Список задач по дисциплине "Алгоритмы, программирование и структуры данных" для групп К0709-22

1. Даны два числа `a` и `b` в десятичной системе счисления и основание некоторой системы счисления `c`. Найдите сумму этих чисел в системе счисления `c`. Результат представить в виде списка.

```
def sum(a, b, c):
    res = []
    carry = 0
    while a or b or carry:
        d1 = a % c
        d2 = b % c
        a //= c
        b //= c

        s = d1+d2+carry
        carry = s // c
        s %= c
        res.append(s)
    return res[::-1]
```

1. Даны два числа `a` и `b`, представленные в виде строк. Найдите произведение этих чисел и верните его в виде строки.

```
def sum(a, b):
    c = 10
    res = []
    if len(a) > len(b):
        a, b = b, a
    n = len(b)
    a = [0]*(n-len(a)) + a

    res = [0]*n
    carry = 0
    for i in range(n-1, -1, -1):
        res[i] = a[i]+b[i]+carry
        carry = res[i] // c
        res[i] %= c
    if carry:
        res = [carry]+res
    return res

def p(a, k):
    c = 10
```

```
    res = [0]*len(a)
    carry = 0
    for i in range(len(a)-1, -1, -1):
        res[i] = a[i]*k+carry
        carry = res[i] // c
        res[i] %= c
    if carry:
        res = [carry]+res
    return res

def product(a, b):
    a = list(map(int, a))
    b = list(map(int, b))
    if len(a) > len(b):
        a, b = b, a
    n = len(b)
    a = [0]*(n-len(a)) + a

    res = []
    power = 0
    for i in range(n-1, -1, -1):
        res = sum(res, p(a, b[i]) + [0]*power)
        power += 1
    for first_dig in range(len(res)):
        if res[first_dig] != 0:
            break
    return res[first_dig:]

a = input()
b = input()
print(''.join(map(str, p(list(map(int, a)), int(b)))))
print(''.join(map(str, product(a, b))))
```

1. Реализуйте алгоритм быстрого возведения числа `a` в степень `b`.

```
def fast_power(a, n):
    if n == 0: return 1
    res = a
    i = 1
    while i < n:
        if i*2 < n:
            res *= res
            i *= 2
        else:
            res *= a
            i += 1
    return res
```

1. Даны два числа `a` и `b`. Найдите их наибольший общий делитель.

```
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b%a, a)
```

1. Даны два числа a и b. Найдите пару чисел x и y, являющуюся решением уравнения вида: $a\,x+b\,y=НОД(a,b)$

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


if __name__ == '__main__':

    gcd, x, y = extended_gcd(30, 50)
    print('The GCD is', gcd)
    print(f'x = {x}, y = {y}')
```

1. Проверить, является ли число a простым.

```
def is_prime(a):
    for d in range(2, int(a**0.5)+1):
        if a % d == 0:
            return False
    return True
```

1. Найти k-тое по счету простое число. Число 2 считать простым числом с номером 1.

```
def nth_prime(n):
    primes = []
    d = 2
    while len(primes) < n:
        is_prime = True
        square_d = d**0.5
        for p in primes:
            if p > square_d:
                break
            if d % p == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(d)
        d += 1
    return primes[-1]
```

1. Найти количество простых чисел в диапазоне от $[0,n)$.

```
def erat(n):
    e = [1]*(n+1)
    e[0] = e[1] = 0
    for d in range(2, len(e)):
        if not e[d]:
            continue
        for d2 in range(d*2, len(e), d):
            e[d2] = 0
    return e

def count_primes(upto):
    return sum(erat(upto))
```

1. Дан список целых чисел `arr`. Реализовать сортировку простыми обменами, в качестве результата вернуть количество перестановок выполненных в процессе сортировки.

```
def swap_sort(arr):
    n = len(arr)
    swaps = 0
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]
                swaps += 1
    return swaps
```

1. Дан список целых чисел `arr`. Реализовать сортировку вставками (без использования бинарного поиска), в качестве результата вернуть количество перестановок выполненных в процессе сортировки.

```
def insertion_sort(arr):
    n = len(arr)
    swaps = 0
    for i in range(n):
        k = i
        while k > 0 and a[k-1] > a[k]:
            a[k], a[k-1] = a[k-1], a[k]
            swaps += 1
            k -= 1
    return swaps
```

1. Дан сортированный список целых чисел `arr` и число `x`. Найти индекс, на котором будет расположено число `x` в списке, после его добавления в список в порядке сортировки.

