- 滑动窗口, 双指针
  - 滑动窗口 :
    - 3. 无重复字符的最长子串 - 力扣（LeetCode）
  - 双指针 :
    - *11. 盛最多水的容器 - 力扣（LeetCode）
  - 快慢指针 (Floyd's Tortoise and Hare Algorithm)
  - 单调栈 :
    - 下一个更大（或更小）元素 : 84. 柱状图中最大的矩形 - 力扣（LeetCode）
    - 84. 柱状图中最大的矩形 - 力扣（LeetCode）
    - 85. 最大矩形 - 力扣（LeetCode）
  - 单调队列 :
    - 239. 滑动窗口最大值 - 力扣（LeetCode） (也可以heap + 懒删除)
- Stack
    - 中序表达式转后序表达式
    - LCR 036. 逆波兰表达式求值 - 力扣（LeetCode）
- 排序
    - Merge Sort, OpenJudge - 07622:求排列的逆序数
- Linked List
  - 引用与赋值
    - 206. 反转链表 - 力扣（LeetCode）
- Tree
    - 手搓Heapq
  - 并查集 Disjoint Set
- Graph
  - DFS / BFS
    - 判断无向图有无环
  - 拓扑排序 (可用于判断有向图中有无环)
    - Kahn, 时间复杂度 $O(V+E)$
    - DFS, 时间复杂度 $O(V+E)$
  - 最短路径
    - Dijkstra
    - Bellman-Ford $O(VE)$
    - SPFA
    - Floyd-Warshall $O(V^3)$, 类似dp
  - 最小生成树
    - Prim, $O(V^2)$, 适用于稠密图
    - Kruskal, $O(E\log E)$
  - 强连通单元 (SCCs)
  - 启发式搜索(Warnsdorff)
    - OpenJudge - 28050:骑士周游
- KMP模式匹配

滑动窗口, 双指针

滑动窗口：

[3. 无重复字符的最长子串 - 力扣（LeetCode）](#)

```python
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        if not s: return 0
        left, MAX = 0, 1
        pos_val = {}
        for right, c in enumerate(s):
            if s[right] in pos_val and pos_val[s[right]] >= left:
                left = pos_val[s[right]] + 1
            pos_val[s[right]] = right  # pos_val : left ~ right
            if right - left + 1 > MAX:
                MAX = right - left + 1
        return MAX
```

双指针：

[*11. 盛最多水的容器 - 力扣（LeetCode）](#)

```python
class Solution(object):
    def maxArea(self, height):
        left, right = 0, len(height) - 1
        MAX = 0
        while left <= right:
            if height[left] <= height[right]:
                MAX = max(MAX, (right - left) * height[left])
                left += 1
            else:
                MAX = max(MAX, (right - left) * height[right])
                right -= 1
        return MAX
```

快慢指针 (Floyd's Tortoise and Hare Algorithm)

求链表中点, 判断链表是否有圈

单调栈：

下一个更大（或更小）元素：[84. 柱状图中最大的矩形 - 力扣（LeetCode）](#)

[84. 柱状图中最大的矩形 - 力扣（LeetCode）](#)

例如 `[3, 1, 4, 1, 5, 9, 2, 6]`

`[3]` => `[1]` => `[1,4]` => `[1]` => `[1,5]` => `[1,5,9]` => `[1,2]` => `[1,2,6]`

例如其中 `pop 4` 时, 会计算以 `4` 为右边界矩形 (`4` 为矩形高度, `4` 为矩形最右侧一列) 的最大面积.

