Summary of this semester's DSA

How to use set, dict, defaultdict, heap, deque, itertools etc.

1. set

set(): Creates an empty set.
set(iterable): Creates a set from an iterable (e.g., list, tuple, string).
add(element): Adds an element to the set.
update(iterable): Adds multiple elements from an iterable to the set.
remove(element): Removes an element from the set. Raises an error if the element is not found.
discard(element): Removes an element from the set if it exists. Does not raise an error if the element is not found.
pop(): Removes and returns an arbitrary element from the set. Raises an error if the set is empty.
union(set1, set2, ...): Returns a new set containing all unique elements from all given sets.

```
u = union(v, w) # or v.union(w)
```

intersection(set1, set2, ...): Returns a new set containing common elements among all given sets.
difference(set1, set2): Returns a new set with elements present in set1 but not in set2.
symmetric_difference(set1, set2): Returns a new set with elements present in either set1 or set2, but not in both.
issubset(set): Checks if the set is a subset of another set.(u.issubset(v) means whether u is a subset of v)
issuperset(set): Checks if the set is a superset of another set.
isdisjoint(set): Checks if the set has no common elements with another set.
len(set): Returns the number of elements in the set.
element in set: Checks if an element is present in the set.
set.copy(): Returns a shallow copy of the set.

2. dict

dict(): Creates an empty dictionary.
dict(mapping): Creates a dictionary from a mapping (e.g., another dictionary).
dict(**kwargs): Creates a dictionary from keyword arguments.
dict[key]: Retrieves the value associated with the specified key.
dict[key] = value: Associates the value with the specified key.
del dict[key]: Removes the key-value pair with the specified key from the dictionary.
dict.get(key, default): Retrieves the value associated with the specified key. Returns the default value if the key is not found.
dict.setdefault(key, default): Retrieves the value associated with the specified key. If the key is not found, sets the default value and returns it.
dict.pop(key, default): Removes and returns the value associated with the specified key. Returns the default value if the key is not found.
dict.update(mapping): Updates the dictionary with key-value pairs from another mapping (e.g., another dictionary).
len(dict): Returns the number of key-value pairs in the dictionary.
key in dict: Checks if a key is present in the dictionary.
dict.keys(): Returns a view object containing the keys of the dictionary.
dict.values(): Returns a view object containing the values of the dictionary.
dict.items(): Returns a view object containing the key-value pairs of the dictionary.
for key in dict: Iterates over the keys of the dictionary.
for key, value in dict.items(): Iterates over the key-value pairs of the dictionary.

3. defaultdict

```
from collections import defaultdict
```

defaultdict(default_factory): Creates a new defaultdict with a default factory function. The default_factory can be any callable object, such as a function or a class.
defaultdict[key]: Accesses the value associated with the specified key. If the key is not present, a new entry is created with the default value provided by the default_factory.
defaultdict[key] = value: Associates the value with the specified key. If the key is not present, a new entry is created with the default value provided by the default_factory.
defaultdict.default_factory: Returns the default factory function used by the defaultdict.

```
defaultdict(int) # has default value 0
defaultdict(list) # has default dict 0
defaultdict(lambda: 'Unknown') # has default value 'Unknown'
```

4. heap

```
import heapq
```

heapq.heapify(x): Converts a list x into a heap in-place. The list x is rearranged to satisfy the heap property.
heapq.heappush(heap, item): Pushes the item onto the heap while maintaining the heap property.
heapq.heappop(heap): Removes and returns the smallest element from the heap while maintaining the heap property.
heapq.heappushpop(heap, item): Pushes the item onto the heap, and then removes and returns the smallest element from the heap. This operation is more efficient than performing separate heappush

and heappop operations.
heapq.heapreplace(heap, item): Removes and returns the smallest element from the heap, and then pushes the item onto the heap. This operation is more efficient than performing separate heappop and heappush operations.
heapq.nsmallest(n, iterable): Returns the n smallest elements from the iterable as a list, ordered from smallest to largest.
heapq.nlargest(n, iterable): Returns the n largest elements from the iterable as a list, ordered from largest to smallest.
heapq.merge(*iterables): Merges multiple sorted inputs into a single sorted iterator.
heapq.heapreplace(heap, item): Replaces the smallest element in the heap with the item, and returns the original smallest element.

5. deque

deque.append(x): Adds element x to the right (back) end of the deque.
deque.appendleft(x): Adds element x to the left (front) end of the deque.
deque.extend(iterable): Extends the deque by appending elements from the iterable to the right end.
deque.extendleft(iterable): Extends the deque by appending elements from the iterable to the left end. The elements are added in reverse order.
deque.pop(): Removes and returns the rightmost (last) element from the deque.
deque.popleft(): Removes and returns the leftmost (first) element from the deque.
deque.remove(value): Removes the first occurrence of value from the deque.
deque.clear(): Removes all elements from the deque.
deque[index]: Accesses the element at the specified index in the deque.
deque[-1]: Accesses the rightmost (last) element in the deque.
deque[0]: Accesses the leftmost (first) element in the deque.
deque.count(x): Returns the number of occurrences of element x in the deque.
deque.index(x[, start[, end]]): Returns the index of the first occurrence of element x in the deque within the specified range.
len(deque): Returns the number of elements in the deque.
deque.rotate(n): Rotates the deque n steps to the right (positive n) or left (negative n).
deque.reverse(): Reverses the order of elements in the deque.

