Abstract

This is simplified version, as the number of pages is limited.

COMMON

USEFUL Tools

Counter

```
from collections import Counter
a = [12, 3, 4, 3, 5, 11, 12, 6, 7]
x=Counter(a)
for i in x.keys():
      print(i, ":", x[i])
x_keys = list(x.keys()) #[12, 3, 4, 5, 11, 6, 7]
x_values = list(x.values()) #[2, 2, 1, 1, 1, 1, 1]
for i in x.elements():
    print ( i, end = " ") #[12,12,3,3,4,5,11,6,7]
c=Counter('1213123343521231255555555')
cc=sorted(c.items(),key=lambda x:x[1],reverse=True)
#[('5', 9), ('1', 5), ('2', 5), ('3', 5), ('4', 1)]
```

cmp_to_key

```
from functools import cmp_to_key
def compar(a,b):
    if a>b:
        return 1#大的在后
    if a<b:
        return -1#小的在前
    else:
        return 0#返回零不变位置
l=[1,5,2,4,6,7,6]
l.sort(key=cmp_to_key(compar))
print(l)#[1,2,4,5,6,6,7]
```

permutations

```
from itertools import permutations
# Get all permutations of [1, 2, 3]
perm = permutations([1, 2, 3])
# Get all permutations of length 2
perm2 = permutations([1, 2, 3], 2)
# Print the obtained permutations
for i in list(perm):
    print (i)
```

Number Theory

Prime

Euler Seive

```python
def euler_sieve(n):
    primes = []
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, 10002):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if i * p > 10001:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes
```

PrimeQ (single prime query)

```python
def is_prime(n):
    if n <= 1:
        return False
    elif n <= 3:
        return True
    elif n % 2 == 0:
        return False

    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1

    if n < 2047:
        bases = [2]
    elif n < 1_373_653:
        bases = [2, 3]
    elif n < 25_326_001:
        bases = [2, 3, 5]
    elif n < 3_215_031_751:
        bases = [2, 3, 5, 7]
    else:
        bases = [2, 3, 5, 7, 11]

    for a in bases:
        if a >= n:
            continue
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
```

```python
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Mod Inverse

may not exist

```python
def mod_inverse(a, m):
    g, x, y = extended_gcd(a, m)
    if g != 1:
        return None  # 不存在逆元
    else:
        return x % m  # 确保结果是正数

def extended_gcd(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        g, x1, y1 = extended_gcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return (g, x, y)
```

SORT

MergeSort

```python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
```

```
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

QuickSort

```
def quicksort(arr, left, right):
    if left < right:
        partition_pos = partition(arr, left, right)
        quicksort(arr, left, partition_pos - 1)
        quicksort(arr, partition_pos + 1, right)
def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
arr = [22, 11, 88, 66, 55, 77, 33, 44]
quicksort(arr, 0, len(arr) - 1)
print(arr)
```

bisect

from build-in module

```
def bisect_left(x, lo, hi, check): # check: key(a[mid]) < x
    while lo < hi:
        mid = (lo + hi) // 2
        if check(mid, x):
            lo = mid + 1
        else:
            hi = mid
    return lo

def bisect_right(x, lo, hi, check): # check: x < key(a[mid])
    while lo < hi:
        mid = (lo + hi) // 2
        if check(x, mid):
            hi = mid
        else:
            lo = mid + 1
    return lo
```

STRING

KMP

```python
""""
compute_lps 函数用于计算模式字符串的LPS表。LPS表是一个数组，
其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。
该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。
"""
def compute_lps(pattern):
    """
    计算pattern字符串的最长前缀后缀（Longest Proper Prefix which is also Suffix）表
    :param pattern: 模式字符串
    :return: lps表
    """

    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

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return 0
    lps = compute_lps(pattern)
    matches = []

    # 在 text 中查找 pattern
    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


text = "ABABABABCABABABABCABABABABC"
pattern = "ABABCABAB"
```

```
index = kmp_search(text, pattern)
print("pos matched: ", index)
# pos matched:  [4, 13]
```

DATA STUCTURE

Stack

{[()]} match

...