其中在 `height` 末尾加 `0` 是为了保证最后把 `[1,2,6]` 完整的 `pop` 一遍

例如 `pop 2` 时计算以 `2` 为右边界矩形 (`2` 为矩形高度, `2` 为矩形最右侧一列) 的最大面积, 即 `[5, 9, 2]` 三列组成的矩形

```python
class Solution(object):
    def largestRectangleArea(self, heights):
        heights.append(0)
        st = []
        MAX = 0
        for i in range(len(heights)):
            while st and heights[st[-1]] > heights[i]:
                h = heights[st.pop()]
                w = i if not st else i - st[-1] - 1 # st 为空表明 heights[i]是目前最小的
                MAX = max(MAX, h * w)
            st.append(i)
        return MAX
```

85. 最大矩形 - 力扣（LeetCode）

```python
class Solution(object):
    def maximalColum(self, col):
        col.append(0)
        st = []
        MAX = 0
        for i, x in enumerate(col):
            while st and col[st[-1]] > x:
                if len(st) >= 2:
                    MAX = max(MAX, col[st[-1]] * (i - st[-2] - 1))
                else:
                    MAX = max(MAX, col[st[-1]] * i)
                st.pop()
            st.append(i)
        return MAX
    def maximalRectangle(self, matrix):
        m, n = len(matrix), len(matrix[0])
        pre = [0] * n
        MAX = 0
        for i in range(m):
            for j in range(n):
                pre[j] = pre[j] + 1 if matrix[i][j] == "1" else 0
            MAX = max(MAX, self.maximalColum(pre.copy()))
        return MAX
```

单调队列：

例如 `nums = [1,3,-1,-3,5,3,6,7], k = 3`

`[1]` => `[3]` => `[3, -1]` => `[3, -1, -3]` => `[5]` => `[5, 3]` => `[6]` => `[7]`

239. 滑动窗口最大值 - 力扣（LeetCode） **(也可以heap + 懒删除)**

```python
from collections import deque

class Solution(object):
    def maxSlidingWindow(self, nums, k):
        dq = deque([])
        res = []
        for i, x in enumerate(nums):
```

```python
            while dq and dq[0] <= i - k:
                dq.popleft()
            while dq and nums[dq[-1]] <= x:
                dq.pop()
            dq.append(i)
            if i >= k - 1:
                res.append(nums[dq[0]])
        return res
```

Stack

[中序表达式转后序表达式]()

```python
opr_pri = {"+" : 1, "-" : 1, "*" : 2, "/" : 2, "(" : 3, ")" : 3}

def infix_to_postfix_list(tokens: list[int | str]) -> list[int | str]:
# e.g. ["(", 2, "+", 6, "/", 3 , ")", "*", 4] -> [2, 6, 3, '/', '+', 4, '*']
    res, opr_st = [], [] # 初始化运算符栈和输出栈为空
    for tok in tokens:
        if tok == "(": # 如果是左括号，则将其推入运算符栈.
                opr_st.append("(")

        elif tok == ")": # 如果是右括号
            while opr_st and opr_st[-1] != "(":
                res.append(opr_st.pop())
                # 则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号.
            opr_st.pop() #将左括号弹出但不添加到输出栈中.

        elif tok in opr_pri: # 如果是运算符
            while opr_st and opr_st[-1] != "(" and \
                opr_pri[tok] <= opr_pri[opr_st[-1]]:
                res.append(opr_st.pop())
                # 不断将运算符栈顶的运算符弹出并添加到输出栈中，
                # 直到运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号
            opr_st.append(tok) # 则将当前运算符推入运算符栈

        else: # 如果是操作数(数字)，则将其添加到输出栈.
            res.append(tok)
    while opr_st:
        res.append(opr_st.pop()) # 输出栈中的元素就是转换后的后缀表达式.
    return res
```

[LCR 036. 逆波兰表达式求值 - 力扣（LeetCode）]()

```python
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        num = []
        for token in tokens:
            if token not in {"*", "/", "+", "-"}:
                num.append(int(c))
            else:
                b = num.pop()
                a = num.pop()
                if token == "+":
```

```
                    num.append(a + b)
                elif token == "-":
                    num.append(a - b)
                elif token == "*":
                    num.append(a * b)
                else:
                    num.append(int(a / b))
        return num[0]
```