6. itertools

Infinite Iterators:

itertools.count(start=0, step=1): Generates an infinite arithmetic progression starting from start with a step of step.
itertools.cycle(iterable): Creates an infinite iterator that cycles through the elements of iterable.
itertools.repeat(object[, times]): Repeats object infinitely or times times.

Iterators Terminating on the Shortest Input:

itertools.chain(*iterables): Combines multiple iterables into a single iterator that sequentially yields elements from each iterable.
itertools.compress(data, selectors): Returns an iterator that filters elements from data based on the truth values of corresponding elements in selectors.
itertools.dropwhile(predicate, iterable): Skips elements from the iterable as long as the predicate is true and then returns the remaining elements.
itertools.takewhile(predicate, iterable): Returns elements from the iterable as long as the predicate is true and then stops.

Combinatoric Generators:

itertools.product(*iterables[, repeat=1]): Generates the Cartesian product of the input iterables.
itertools.permutations(iterable, r=None): Generates all possible permutations of the elements in the iterable.
itertools.combinations(iterable, r): Generates all possible combinations of length r from the elements in the iterable.
itertools.combinations_with_replacement(iterable, r): Generates all possible combinations of length r from the elements in the iterable, allowing repeated elements.

Other Functions:

itertools.islice(iterable, start, stop[, step]): Returns an iterator that generates selected elements from the iterable by index range.
itertools.groupby(iterable, key=None): Groups consecutive elements of the iterable by a common key.
itertools.chain.from_iterable(iterable): Creates an iterator that flattens nested iterables into a single sequence.
itertools.starmap(function, iterable): Applies function to each element of the iterable as arguments unpacked from tuples or lists.

7. collections

collections.Counter([iterable-or-mapping]): Creates a dictionary subclass for counting hashable objects. It provides a convenient way to count elements in an iterable or create a histogram of items.

```
from collections import Counter

c = Counter(['a', 'b', 'a', 'c', 'b', 'a'])
print(c)  # Output: Counter({'a': 3, 'b': 2, 'c': 1})
```

Example:

```python
from collections import Counter
from itertools import product
n = int(input())
A = []
B = []
C = []
D = []
for i in range(n) :
    u = list(map(int, input().split()))
    A.append(u[0])
    B.append(u[1])
    C.append(u[2])
    D.append(u[3])
A = sorted(A)
B = sorted(B)
C = sorted(C)
D = sorted(D)
res = 0
AB = Counter(map(lambda x : x[0] + x[1], product(A, B)))
for c in C :
    for d in D :
        res += AB.get(-c - d, 0)
print(res)
```

Frequently used tricks

0. Increase the Stack Limit

```python
import sys

new_limit = 5000  # Set the desired stack size limit
sys.setrecursionlimit(new_limit)
```

1. Binary-indexed tree

```python
def lowbit (x) :
    return x & -x
class BIT:
    def __init__(self, n):
        self.sum = [0] * (n + 1) # the occupied index are 1 ~ n
    def add(self, m, x):
        tmp = m
        while tmp < len(self.sum):
            self.sum[tmp] += x
            tmp += lowbit(tmp)
    def getPrefixSum(self, m):
        res = 0
        tmp = m
        while tmp:
            res += self.sum[tmp]
            tmp = tmp - lowbit(tmp)
        return res
```

2. Segment Tree

```
class SegtreeNode:
    def __init__ (self, val = 0, left = None, right = None, lazy = 0):
        self.sum = val
        self.lazy = lazy
        self.left = left
        self.right = right
    def pushup(self):
        self.sum = (0 if not self.left else self.left.sum) + (0 if not self.right else self.ri
    def pushdown(self, l, r):
        m = (l + r) // 2
        tmp = self.lazy
        self.lazy = 0
        if self.left:
            self.left.lazy += tmp
            self.left.sum += (m - l + 1) * tmp
        if self.right:
            self.right.lazy += tmp
            self.right.sum += (r - m) * tmp

    def update(self, l, r, x, y, k):
        if l > y or r < x:
            return
        if x <= l and r <= y:
            self.lazy += k
            self.sum += (r - l + 1) * k
            return
        m = (l + r) // 2
        self.pushdown(l, r)
        if self.left:
            self.left.update(l, m, x, y, k)
        if self.right:
            self.right.update(m + 1, r, x, y, k)
        self.pushup()

    def query(self, l, r, x, y):
        if l > y or r < x:
            return 0
        if x <= l and r <= y:
            return self.sum
        m = (l + r) // 2
        self.pushdown(l, r)
        return (0 if not self.left else self.left.query(l, m, x, y)) + (0 if not self.right els

def buildtree(l, r, s):
    if l > r:
        return None
    if l == r :
        return SegtreeNode(s[l])
    m = (l + r) // 2
    p = SegtreeNode(0, buildtree(l, m, s), buildtree(m + 1, r, s))
    p.pushup()
    return p
```

