self.deleted_indices.add(index)

    def _lazy_delete(self):
        while self.data and self.data[0][1] in self.deleted_indices:
            heappop(self.data)

class EventType:
    START = 0
    END = 1

class Solution:
    def getSkyline(self, buildings: List[List[int]]) -> List[List[int]]:
        events = []
        for index, building in enumerate(buildings):
            start, end, height = tuple(building)
            events.append((start, EventType.START, -height, index))
            events.append((end, EventType.END, height, index))

        events.sort()

        heap = MaxHeap()
        heap.push(-1, 0)

        result = []
        for pos, event_type, height, index in events:
            if event_type == EventType.START:
                height = -height
                cur_height = heap.top()
                if height > cur_height:
                    if not result or result[-1][1] != height:
                        result.append([pos, height])
                heap.push(index, height)
            else:
                cur_height = heap.top()
                heap.delete(index)
                if height == cur_height:
                    cur_height = heap.top() if help else 0
                    if not result or result[-1][1] != cur_height:
                        result.append([pos, cur_height])
        return result
```

2. Sweeper Line + TreeMap

```
def getSkyline(self, buildings: List[List[int]]) -> List[List[int]]:
        events = [(building[0], 1, building[2]) for building in
buildings]
        events.extend([(building[1], 0, building[2]) for building in
buildings])
        events.sort()

        from sortedcontainers import SortedDict
        heights = SortedDict()
        outline = []
        for event in events:
            time, is_start, height = event
            if is_start:
                if height not in heights:
                    heights[height] = 1
                else:
                    heights[height] += 1

            else:
                heights[height] -= 1
                if heights[height] == 0:
                    del heights[height]

            if len(heights) == 0:
                outline.append([time, 0])
            elif not outline:
                outline.append([time, height])
            elif outline[-1][0] == time:
                outline[-1][1] = max(outline[-1][1], height)
                if len(outline) > 1 and outline[-1][1] ==
outline[-2][1]:
                    outline.pop()
            elif outline[-1][1] != heights.peekitem()[0]:
                outline.append([time, heights.peekitem()[0]])

        return outline
```

Greedy

Concurrency

Running multiple processes/threads to speedup

speedup IO bound jobs (for CPU bound ones, use multiprocessing)
 I/O bound: read/write file system, network operations
if use threads for CPU bound, it can make the program slower due to overhead on create/destroy

```
threading.Thread(target, args)
# only __init__, run() should be overridden
# blocking = False, still run the program (check if the lock is released
without blocking the method)
with concurrent.futures.ThreadPoolExecutor as executor:
    future = executor.submit(target, *args)
results = executor.map(target, *arg) # in the order of thread starting
# do not need to join explicitly
```

Synchronization

https://leetcode.com/problems/print-in-order/discuss/335939/5-Python-threading-solutions-(Barrier-Lock-Event-Semaphore-Condition)-with-explanation

Barrier

当满足数目的thread都wait了之后，同时release

```
b = Barrier(2, timeout=5)
b.wait(timeout=None)
reset() => BrokenBarrierException
```

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously.

Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available,

RLock 可以被同一线程多次获取 多次释放

Event Objects

```
event = Event()
event.set()
event.wait()
```

Semaphore Objects

semaphore = Semaphore(value=1)

Lock

The underlying abstraction used to implement the later constructs. It controls access to one resource for one thread.

Condition

Another primitive used with Lock to implement the other structures. Gives you finer control over what happens after you release a lock.

Semaphore

Can be used to share one resource among a limited number of threads. Can be chained together (e.g. a task releases on semaphore1 every time it executes. Another thread acquires semaphore1 10 times and releases semaphore2 to indicate the task completion.

Barrier

Once the barrier threshold is reached, every thread will be passed through - could be a good fit for batched processes where you want to wait on a certain percentage before starting a process, but accept the remainder.

Event

Contrary to the traditional 'event' concept in other forms (one event results in one callback) - once it is in a triggered state, all threads will not block on the 'wait' call until the event is 'cleared' - reset to the untriggered state. It could be thought of as a Barrier(1) that can be reset easily.

Large Scale

External Sorting

1. 将大文件切分为若干个个小文件，并分别使用内存排好序
2. 使用K路归并算法将若干个排好序的小文件合并到一个大文件中
    a. 用堆进行k路归并
    b. 每次归并的时候不一定要逐个字符读，可以设置一个缓冲区

Map Reduce

HashHeap

BloomFilter

Bitmap

Python Syntax