排序

Merge Sort, [OpenJudge - 07622:求排列的逆序数]()

```
def merge_count(arr1, arr2):
    cnt, j = 0, 0
    for x in arr1:
        while j < len(arr2) and arr2[j] < x:
            j += 1
        cnt += j
    res, i, j = [], 0, 0
    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            res.append(arr1[i]); i += 1
        else:
            res.append(arr2[j]); j += 1
    return res + arr1[i:] + arr2[j:], cnt

def sortArray(nums):
    if not nums or len(nums) == 1:
        return nums, 0
    mid = len(nums) // 2
    arr1, sum1 = sortArray(nums[:mid])
    arr2, sum2 = sortArray(nums[mid:])
    arr, cnt = merge_count(arr1, arr2)
    return arr, sum1 + sum2 + cnt
```

Linked List

引用与赋值

```
# 定义链表节点类
class ListNode:
    def __init__(self, val, next = None):
        self.val = val
        self.next = next
    def __str__(self):
        return f"ListNode({self.val} -> {self.next.val})"

d = ListNode(4)
c = ListNode(3, d)
b = ListNode(2, c)
a = ListNode(1, b)
```

1.
```python
# Example 1 : `prev` 和 `cur` 指向相同的节点, 修改 `prev` 后 `cur` 不受影响
prev = a
cur = prev
prev = b
print(cur == a, a) # output : True ListNode(1 -> 2)
```

2.
```python
# Example 2 : `cur` 指向 `a.next` (i.e. `b`), 修改 `prev` 后 `cur` 不受影响
prev = a
cur = prev.next
prev = c
print(cur == b, b) # output : True ListNode(2 -> 3)
```

3.
```python
# Example 3 : `cur` 指向 `a`, 修改 `a.val`, `cur.val` 也受影响
cur = a
a.val = 0
print(cur) # output : ListNode(0 -> 2)
```

4.
```python
# Example 4 : `prev` 和 `cur` 指向相同对象 `a`, 修改 `prev.val`, `cur.val` 也受影响
prev = a
cur = a
prev.val = 0
print(cur) # output : ListNode(0 -> 2)
```

5.
```python
# Example 5 : `cur` 指向 `a`, 修改 `a.next`, `cur.next` 也受影响
prev = a
cur = prev
prev.next = c
print(cur) # output : ListNode(0 -> 3)
```

引用变更不会同步, 赋值变更 (`prev.next = ...` 或者 `prev.val = ...`) 会同步

6. `p.next` **需要提前检查** `if not p`

[206. 反转链表 - 力扣（LeetCode）](#)

```python
class ListNode:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next

class Solution(object):
    def reverseList(self, head):
        pre = None
        cur = head
        while cur:
            cur_next = cur.next
            cur.next = pre
            pre = cur
            cur = cur_next
        return pre
```

Tree

```python
class Tree():
    def __init__(self, val = 0, left = None, right = None):
        self.val = val
        self.left = left
        self.right = right
```

手搓Heapq

此略

并查集 Disjoint Set

- 常规版见后Kruskal
- 变种：以食物链 为例 (类似的, 发现它，抓住它 也可以看成一种食物链)