3. Binary Search

```
import bisect

a = [1, 3, 5, 7, 9]

# Example 1: Using bisect_left()
x = 4
index_left = bisect.bisect_left(a, x)
print(index_left)  # Output: 2

# Example 2: Using bisect_right()
x = 4
index_right = bisect.bisect_right(a, x)
print(index_right)  # Output: 2

# Example 3: Using insort_left()
x = 4
bisect.insort_left(a, x)
print(a)  # Output: [1, 3, 4, 5, 7, 9]

# Example 4: Using insort_right()
x = 4
bisect.insort_right(a, x)
print(a)  # Output: [1, 3, 4, 4, 5, 7, 9]
```

bisect.bisect_left(a, x, lo=0, hi=len(a)): Returns the index at which element x should be inserted into a sorted sequence a (in non-decreasing order). If x is already present, it returns the leftmost index before any existing occurrences.
bisect.bisect_right(a, x, lo=0, hi=len(a)): Returns the index at which element x should be inserted into a sorted sequence a (in non-decreasing order). If x is already present, it returns the rightmost index after all existing occurrences.
bisect.insort_left(a, x, lo=0, hi=len(a)): Inserts the element x into a sorted sequence a (in non-decreasing order) at the appropriate index, maintaining the sorted order. If x is already present, it is

inserted to the left of existing occurrences.
bisect.insort_right(a, x, lo=0, hi=len(a)): Inserts the element x into a sorted sequence a (in non-decreasing order) at the appropriate index, maintaining the sorted order. If x is already present, it is inserted to the right of existing occurrences.

4. Monotone Stack

单调栈模板，找到每个元素之后第一个比该元素大的元素。

```
n = int(input())
s = list(map(int, input().split()))
q = []
res = []
for i in range(n - 1, -1, -1):
    while q and s[i] >= s[q[-1]]:
        q.pop()
    res.append(-1 if not q else q[-1])
    q.append(i)
res.reverse()
for i in res:
    print(i + 1, end=" ")
```

5. Monotone Queue

滑动窗口最大值（窗口长为k）

```
n, k = map(int, input().split())
val = list(map(int, input().split()))
h = []
head = 0
for i in range(k) :
    while head < len(h) and val[h[len(h) - 1]] <= val[i] :
        h.pop()
    h.append(i)
print(val[h[head]], end = "")
for i in range(k, n) :
    while head < len(h) and val[h[len(h) - 1]] <= val[i] :
        h.pop()
    h.append(i)
    while head < len(h) and h[head] <= i - k :
        head += 1
    print(f" {val[h[head]]}", end = "")
```