shutting yard

```
n=int(input())
value={'(':1,'+':2,'-':2,'*':3,'/':3}
for _ in range(n):
    put=input()
    stack=[]
    out=[]
    number=''
    for s in put:
        if s.isnumeric() or s=='.':
            number+=s
        else:
            if number:
                num=float(number)
                out.append(int(num) if num.is_integer() else num)
                number=''
            if s=='(':
                stack.append(s)
            elif s==')':
                while stack and stack[-1]!='(':
                    out.append(stack.pop())
                stack.pop()
            else:
                while stack and value[stack[-1]]>=value[s]:
                    out.append(stack.pop())
                stack.append(s)
    if number:
        num = float(number)
        out.append(int(num) if num.is_integer() else num)
    while stack:
        out.append(stack.pop())
    print(*out,sep=' ')
```

LinkedList

```
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, value):
        new_node = Node(value)
```

```
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def delete(self, value):
        if self.head is None:
            return
        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next

class Node:
    def __init__(self, data):
        self.data = data  # 节点数据
        self.next = None  # 指向下一个节点
        self.prev = None  # 指向前一个节点
class DoublyLinkedList:
    def __init__(self):
        self.head = None  # 链表头部
        self.tail = None  # 链表尾部
    def append(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
    def prepend(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
    def delete(self, node):
        if not self.head:  # 链表为空
            return
        if node == self.head:  # 删除头部节点
            self.head = node.next
            if self.head:  # 如果链表非空
                self.head.prev = None
        elif node == self.tail:  # 删除尾部节点
            self.tail = node.prev
```

```
            if self.tail:  # 如果链表非空
                self.tail.next = None
        else:  # 删除中间节点
            node.prev.next = node.next
            node.next.prev = node.prev
        node = None  # 删除节点

```

Fast-Slow Pointer

```
def find_middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

TREE

Binary Tree

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

preorder traversal

```
def preorder_traversal(root):
    if root:
        print(root.val)
        preorder_traversal(root.left)
        preorder_traversal(root.right)
```

inorder traversal

```
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val)
        inorder_traversal(root.right)
```

postorder traversal

```python
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.val)
```

level order traversal

```python
from collections import deque

def level_order_traversal(root):
    if not root:
        return []
    queue = deque([root])
    result = []
    while queue:
        level_size = len(queue)
        level = []
        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

color mark

similar to recursion dfs

```python
from collections import deque

def level_order_traversal(root):
    if not root:
        return []
    queue = deque([(root, "white")])
    result = []
    while queue:
        node, color = queue.popleft()
        if color == "white":
            result.append(node.val)
            queue.append((node.left, "gray"))
            queue.append((node.right, "gray"))
        else:
            result.append(node.val)
    return result
```

AST

Union Find

```
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))  # 初始化为自己是自己的父节点
        self.rank = [0] * size           # 用于按秩合并

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # 已经在一个集合中

        # 按秩合并
        if self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        elif self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1

        return True
```

Trie

```
class Node:
    def __init__(self, val=None):
        self.val = val
        self.children = {}
        self.is_end = False


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

    def insert(self, text):
        node = self.root
        has_prefix = False
        for word in text:
            if word not in node.children:
                node.children[word] = Node(word)
            node = node.children[word]
            if node.is_end:
                has_prefix = True
```

```python
        node.is_end = True
        return has_prefix
```

Huffman Tree

```python
import heapq

def huffman(n, weights):
    if n == 1:
        return weights[0]
    heapq.heapify(weights)

    total_cost = 0
    while len(weights) > 1:
        w1 = heapq.heappop(weights)
        w2 = heapq.heappop(weights)
        combined_weight = w1 + w2
        total_cost += combined_weight
        heapq.heappush(weights, combined_weight)
    return total_cost
```

GRAPH

```python
class Vertex:
    def __init__(self, key):
        self.key = key
        self.neighbors = []   # [key]
        # self.neighbors = []   # [(key, weight)]

class Graph:
    def __init__(self):
        self.vertices = {}   # {key: vertex}
```

bfs

dfs

topological sort

```python
def topological_sort(graph: Graph):
    in_degree = defaultdict(int)
    for u in graph.vertices.values():
        for v_key in u.neighbors:
            in_degree[v_key] += 1

    queue = deque()
    topo_order = []

    for u in graph.vertices.values():
        if in_degree[u.key] == 0:
            queue.append(u.key)

```

```python
    while queue:
        u_key = queue.popleft()
        topo_order.append(u_key)
        for v_key in graph.vertices[u_key].neighbors:
            in_degree[v_key] -= 1
            if in_degree[v_key] == 0:
                queue.append(v_key)
    if len(topo_order) != len(graph.vertices):
        return
    return topo_order
```