  我们构建 `parent` 为长度 $3n$ 的 `list`

  如果 `a`, `b` 同类, 则将 `a, b` 分支合并, `a + n, b + n` 合并, `a + 2 * n, b + 2 * n` 分支合并

  如果 `a` 吃 `b`，则将 `a, b + n` 分支合并, `a + n, b + 2 * n` 分支合并, `a + 2 * n, b` 分支合并

  如果 `a` 被 `b` 吃, 则将 `a, b + 2 * n` 分支合并, `a + n, b` 分支合并, `a + 2 * n, b` 分支合并

Graph

DFS / BFS

判断无向图有无环

```python
def has_cycle_dfs(n, graph):
    visited = [False] * n
    def dfs(u, parent):
        for w in graph[u]:
            if not visited[w]:
                if dfs(w, u):
                    return True
            elif w != parent:
                return True
    for u in range(n):
        if not visited[u]:
            if dfs(u, -1):
                return True
    return False
```

```python
def has_cycle_bfs(n, graph):
    visited, parent = [False] * n, [-1] * n
    for u in range(n):
        if not visited[u]:
            visited[u], parent[u] = True, -1
            que = deque([u])
            while que:
                cur = que.popleft()
                for w in graph[cur]:
                    if not visited[w]:
```

```
                        visited[w], parent[w] = True, cur
                        que.append(w)
                    elif w != parent[cur]:
                        return True
    return False
```

拓扑排序 (可用于判断有向图中有无环)

Kahn, 时间复杂度 $O(V+E)$

```
def topological_sort(graph : Dict[str : List[str]]):
    in_degree = defaultdict(int)
    res, que = [], deque()
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1
    for u in graph:
        if in_degree[u] == 0:
            que.append(u)
    while que:
        u = que.popleft()
        res.append(u)
        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                que.append(v)
    return res if len(res) == len(graph) else None # return None if has a cycle
```

DFS, 时间复杂度 $O(V+E)$

```
def toposort(n, graph):
    have_cycle, res = False, []
    state = [0] * n  # 0 : unvisited, 1 : visiting, 2 : visited
    def dfs(start):
        nonlocal have_cycle
        if state[start] == 1:
            have_cycle = True
        if have_cycle or state[start] == 2:
            return
        state[start] = 1
        for neighbour in graph[start]:
            if state[start] != 2:
                dfs(neighbour) # 把所有从 start 出发可到达的点放入 res 中
        state[start] = 2
        res.append(start) # 然后把 start 也放入 res 中
    for i in range(n):
        if state[i] == 0:
            dfs(i)
    return res[::-1] if not have_cycle else None
```

最短路径

Dijkstra

key : 每个点一进一出, 但要求图无负权边

Bellman-Ford $O(VE)$

```python
def bellman_ford(edges, V, source):
    dist = [float('inf')] * V # 初始化距离
    dist[source] = 0
    for _ in range(V - 1): # 松弛 V-1 次
        for u, v, w in edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    for u, v, w in edges: # 检测负权环
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("图中存在负权环")
            return None
    return dist

edges = [(0, 1, 5), (0, 2, 4), (1, 3, 3), (2, 1, 6), (3, 2, -2)] # 图是边列表，每条边是 (起点, 终点, 权重)
V, source = 4, 0 # V 总点数, source 起点
print(bellman_ford(edges, V, source))
```

SPFA

```python
def spfa(adj, V, source):
    dist = [float('inf')] * V # 初始化距离
    dist[source] = 0
    in_que = [False] * V # 初始化入队状态
    in_que[source] = True
    cnt = [0] * V # 初始化松弛次数
    que = deque([source])
    while que:
        u = que.popleft()
        in_que[u] = False # in_que 相当于存储 set(que)
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if in_que[v] == False:
                    que.append(v)
                    in_que[v] = True
                    cnt[v] += 1
                    if cnt[v] > V:
                        print("图中存在负权环")
                        return None
    return dist

adj = [[(1, 5), (2, 4)], [(3, 3)], [(1, 6)], [(2, -2)]] # 图的邻接表表示
V, source = 4, 0 # V 总点数, source 起点
print(spfa(agj, V, source))
```

Floyd-Warshall $O(V^3)$, 类似dp

```
def floyd_warshall(n, graph):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
              dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dis
```