6. AVL Tree

```
'''REMARK: The insertion methods is almost surely correct, but there could be mistakes in the d
INF = 10000000000

class Node :
    def __init__(self, val) : # making a new leaf node with value = val
        self.val = val
        self.left = None
        self.right = None
        self.height = 1
        self.size = 1
    def upd_height(self) :
        if not self.left and not self.right :
            self.height = 1
            self.size = 1
            return
        if not self.left :
            self.height = self.right.height + 1
            self.size = self.right.size + 1
            return
        if not self.right :
            self.height = self.left.height + 1
            self.size = self.left.size + 1
            return
        self.height = max(self.left.height, self.right.height) + 1
        self.size = self.left.size + self.right.size + 1
    def balanceness(self) :
        self.upd_height()
        u = 0
        v = 0
        if self.left != None :
            u = self.left.height
        if self.right != None :
            v = self.right.height
    #    print(u, v)
        return u - v
class AVL :
    def __init__(self) :
        self.root = None
        self.count = 0
        self.d = {}
    def _rotate_right(self, node) :
    #    print("rotate right")
        T1 = node.left.right
        tmp = node.left
```

```
        tmp.right = node
        tmp.right.left = T1
        tmp.right.upd_height()
        tmp.upd_height()
        return tmp
    def _rotate_left(self, node) :
    #    print("rotate left")
        T1 = node.right.left
        tmp = node.right
        tmp.left = node
        tmp.left.right = T1
        tmp.left.upd_height()
        tmp.upd_height()
        return tmp

    def _rebalance(self, node) :
    #    print("rebalance")
        if node.balanceness() >= 2 :
            if node.left.balanceness() == 1 :
    #            print("LL")
                node = self._rotate_right(node)
                return node
            else :
    #            print("LR")
                node.left = self._rotate_left(node.left)
                node = self._rotate_right(node)
                return node
        if node.balanceness() <= -2 :
            if node.right.balanceness() == -1 :
    #            print("RR")
                node = self._rotate_left(node)
                return node
            else :
    #            print("RL")
                node.right = self._rotate_right(node.right)
                node = self._rotate_left(node)
                return node
    #    print("nothing")
        return node


    def _insert(self, value, node) :
        if not node :
    #        print("added")
            return Node(value)
        if node.val < value :
    #        print("going right")
            node.right = self._insert(value, node.right)
        elif node.val > value :
    #        print("going left")
            node.left = self._insert(value, node.left)
        node.upd_height()
        node = self._rebalance(node)
        return node
    def insert(self, value) :
        self.count += 1
        if value not in self.d:
            self.d[value] = set()
        self.d[value].add(self.count)
        if not self.root :
            self.root = Node((value, self.count))
        else :
            self.root = self._insert((value, self.count), self.root)

    def delete(self, value) : # nothing will be done if value doesn't exist
        if value not in self.d or not self.d[value]:
            return
        cnt = self.d[value].pop()
        if not self.d[value]:
            del self.d[value]
        self.root = self._delete((value, cnt), self.root)
    def _delete(self, value, node) :
        if not node:
            return None
        if node.val < value :
            node.right = self._delete(value, node.right)
        elif node.val > value :
    #        print("going left")
            node.left = self._delete(value, node.left)
        else :
            if not node.left:
                tmp = node.right
                node = None
                return tmp
            if not node.right:
                tmp = node.left
                node = None
                return tmp

            u = self._get_min_val(node.right)
            node.val = u
            node.right = self._delete(u, node.right)
```

```
    if not node:
        return node

    node.upd_height()
    node = self._rebalance(node)
    return node
def _get_min_val(self, node):
    tmp = node
    while tmp.left :
        tmp = tmp.left
    return tmp.val

def traversal(self, mode, node) :
    if not node :
        return []
    if mode == "pre" :
        return [node.val] + self.traversal("pre", node.left) + self.traversal("pre", node.r
    if mode == "mid" :
        return self.traversal("mid", node.left) + [node.val] + self.traversal("mid", node.r
    if mode == "post" :
        return self.traversal("post", node.left) + self.traversal("post", node.right) + [no
def __str__(self) : # preorder traversal as default
    if not self.root :
        return ""
    return " ".join(str(item[0]) for item in self.traversal("mid", self.root))

def getkth(self, k):
    if k <= 0 or k > self.root.size :
        return None
    return self._getkth(k, self.root)[0]
def _getkth(self, k, node):
    lsize = 0 if not node.left else node.left.size
    if k <= lsize:
        return self._getkth(k, node.left)
    elif k == lsize + 1:
        return node.val
    else :
        return self._getkth(k - lsize - 1, node.right)

def getnext(self, x):
    u = self._getnext((x, INF), self.root)
    return None if not u else u[0]
def _getnext(self, x, node):
    if not node:
        return None
    if x < node.val:
        u = self._getnext(x, node.left)
        return node.val if not u else u
    else:
        return self._getnext(x, node.right)
def getprev(self, x):
    u = self._getprev((x, -INF), self.root)
    return None if not u else u[0]
def _getprev(self, x, node):
    if not node:
        return None
    if x > node.val:
        u = self._getprev(x, node.right)
        return node.val if not u else u
    else:
        return self._getprev(x, node.left)

def getrank(self, x):
    return self._getrank((x, -INF), self.root) + 1
def _getrank(self, x, node):
    if not node:
        return 0
    if x < node.val:
        return self._getrank(x, node.left)
    else :
        return (0 if not node.left else node.left.size) + 1 + self._getrank(x, node.right)

n = int(input())
T = AVL()
for i in range(n):
    # print(T)
    op, x = map(int, input().split())
    if op == 1:
        T.insert(x)
    if op == 2:
        T.delete(x)
    if op == 3:
        print(T.getrank(x))
    if op == 4:
        print(T.getkth(x))
    if op == 5:
        print(T.getprev(x))
    if op == 6:
        print(T.getnext(x))
```

7. Heap

```
class BinHeap : # the top of the heap is the smallest, occupies 1 ~ n
    def __init__(self) :
        self.item = [0]
        self.size = 0
    def percUp(self, i) :
        if i == 1 :
            return
        if self.item[i] < self.item[i // 2] :
            self.item[i], self.item[i // 2] = self.item[i // 2], self.item[i]
            self.percUp(i // 2)
    def insert(self, x) :
        self.item.append(x)
        self.size += 1
        self.percUp(self.size)
    def percDown(self, i) :
        if i * 2 > self.size :
            return
        u = i * 2
        if 2 * i + 1 <= self.size :
            if self.item[2 * i + 1] < self.item[2 * i] :
                u = 2 * i + 1
        if self.item[u] < self.item[i] :
            self.item[i], self.item[u] = self.item[u], self.item[i]
            self.percDown(u)
    def delTop(self) :
        if self.size == 0 :
            return None
        res = self.item[1]
        self.item[1] = self.item[self.size]
        self.item.pop()
        self.size -= 1
        self.percDown(1)
        return res
    def heapify(self, items) :
        self.item.extend(items)
        self.size = len(self.item) - 1
        i = self.size // 2
        while i >= 1 :
            self.percDown(i)
            i -= 1
```

8. ST Table

```
class STtable:
    def __init__ (self, s): # s[0 ~ n - 1] -> 1 ~ n
        n = len(s)
        self.logtable = [0]
        for i in range(1, n + 1):
            self.logtable.append(0)
            if (1 << (self.logtable[i - 1] + 1)) <= i:
                self.logtable[i] = self.logtable[i - 1] + 1
            else:
                self.logtable[i] = self.logtable[i - 1]
        self.table = []
        m = self.logtable[n]
        for j in range(m + 1):
            self.table.append([0])
        for i in range(n):
            self.table[0].append(s[i])
        for j in range(1, m + 1):
            for i in range(1, n - (1 << j) + 2):
                self.table[j].append(max(self.table[j - 1][i], self.table[j - 1][i + (1 << (j -
    def query(self, l, r):
        u = self.logtable[r - l + 1]
        return max(self.table[u][l], self.table[u][r - (1 << u) + 1])
```