```
def sorted_place(arr, new):
    i = len(arr)//2
    while True:
        if i < len(arr) and arr[i] < new:
```

```
            if i == len(arr)-1:
                return len(arr)
            i += i // 2 + (i==1)
        elif i and arr[i-1] > new:
            if i == 1:
                return 0
            i -= i // 2
        else:
            return i

print(sorted_place(sorted([1, 2, 5, 8]), 3))
print(sorted_place([1, 3, 4, 5, 6], 2))
print(sorted_place([1, 2, 3], 100))
print(sorted_place([3, 4, 5, 6], 1))
```

1. Даны два сортированных списка `arr1` и `arr2`. Выполнить их слияние так, чтобы полученный список так же был сортирован.

```
def merge(a, b):
    merged = []
    a = a[::-1]
    b = b[::-1]
    while b:
        while a and a[-1] < b[-1]:
            merged.append(a.pop())
        merged.append(b.pop())
    merged.extend(a[::-1])
    return merged
```

1. Реализовать алгоритм сортировки слиянием с использованием галопирования.

```
def galloping(AB, n, C):
    C[:] = AB[:n]
    # r — указатель на конец результата # j — место последней вставки
    # m — длина остатка B
    r, j, m = 0, n, len(AB) - n
    for i in range(n):
        # k — степень двойки
        # l — указатель на 2^k-1 элемент k, l = 0, 0
        while l < m and AB[j+l] < C[i]:
            k += 1
            l = 2**k - 1
            if l >= m: l=m-1
            while l >= 0 and AB[j+l] > C[i]:
                l -= 1
        l += 1
        AB[r:r+l], AB[r+l] = AB[j:j+l], C[i]
        r, j, m = r + l + 1, j + l, m - l
```

1. Реализовать алгоритм быстрой сортировки.

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])

    (array[i + 1], array[high]) = (array[high], array[i + 1])

    return i + 1

def quicksort(array, low, high):
    if low < high:
        pi = partition(array, low, high)

        quicksort(array, low, pi - 1)
        quicksort(array, pi + 1, high)

def sort(arr):
    return quicksort(arr, 0, len(arr)-1)
```

1. На вершине лесенки, содержащей N ступенек, находится мячик, который начинает прыгать по ним вниз, к основанию. Мячик может прыгнуть на следующую ступеньку, на ступеньку через одну или через 2. (То есть, если мячик лежит на 8-ой ступеньке, то он может переместиться на 5-ую, 6-ую или 7-ую.) Определить число всевозможных "маршрутов" мячика с вершины на землю.

```
def routes_to_start(N):
    N += 1
    dp = [0]*N
    dp[-1] = 1

    for step in range(N-2, -1, -1):
        dp[step] = dp[step+1]
        if step+2 < N:
            dp[step] += dp[step+2]
        if step+3 < N:
            dp[step] += dp[step+3]
    return dp[0]

N = 5
print(routes_to_start(N))
```

1. Вычислите n-й член последовательности, заданной формулами:

$$ a_{2n} = a_n + a_{n-1}, \newline a_{2n+1} = a_n – a_{n-1}, \newline a_0 = a_1 = 1. $$

```python
def a(n):
    n += 1
    dp = [0]*n
    dp[0] = dp[1] = 1
    for i in range(n):
        k = i // 2
        if i % 2:
            dp[i] = dp[k] - dp[k-1]
        else:
            dp[i] = dp[k] + dp[k-1]

    return dp[n-1]


n = 5
print(a(n))
```

1. Даны две последовательности, требуется найти длину их наибольшей общей подпоследовательности.

```python
def lcs_len(a, b):
    n = len(a)
    m = len(b)
    dp = [[0 for _ in range(m+1)] for _ in range(n+1)]
    for i in range(1, n):
        for j in range(1, m):
            if a[i-1] == b[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[n-1][m-1]
# n = int(input())
# a = [int(x) for x in input().split()]
# m = int(input())
# b = [int(x) for x in input().split()]
# print(lcs_len(a, b))
```

1. Дано `N` золотых слитков массой $m_1, \ldots, m_N$. Ими наполняют рюкзак, который выдерживает вес не более `M`. Можно ли набрать вес в точности `M`?

```python
def can_get_weight(arr, m):
    can_get = [True] + [0]*m
    for new in arr:
        can_get = [can_get[weight] or weight - new >= 0 and
can_get[weight-new] for weight in range(m+1)]
    return can_get[m]
# n, m = map(int, input().split())
# arr = list(map(int, input().split()))
# print('YES' if can_get_weight(arr, m) else 'NO')
```