```
map(lambda x: x * x, iterable)
reduce(lambda x, y: x + y, iterable)
filter(lambda x: x > 0, iterable)
any([0, False, empty])
all([0, False, empty])
```

frozenset hashable 可以用作其他集合元素
iter

python cookbook
只要元素个数一样就可以解压 from list or tuple or string or iterable?

* 可以解决个数不匹配问题 a, b, *c = iterable

previous_lines = deque(maxlen=history) 保留最后history项

heapq.nlargest, nsmallest

priority queue heapq(priority, index, anything) <index用来避免anything不可比较的情况 ）

defaultdict

为了能控制一个字典中元素的顺序，你可以使用 collections 模块中的
OrderedDict 类

min_price = min(zip(prices.values(), prices.keys())) zip只能访问一次

a.keys() & b.keys() # { 'x', 'y' }

a

named slice. a = slice(20, 23) b = A[a]

collections.Counter 类就是专门为这类问题而设计的，它甚至有一个有用的
most_common()

from collections import namedtuple
 Subscriber = namedtuple('Subscriber', ['addr', 'joined']) ub =
Subscriber('jonesy@example.com', '2012-10-19')
Stock = namedtuple('Stock', ['name', 'shares', 'price']) def compute_cost(records):

total = 0.0
for rec in records:
s = Stock(*rec)
total += s.shares * s.price return total
尽管 namedtuple 的实例看起来像一个普通的类实例，但是它跟元组类型是可交换 的，支持所有的普通元组操作，比如索引和解压
14.在转换序列上直接做聚合

nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums]) # 不好
s = sum(x * x for x in nums)
ChainMap

from collections import ChainMap c = ChainMap(a,b)
print(c['x']) # Outputs 1 (from a) print(c['y']) # Outputs 2 (from b) print(c['z']) # Outputs 3 (from a)

并不是真正合并了
对于字典的更新或删除操作总是影响的是列表中第一个字典。

作为 ChainMap 的替代，你可能会考虑使用 update() 方法将两个字典合并
random.randrange
random.choice

Heap:

heapq.heappush(heap, item)
heapq.heappop(heap)
heapq.heappushpop(heap, item)
heapq.heapify(x)
heapq.nlargest
heapq.nsmallest
first push then pop to avoid comparison

collections.Counter

How to explain verbally

Sliding Window

Naive method is to enumerate all substrings by two loops...There are a lot of duplicate or redundant operations/computations in the brute-force solution. For example, when we start enumerating all the substrings from the character a, ..., we do not need to check the rest of . We can move forward

Sliding window is a range of elements identified by their start and end indices, where both indices can slide to a certain direction. They never slide towards the back, so the time complexity is linear.

BFS

transformation/dependency reminds me of graph data structure
We will essentially be working with an undirected and unweighted graph with words as nodes and edges between words which differ by just one letter. The problem boils down to finding the shortest path from a start node to a destination node, if there exists one. Hence it can be solved using the Breadth First Search approach.
(双向）The search space considered by the breadth first search algorithm depends upon the branching factor of the nodes at each level. If the branching factor remains the same for all the nodes, the search space increases exponentially along with the number of levels. We can considerably cut down the search space of the standard breadth first search algorithm if we launch two simultaneous BFS.

Backtracking

Backtracking is a general algorithm for finding all (or some) solutions which incrementally builds candidates to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution. Conceptually, one can imagine the procedure of backtracking as the tree traversal. Starting from the root node, one sets out to search for solutions that are located at the leaf nodes. Each intermediate node represents a partial candidate solution that could potentially lead us to a final valid solution. At each node, we would fan out to move one step further to the final solution, i.e. we iterate the child nodes of the current node. Once we can determine if a certain node cannot possibly lead to a valid solution, we abandon the current node and backtrack to its parent node to explore other possibilities.