Graph

1. Disjoint Set

```
class DisjointSet :
    def __init__(self, items) :
        self.sizes = dict(zip(items, [1] * len(items)))
        self.rep = dict(zip(items, items))
        self.length = len(items)
    def getrep(self, i) :
        if self.rep[i] == i :
            return i
        res = self.getrep(self.rep[i])
        self.rep[i] = res
        return self.rep[i]
    def check_if_same(self, i, j) :
        return (self.getrep(i) == self.getrep(j))
    def merge(self, i, j) :
        x = self.getrep(i)
        y = self.getrep(j)
        if x == y :
            return
        u = self.sizes[x]
        v = self.sizes[y]
        if u < v :
            self.rep[x] = y
            self.sizes[y] = u + v
        else :
            self.rep[y] = x
            self.sizes[x] = u + v
```

2.1. Shortest Path (Dijkstra)

```
import heapq

class Graph :
    def __init__(self) :
        self.edges = {}
    def addv(self, name) :
        self.edges[name] = []
    def addedge(self, u, v, w) :
        if u in self.edges and v in self.edges:
            self.edges[u].append((v, w))
    def dijkstra(self, start, end) :
        if start not in self.edges or end not in self.edges :
            return -1
        dist = {}
        for i in self.edges :
            dist[i] = -1
        vis = set()
        q = [(0, start)]
        heapq.heapify(q)
        while q :
            d, u = heapq.heappop(q)
            #print(d, u)
            if u == end :
                return d
            if u in vis :
                continue
            dist[u] = d
            vis.add(u)
            for e in self.edges[u] :
                v, w = e
                if dist[u] + w < dist[v] or dist[v] == -1 :
                    dist[v] = dist[u] + w
                heapq.heappush(q, (dist[v], v))
        return -1
```

2.2. Shortest Path (Floyd)

Pseudocode [edit]

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

2.3. Shortest Path (Bellman-Ford)

```
function BellmanFord(list vertices, list edges, vertex source) is

    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance  := list of size n
    predecessor  := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do
        // Initialize the distance to all vertices to infinity
        distance[v]  := inf
        // And having a null predecessor
        predecessor[v]  := null

    // The distance from the source to itself is, of course, zero
    distance[source]  := 0

    // Step 2: relax edges repeatedly
    repeat |V|-1 times:
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v]  := distance[u] + w
                predecessor[v]  := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            predecessor[v]  := u
            // A negative cycle exists; find a vertex on the cycle
            visited  := list of size n initialized with false
            visited[v]  := true
            while not visited[u] do
                visited[u]  := true
                u  := predecessor[u]
            // u is a vertex in a negative cycle, find the cycle itself
            ncycle  := [u]
            v  := predecessor[u]
            while v  != u do
                ncycle  := concatenate([v], ncycle)
                v  := predecessor[v]
            error "Graph contains a negative-weight cycle", ncycle
    return distance, predecessor
```

```cpp
// SPFA
int n,m,s,tot,dis[MAXN],vis[MAXN];
struct edge
{
	int v,w;
	edge *next;
}pool[MAXM],*h[MAXN];
void addedge(int u,int v,int w)
{
	edge *p=&pool[tot++];
	p->v=v;p->w=w;p->next=h[u];h[u]=p;
}
void spfa()
{
	queue<int> q;
	for(int i=1;i<=n;i++)dis[i]=2147483647;
	dis[s]=0;vis[s]=1;q.push(s);
	while(!q.empty())
	{
		int k=q.front();q.pop();vis[k]=0;
		for(edge *p=h[k];p;p=p->next)
		{
			if(dis[p->v]>dis[k]+p->w)
			{
				dis[p->v]=dis[k]+p->w;
				if(!vis[p->v])
				{
					vis[p->v]=1;
					q.push(p->v);
				}
			}
		}
	}
}
```

2.4 Shortest Path (Johnson)

Algorithm description [edit]

Johnson's algorithm consists of the following steps:[1][2]

1. First, a new node q is added to the graph, connected by zero-weight edges to each of the other nodes.
2. Second, the Bellman–Ford algorithm is used, starting from the new vertex q, to find for each vertex v the minimum weight $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from u to v, having length $w(u,v)$, is given the new length $w(u,v) + h(u) - h(v)$.
4. Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other verte in the reweighted graph. The distance in the original graph is then computed for each distance $D(u , v)$, by adding $h(v) - h(u)$ to the distance returned by Dijkstra's algorithm.