1. Определите расстояние Левенштейна для двух данных строк `s1` и `s2`.

```
def levenstein(str_1, str_2):
    n, m = len(str_1), len(str_2)
    if n > m:
        str_1, str_2 = str_2, str_1
        n, m = m, n

    current_row = range(n + 1)
    for i in range(1, m + 1):
        previous_row, current_row = current_row, [i] + [0] * n
        for j in range(1, n + 1):
            add, delete, change = previous_row[j] + 1, current_row[j -
1] + 1, previous_row[j - 1]
            if str_1[j - 1] != str_2[i - 1]:
                change += 1
            current_row[j] = min(add, delete, change)
    return current_row[n]

print(levenstein('алгоритмы', 'алкоритмы'))
```

1. Даны два упорядоченных по невозрастанию односвязных списка. Объедините их в новый упорядоченный по невозрастанию односвязный список.

```
class Node:
    def __init__(self, val, next):
        self.val = val
        self.next = next
    def __str__(self):
        return f'{self.val} {self.next}'

def merge_sorted(n1, n2):
    first = Node(None, None)
    last = first
    while n1 or n2:
        new_val = None
        if n1 and n2 and n1.val >= n2.val or n1 and not n2:
            new_val = n1.val
            n1 = n1.next
        else:
            new_val = n2.val
            n2 = n2.next
        last.next = Node(new_val, None)
        last = last.next
    return first.next

n1 = Node(12, Node(9, Node(7, Node(5, None))))
print(n1)
n2 = Node(15, Node(8, Node(6, Node(4, None))))
print(n2)
```

```
print(merge_sorted(n1, n2))
```

1. Дан односвязный список. Определить содержит ли он цикл. Список может содержать петли.

```
class Node:
    def __init__(self, val, next):
        self.val = val
        self.next = next
    def __str__(self):
        return f'{self.val} {self.next}'

def has_cycle(node):
    visited = set()
    while node.next:
        if node.next == node:
            return False
        if node.next in visited:
            return True
        node = node.next
        visited.add(node)
    return False


print(has_cycle(Node(12, Node(9, Node(7, Node(5, None))))))
n = Node(12, None)
n.next = n
print(has_cycle(n))
n2 = Node(1, Node(2, Node(3, None)))
n2.next.next.next = n2
print(has_cycle(n2))
```

1. Дана строка `S` состоящая из открывающихся и закрывающихся скобок '(' и ')'. Найти длину наибольшей правильной последовательности скобок.

   Последовательность скобок верна если:

   - Для каждой открытой скобки есть закрытая
   - Открытые скобки должны закрываться в соответствующем порядке.

```
def _max_correct_braces(s):
    nest = best = best_cnt = start_at = curr = 0
    for i, c in enumerate(s):
        if c == '(':
            curr = 0
            nest += 1
            continue
```

```
            nest -= 1
            curr += 2
            if nest < 0:
                nest = 0
                curr = 0
                start_at = i+1
                continue

            if nest == 0:
                curr = i-start_at+1

            if curr == best:
                best_cnt += 1
            elif curr > best:
                best = max(best, curr)
                best_cnt = 1

    if not best:
        best_cnt = 1

    return best, best_cnt

def max_correct_braces(s):
    m1, c1 = _max_correct_braces(s)
    m2, c2 = _max_correct_braces(s[::-1].translate(str.maketrans('()',
')(')))
    if m2 > m1:
        return m2, c2
    return m1, c1

s = input()
print(*max_correct_braces(s))
```

1. Дан список цифр (значения от 0 до 9), найти минимальную возможную сумму двух чисел, составленных из цифр в списке. Все цифры должны быть использованы.

   Любое сочетание цифр может быть использовано для составления чисел. Ведущие нули разрешены.

   Если составить 2 числа невозможно (например, n==0), тогда "сумма" - это значение единственно возможного числа.

```
def min_sum_from_digits(dig):
    dig = sorted(dig)
    res = 0
    offset = 1
    while dig:
```

```
        res += dig.pop() * offset
        if dig: res += dig.pop() * offset
        offset *= 10
    return res

print(min_sum_from_digits([1, 2, 3]))
print(min_sum_from_digits([3, 2, 1, 4, 5, 6, 9, 8, 7, 0]))
```