最小生成树

Prim, $O(V^2)$, 适用于稠密图

不断往MST中添加Vertex (greedy思想, 选距离 *现有MST* 权值最小的Vertex)

```
def prim(n, matrix : List[List[int]]):
    MST, low = set(), [float("inf")] * n # low[k] 表示当前 MST 距离 k 点的最小权值.
    low[0], tot = 0, 0
    for _ in range(n):
        new, MIN = 0, float("inf")
        for i, dis in enumerate(low):
             if i not in MST and dis < MIN:
                new, MIN = i, dis
        MST.add(new)
        tot += MIN
        for i in range(n):
            if i not in MST:
                low[i] = min(low[i], matrix[i][new]) # 更新新版 MST 距离 k 点的最小权值.
    return tot
```

Kruskal, $O(E \log E)$

```
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
              self.parent[root_x] = root_y
```

```python
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_x] = root_y
                self.rank[root_y] += 1

def kruskal(n, graph):
    edges = [] # 构建边集
    for i in range(n):
        for j in range(i + 1, n):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))
    edges.sort(key=lambda x: x[2]) # 按照权重排序
    disjoint_set = DisjointSet(n) # 初始化并查集
    MST = [] # 构建最小生成树的边集
    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            MST.append((u, v, weight))
    return MST
```

强连通单元 (SCCs)

```python
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(n, graph):
    # Step 1: Perform first DFS to get finishing times
    stack, visited = [], [False] * n
    for node in range(n):
        if not visited[node]:
            dfs1(graph, node, visited, stack)
    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(n)]
    for node in range(n):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)
    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited, sccs = [False] * n, []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
```

```python
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

启发式搜索(Warnsdorff)

[OpenJudge - 28050:骑士周游]()

```python
dir = [(2, 1), (1, 2), (-1, 2), (-2, 1),
       (-2, -1), (-1, -2), (1, -2), (2, -1)]

def isValid(r, c):
    return 0 <= r < n and 0 <= c < n

def knight_tour(n, sr, sc):
    board = [[-1]*n for _ in range(n)]
    board[sr][sc] = 0
    def dfs(step, r, c):
        if step == n*n - 1:
            return True
        candidates = []
        for dr, dc in dir:
            nr, nc = r + dr, c + dc
            if isValid(nr, nc) and board[nr][nc] == -1:
                cnt = 0
                for dr2, dc2 in dir:
                    tr, tc = nr + dr2, nc + dc2
                    if isValid(tr, tc) and board[tr][tc] == -1:
                        cnt += 1
                candidates.append((cnt, nr, nc))
        candidates.sort()
        for _, nr, nc in candidates:
            board[nr][nc] = step + 1
            if dfs(step + 1, nr, nc):
                return True
            board[nr][nc] = -1
        return False
    return dfs(0, sr, sc)

n = int(input())
sr, sc = map(int, input().split())
print("success" if knight_tour(n, sr, sc) else "fail")
```

KMP模式匹配

首先 define **真前缀 (proper prefix)** 和 **真后缀(proper suffix)**

例如 `ABCD` 的真前缀为集合 `{"", A", "AB", "ABC"}` , 真后缀为 `{"", D", "CD", "BCD"}`

对于 `pattern` 构造 `lps` 表, 其中 `lps[i]` 表示 `pattern[:i]` 真前缀与真后缀交集的最大长度

```python
def compute_lps(pattern): # pattern: 模式字符串
    m = len(pattern)
    lps = [0] * m  # 初始化lps数组
    length = 0  # 当前最长前后缀长度
    for i in range(1, m):  # 注意i从1开始，lps[0]永远是0
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]  # 回退到上一个有效前后缀长度
        if pattern[i] == pattern[length]:
            length += 1
        lps[i] = length
    return lps
```

```python
def kmp_search(text, pattern): # 在 text 中查找 pattern
    n = len(text)
    m = len(pattern)
    if m == 0:
        return 0
    lps = compute_lps(pattern)
    matches = []
    j = 0  # 模式串指针
    for i in range(n):  # 主串指针
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]  # 模式串回退
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)  # 匹配成功
            j = lps[j - 1]  # 查找下一个匹配
    return matches
```