3.1. Minimum Spanning Tree (Kruskal)

```
class DisjointSet :
    def __init__(self, items) :
        self.rep = dict(zip(items, items))
        self.siz = dict(zip(items, [1] * len(items)))
    def getrep(self, item) :
        if self.rep[item] == item :
            return item
        else :
            self.rep[item] = self.getrep(self.rep[item])
            return self.rep[item]
    def merge(self, u, v) :
        fu = self.getrep(u)
        fv = self.getrep(v)
        su = self.siz[fu]
        sv = self.siz[fv]
        if fu == fv :
            return True
        else :
            if su > sv :
                self.rep[fv], self.siz[fu] = fu, su + sv
            else :
                self.rep[fu], self.siz[fv] = fv, su + sv
            return False
E = sorted(E) # the set of edges sorted in the increasing order of weights
D = DisjointSet(list(i for i in range(N)))
ans = 0
for e in E :
    flag = D.merge(e[1], e[2])
    if not flag :
        ans += e[0]
print(ans)
```

3.2. Minimum Spanning Tree (Prim)

```
import heapq

class Graph :
    def __init__(self) :
        self.edges = {}
    def addv(self, name) :
        self.edges[name] = []
    def addedge(self, u, v, w) :
        self.edges[u].append((v, w))
    def Prim(self, start) :
        res = 0
        q = [(0, start)]
        heapq.heapify(q)
        vis = set()
        while q :
            d, u = heapq.heappop(q)
            if u in vis :
                continue
            res += d
            for e in self.edges[u] :
                v, w = e
                heapq.heappush(q, (w, v))
            vis.add(u)
            #print(q, d, u)
        return res
```

4 Strong Connected Component (Kosaraju)

If strong components are to be represented by appointing a separate root vertex for each component, and assigning to each vertex the root vertex of its component, then Kosaraju's algorithm can be stated as follows.

1. For each vertex u of the graph, mark u as unvisited. Let L be empty.
2. For each vertex u of the graph do `Visit(u)`, where `Visit(u)` is the recursive subroutine:
   If u is unvisited then:
   1. Mark u as visited.
   2. For each out-neighbour v of u, do `Visit(v)`.
   3. Prepend u to L.

   Otherwise do nothing.
3. For each element u of L in order, do `Assign(u, u)` where `Assign(u, root)` is the recursive subroutine:
   If u has not been assigned to a component then:
   1. Assign u as belonging to the component whose root is *root*.
   2. For each in-neighbour v of u, do `Assign(v, root)`.

   Otherwise do nothing.

```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(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)
```

```
"""
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]

"""
```

5. Warnsdorff's algorithm

```python
class Graph:
    def __init__(self) :
        self.vertices = dict()
    def addVertices(self, names) :
        for name in names :
            if name not in self.vertices :
                self.vertices[name] = []
    def addedge(self, u, v) : # directed edge
        self.vertices[u].append(v)
    def askHamiltonPath(self, u, target_level, level = 1, vis = None) :
        #print(u, target_level, level)
        if vis is None :
            vis = set([u])
        if level == target_level :
            return True
        ss = self.vertices[u]
        ss = sorted(ss, key=lambda x : sum((0 if i in vis else 1) for i in self.vertices[x]))
        for v in ss :
            if v not in vis :
                vis.add(v)
                if self.askHamiltonPath(v, target_level, level + 1, vis=vis) :
                    return True
                vis.remove(v)
        return False
```

Trees

1. Shunting Yard

```
precedence = {"+" : 10, "-" : 10, "*" : 20, "/" : 20, "(" : 0}
ToNum = lambda x : str(x)
Calc = {"+" : (lambda x, y : x + y),
        "-" : (lambda x, y : x - y),
        "*" : (lambda x, y : x * y),
        "/" : (lambda x, y : x / y)}
NumberOfOpr = {i : 2 for i in "+-*/"}
Val = "1234567890."
Opr = "+-*/"
class SyntaxTree :
    def __init__ (self, val, left = None, right = None):
        self.val = val # in the form of a string
        self.left = left
        self.right = right
    def ToExpr(self, mode = "infix"):
        lexpr = "" if not self.left else self.left.ToExpr(mode)
        rexpr = "" if not self.right else self.right.ToExpr(mode)
        if mode == "prefix" :
            return self.val + ("" if not self.left else (" " + lexpr)) + ("" if not self.right
        elif mode == "postfix" :
            return ("" if not self.left else (lexpr + " ")) + ("" if not self.right else (rexpr
        else :
            if self.right and (self.right.val in precedence) and precedence[self.right.val] <=
                rexpr = ("(" + rexpr + ")")
            if self.left and (self.left.val in precedence) and precedence[self.left.val] < prec
                lexpr = ("(" + lexpr + ")")
            return ("" if not self.left else lexpr) + self.val + ("" if not self.right else rex
    def Eval(self):
        if self.val not in precedence:
            return ToNum(self.val)
        else:
            if not self.left:
                return Calc[self.val](self.right.Eval())
            else:
                return Calc[self.val](self.left.Eval(), self.right.Eval())
    def __str__(self):
        return self.ToExpr("prefix")

def InfixToSyntax(s):
    output_stack = []
    opr_stack = []
    tmp = ""
    for char in s:
        #print(tmp, opr_stack)
        #print(" ".join(str(i) for i in output_stack))
        if char in Val:
            tmp += char
        else:
            if tmp:
                output_stack.append(SyntaxTree(tmp))
                tmp = ""
            if char in Opr:
                while opr_stack and precedence[opr_stack[-1]] >= precedence[char]:
                    opr = opr_stack.pop()
                    if NumberOfOpr[opr] == 1 :
                        x = output_stack.pop()
                        output_stack.append(SyntaxTree(opr, None, x))
                    else:
                        y = output_stack.pop()
                        x = output_stack.pop()
                        output_stack.append(SyntaxTree(opr, x, y))
                opr_stack.append(char)
            elif char == "(":
                opr_stack.append(char)
            elif char == ")":
                while opr_stack and opr_stack[-1] != "(":
                    opr = opr_stack.pop()
                    if NumberOfOpr[opr] == 1 :
                        x = output_stack.pop()
                        output_stack.append(SyntaxTree(opr, None, x))
                    else:
                        y = output_stack.pop()
                        x = output_stack.pop()
                        output_stack.append(SyntaxTree(opr, x, y))
                opr_stack.pop()
    if tmp:
        output_stack.append(SyntaxTree(tmp))
    while opr_stack:
        opr = opr_stack.pop()
        if NumberOfOpr[opr] == 1 :
            x = output_stack.pop()
            output_stack.append(SyntaxTree(opr, None, x))
        else:
            y = output_stack.pop()
            x = output_stack.pop()
            output_stack.append(SyntaxTree(opr, x, y))
    return output_stack.pop()
```

```python
# 布尔表达式 AC代码
# 23n2300011119(武)
def ShuntingYard(l:list):
    stack,output=[],[]
    for i in l:
        if i==" ":continue
        if i in 'VF':output.append(i)
        elif i=='(':stack.append(i)
        elif i in '&|!':
            while True:
                if i=='!':break
                elif not stack:break
                elif stack[-1]=="(":
                    break
                else:output.append(stack.pop())
            stack.append(i)
        elif i==')':
            while stack[-1]!='(':
                output.append(stack.pop())
            stack.pop()
    if stack:output.extend(reversed(stack))
    return output

def Bool_shift(a):
    if a=='V':return True
    elif a=='F':return False
    elif a==True:return 'V'
    elif a==False:return 'F'

def cal(a,operate,b=None):
    if operate=="&":return Bool_shift(Bool_shift(a) and Bool_shift(b))
    if operate=="|":return Bool_shift(Bool_shift(a) or Bool_shift(b))
    if operate=="!":return Bool_shift(not Bool_shift(a))

def post_cal(l:list):
    stack=[]
    for i in l:
        if i in 'VF':stack.append(i)
        elif i in "&|!":
            if i=="!":
                stack.append(cal(stack.pop(),'!'))
            else:
                a,b=stack.pop(),stack.pop()
                stack.append(cal(a,i,b))
    return stack[0]

while True:
    try:print(post_cal(ShuntingYard(list(input()))))
    except EOFError:break
```