Shortest Path

dijkstra

```python
def dijkstra(graph: Graph, start: Vertex):
    path = {key: {'distance': float("inf"), 'path': []} for key in graph.vertices}
    path[start.key]['distance'] = 0
    heap = [(0, start.key)]
    while heap:
        current_distance, current_vertex_key = heappop(heap)

        for neighbor_key, weight in graph.vertices[current_vertex_key].neighbors:
            new_distance = current_distance + weight
            if new_distance < path[neighbor_key]['distance']:
                path[neighbor_key]['distance'] = new_distance
                path[neighbor_key]['path'] = path[current_vertex_key]['path'] + [(current_vertex_key, weight)]
                heappush(heap, (new_distance, neighbor_key))
    return path
```

*A-star

```python

```

bellman-ford

```python
def bellman_ford(graph: Graph, start: Vertex):
    distances = {key: float('inf') for key in graph.vertices}
    distances[start.key] = 0

    for _ in range(len(graph.vertices) - 1):
        for vertex in graph.vertices.values():
            for neighbor_key, weight in vertex.neighbors:
                if distances[vertex.key] + weight < distances[neighbor_key]:
                    distances[neighbor_key] = distances[vertex.key] + weight

```

```python
    for vertex in graph.vertices.values():
        for neighbor_key, weight in vertex.neighbors:
            if distances[vertex.key] + weight < distances[neighbor_key]:
                return
    return distances
```

*SPFA

SPFA IS DEAD

use queue, same to bellman-ford

floyd-warshall

```python
def floyd(graph: Graph):
    vertices = graph.vertices.values()
    dist = {v.key: {u.key: float('inf') for u in vertices} for v in vertices}

    for v in graph.vertices.values():
        dist[v.key][v.key] = 0
        for u in v.neighbors:
            dist[v.key][u[0]] = u[1]

    for k in vertices:
        for i in vertices:
            for j in vertices:
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

*Johnson's algorithm

Use potential-like method to make weights non-negative. $O(V(V+E)logV)$

```python
from bellman_ford import Graph, Vertex, bellman_ford
from dijkstra import dijkstra

def johnson(graph: Graph):
    virtual_vertex = Vertex(-1)
    for vertex in graph.vertices.values():
        virtual_vertex.neighbors.append([vertex.key, 0])
    graph.vertices[-1] = virtual_vertex
    h = bellman_ford(graph, virtual_vertex)
    if h is None:
        return
    for vertex in graph.vertices.values():
        for neighbor in vertex.neighbors:
            neighbor[1] += h[vertex.key] - h[neighbor[0]]
    full_distances = {}
    for v in graph.vertices.values():
```

```
        if v.key == -1:
            continue
        distances = dijkstra(graph, v)
        adjusted = {}
        for u, d in distances.items():
            d['distance'] = d['distance'] + h[v.key] - h[u]
            adjusted[u] = d
        full_distances[v.key] = adjusted
    return full_distances

```

MST

Prim

```
def prim(graph, start):
    visited = set() # {key}
    heap = [(0, None, start.key)]
    mst = [] # [(from, to, weight)]
    total_weight = 0

    while heap:
        weight, u_key, v_key = heappop(heap)
        if v_key in visited:
            continue
        visited.add(v_key)
        mst.append((u_key, v_key, weight))
        total_weight += weight

        v = graph.vertices[v_key]
        for neighbor_key, weight in v.neighbors:
            if neighbor_key not in visited:
                heappush(heap, (weight, v_key, neighbor_key))

    return mst, total_weight
```

Kruskal

Minimum Spanning Forest

```
from ..tree.union_find import UnionFind

def kruskal(graph):
    n = len(graph.vertices)
    edges = []

    for v in graph.vertices.values():
        for neighbor_key, weight in v.neighbors:
            edges.append((weight, v.key, neighbor_key))

```

```
    edges.sort()

    union_find = UnionFind(n)
    mst = [] # [(from, to, weight)]
    total_weight = 0

    for weight, u_key, v_key in edges:
        if union_find.find(u_key) != union_find.find(v_key):
            union_find.union(u_key, v_key)
            mst.append((u_key, v_key, weight))
            total_weight += weight

    return mst, total_weight
```