2. LCA

也可以倍增求

```cpp
#include <iostream>
#include <cstdio>
#define MAXN 500005
using namespace std;
int n,m,root,MOD,tot,tot1,tot2,init_val[MAXN],depth[MAXN];
int fa[MAXN],subtree_siz[MAXN],heavy_son[MAXN];
int dfs_ord[MAXN],hchain_top[MAXN],dfs_to_ori[MAXN];
int read(){
	int x=0;char c=getchar();
	while(c<'0'||c>'9')c=getchar();
	while('0'<=c&&c<='9')x=x*10+c-48,c=getchar();
	return x;
}
struct edge{
	int v;
	edge *next;
}pool[MAXN<<1],*h[MAXN];
void addedge(int u,int v){
	edge *p=&pool[tot++];
	p->v=v;p->next=h[u];h[u]=p;
}
void buildtree(){
	n=read();m=read();root=read();
	for(int i=1,j,k;i<n;i++)
		j=read(),k=read(),addedge(j,k),addedge(k,j);
}
void dfs1(int u,int father){
	depth[u]=depth[father]+1;fa[u]=father;subtree_siz[u]=1;heavy_son[u]=0;
	for(edge *p=h[u];p;p=p->next){
		if(p->v==father)continue;
		dfs1(p->v,u);subtree_siz[u]+=subtree_siz[p->v];
		if(subtree_siz[p->v]>=subtree_siz[heavy_son[u]])heavy_son[u]=p->v;
	}
}
void dfs2(int u,int father,int hchain_top_ori){
	dfs_ord[u]=++tot1;dfs_to_ori[tot1]=u;hchain_top[u]=hchain_top_ori;
	if(heavy_son[u]){
		dfs2(heavy_son[u],u,hchain_top_ori);
	}
	for(edge *p=h[u];p;p=p->next){
		if(p->v==father)continue;
		if(p->v==heavy_son[u])continue;
		dfs2(p->v,u,p->v);
	}
}
int getlca(int u,int v){
//	printf("getting LCA :%d %d\n",u,v);
	while(hchain_top[u]!=hchain_top[v]){
//		printf("getting LCA :%d %d\n",u,v);
		if(depth[hchain_top[u]]>depth[hchain_top[v]])u=fa[hchain_top[u]];
		else v=fa[hchain_top[v]];
	}
	return (depth[u]<depth[v])?u:v;
}
void work(){
//	for(int i=1;i<=n;i++){
//		printf("node %d :\n fa:%d depth:%d size:%d hchain_top:%d\n",i,fa[i],depth[i],su
//	}
	for(int i=1,x,y;i<=m;i++){
		x=read();y=read();
		printf("%d\n",getlca(x,y));
	}
}
int main(){
	buildtree();
	dfs1(root,0);
	dfs2(root,0,root);
	work();
	return 0;
}
```

3. Building Trees

前中序遍历建树

```python
class node :
    def __init__ (self, left, right, val) :
        self.left = left
        self.right = right
        self.val = val

def getIndex(c, s) :
    for i in range(len(s)) :
        if c == s[i] :
            return i
    return -1

def ToPostString(s) :
    if s == None :
        return ""
    return ToPostString(s.left) + ToPostString(s.right) + s.val

def BuildTree_pre_in(sp, si) :
    if sp == "" :
        return None
    l = getIndex(sp[0], si)
#    print(sp, si, l, sep = " ")
    return node(BuildTree_pre_in(sp[1 : l + 1], si[0 : l]), BuildTree_pre_in(sp[l + 1 : len(sp)

while True :
    try :
        s1 = input()
        s2 = input()
        print(ToPostString(BuildTree_pre_in(s1, s2)))
    except :
        break
```

Strings

1. 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
    length = 0
    for i in range(1, m):
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]    # 跳过前面已经比较过的部分
        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 = []

    j = 0  # j是pattern的索引
    for i in range(n):  # i是text的索引
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)
            j = lps[j - 1]
    return matches
```

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

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
int next[1000005],n,m;
char s1[1000005],s2[1000005];
int main()
{
        scanf("%s",s1);scanf("%s",s2);
        n=strlen(s1);m=strlen(s2);
        for(int i=1;i<m;i++)
        {
                int j=next[i];
                while(j&&s2[j]!=s2[i])j=next[j];
                if(s2[j]==s2[i])next[i+1]=j+1;
                else next[i+1]=0;
        }
        for(int i=0,j=0;i<n;i++)
        {
                while(j&&s1[i]!=s2[j])j=next[j];
                if(s1[i]==s2[j])j++;
                if(j==m)printf("%d\n",i-m+2);
        }
        for(int i=1;i<=m;i++)printf("%d ",next[i]);
        return 0;
}
```

2. Trie

```
class Trienode :
    def __init__ (self) :
        self.children = {}
        self.is_end_of_word = False
class Trie :
    def __init__ (self) :
        self.root = Trienode()
    def insert(self, word) :
        cur = self.root
        for char in word :
            if not (char in cur.children) :
                cur.children[char] = Trienode()
            cur = cur.children[char]
        cur.is_end_of_word = True

    def search(self, word) :
        cur = self.root
        for char in word :
            if not (char in cur.children) :
                return False
            cur = cur.children[char]
        return cur.is_end_of_word

    def compatbl(self, word) :
        cur = self.root
        for char in word :
            #print(word, list(item for item in cur.children), cur.is_end_of_word, end = " ")
            if cur.is_end_of_word :
                #print("2")
                return False
            # DO NOT EXCHANGE THE POSITION OF TWO CONDITIONS!
            if not (char in cur.children) :
                #print("1")
                return True
            #print("3")
            cur = cur.children[char]
        return False

T = int(input())
for _ in range(T) :
    n = int(input())
    flag = True
    trie = Trie()
    for _ in range(n) :
```

```python
    s = input()
    if not trie.compatbl(s) :
        flag = False
    trie.insert(s)
print("YES" if flag else "NO")
```

```cpp
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#define mod 998244353
using namespace std;
int n,m,tot;
char s[51];
const int sigma_size=26;
struct trie
{
        int isname,check;
        trie *next[sigma_size];
}*root,pool[510005];
void insert()
{
        trie *p=root;
        int len=strlen(s);
        for(int i=0;i<len;i++)
        {
                int tmp=s[i]-'a';
                if(p->next[tmp]==0)
                    p->next[tmp]=&pool[tot++];
                p=p->next[tmp];
        }
        p->isname=1;
}
void check()
{
        trie *p=pool;
        int len=strlen(s);
        for(int i=0;i<len;i++)
        {
                int tmp=s[i]-'a';
                if(p->next[tmp]==0)
                {
                        printf("WRONG\n");
                        return;
                }
                p=p->next[tmp];
        }
        if(!p->isname)printf("WRONG\n");
        else if(p->check)printf("REPEAT\n");
        else printf("OK\n");
        p->check=1;
```

```
}
int main()
{
        scanf("%d",&n);
        root=&pool[tot++];
        for(int i=1;i<=n;i++)
        {
                scanf("%s",s);
                insert();
        }
        scanf("%d",&m);
        for(int i=1;i<=m;i++)
        {
                scanf("%s",s);
                check();
        }
        return 0;
}
```

Math

1. Euler's Sieve*

```
limit = LIMIT
isprime = [1] * (limit + 1)
prime = []
for i in range(2, limit + 1):
    if isprime[i]:
        prime.append(i)
    for j in prime:
        isprime[i * j] = 0
        if i % j == 0:
            break
```

```
#include <iostream>
#include <cstdio>
using namespace std;
bool isp[100000005];
int n,q,k,prime[10000005],cnt;
void sieve()
{
        for(int i=2;i<=n;i++)isp[i]=1;
        for(int i=2;i<=n;i++)
        {
                if(isp[i])cnt++,prime[cnt]=i;
                for(int j=1;j<=cnt&&prime[j]*i<=n;j++)
                {
                        isp[i*prime[j]]=0;
                        if(i%prime[j]==0)break;
                }
        }
}
```