1. Реализовать алгоритм пирамидальной сортировки.

```
import abc
class Heap(abc.ABC):
    @abc.abstractmethod
    def _higher(self, x, y):
        pass

    def __init__(self, arr=None):
        arr = [] if arr is None else arr
        self.heap = []
        for a in arr:
            self.push(a)

    def push(self, a):
        self.heap.append(a)
        self.__bottom_to_top(len(self.heap)-1)

    def pop(self):
        if not self.heap:
            raise ValueError('heap is empty!')
        self.__swap(0, len(self.heap)-1)
        max = self.heap.pop()
        self.__top_to_bottom(0)
        return max

    def __top_to_bottom(self, i):
        child = 2*i + 1
        if child >= len(self.heap):
            return
        if child+1 < len(self.heap) and
self._higher(self.heap[child+1], self.heap[child]):
            child += 1
        if self._higher(self.heap[child], self.heap[i]):
            self.__swap(i, child)
            self.__top_to_bottom(child)

    def __bottom_to_top(self, i):
        parent = (i-1) // 2
        if parent < 0:
            return
        if self._higher(self.heap[i], self.heap[parent]):
```

```
            self.__swap(i, parent)
            self.__bottom_to_top(parent)

    def __swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

class MinHeap(Heap):
    def _higher(self, a, b):
        return a < b

class MaxHeap(Heap):
    def _higher(self, a, b):
        return a >= b

def pyramid_sort(arr):
    h = MaxHeap(arr)
    res = []
    while h.heap:
        print(*h.heap)
        res.append(h.pop())
    return res

print(*pyramid_sort([5, 33, 6, 1, 42, 105, 68]))
print(*pyramid_sort([1, 2, 3, 4, 5, 6]))
```

1. Дана строка `S` с повторяющимися буквами. Переставить буквы таким образом, чтобы одинаковые буквы не стояли рядом.

   Примечание: строка содержит только строчные латинские буквы и может иметь множество решений. Верните любое из них.

```
def split_adj(s):
    s = list(s)
    cnt = {}
    for c in s:
        if c not in cnt:
            cnt[c] = 0
        cnt[c] += 1
    res = ''
    while cnt:
        keys = list(cnt.keys())
        if len(keys) == 1 and (cnt[keys[0]] > 1 or res and res[-1] ==
keys[0]):
            raise ValueError('It is impossible to swap items to have
no adjacent duplicates')
        for k in keys:
            if not cnt[k]:
                del cnt[k]
            else:
```

```
                res += k
                cnt[k] -= 1
    return res

print(split_adj('aabbcc'))
```

```
abcabc
```

1. Конвертация из разных типов представления графов. Четыре типа, значит 4*3=12 функций.

```
from pprint import pprint

def edges_to_adj(e):
    adj = {}
    for a, b in edges:
        if a not in adj:
            adj[a] = set()
        adj[a].add(b)
    return adj

def edges_to_matrix(edges):
    """Convert graph edges representation to a matrix representation.
Note: vertices must be numbered 0 to N"""
    N = max(max(a, b) for a, b in edges)
    matrix = [[0 for _ in range(N+1)] for _ in range(N+1)]
    for a, b in edges:
        matrix[a][b] = 1
    return matrix

def edges_to_incidence(edges):
    M = len(edges)
    N = max(max(a, b) for a, b in edges) + 1
    inc = [[] for _ in range(N)]
    handled = set()
    for a, b in edges:
        if (a, b) in handled:
            continue
        for i in range(N):
            inc[i].append(0)
        inc[a][-1] = +1
        inc[b][-1] = -1
        if (b, a) in edges:
            inc[b][-1] = 1
            handled.add((b, a))
    return inc

def adj_to_matrix(adj):
    """Convert graph adj representation to a matrix representation.
Note: vertices must be numbered 0 to N"""
```

```
    N = max(adj.keys())
    matrix = [[0 for _ in range(N+1)] for _ in range(N+1)]
    for a, connected in adj.items():
        for b in connected:
            matrix[a][b] = 1
    return matrix

def adj_to_edges(adj):
    edges = set()
    for a, connected in adj.items():
        for b in connected:
            edges.add((a, b))
    return edges

def adj_to_incidence(adj):
    N = max(adj.keys())+1
    inc = [[] for _ in range(N)]
    handled = set()
    for a, connected in adj.items():
        for b in connected:
            if (a, b) in handled:
                continue
            for i in range(N):
                inc[i].append(0)
            inc[a][-1] = 1
            inc[b][-1] = -1
            if b in adj and a in adj[b]:
                inc[b][-1] = 1
                handled.add((b, a))
    return inc

def matrix_to_edges(matrix):
    edges = set()
    for a in range(len(matrix)):
        for b in range(len(matrix[0])):
            if matrix[a][b]:
                edges.add((a, b))
    return edges

def matrix_to_adj(matrix):
    adj = {}
    for a in range(len(matrix)):
        adj[a] = set()
        for b in range(len(matrix[0])):
            if matrix[a][b]:
                adj[a].add(b)
    return adj

def matrix_to_incidence(matrix):
    N = len(matrix)
```

```
    if not matrix:
        return []
    inc = [[] for _ in range(N)]
    handled = set()
    for a in range(N):
        for b in range(N):
            if (a, b) in handled:
                continue
            if not matrix[a][b]:
                continue
            for i in range(N):
                inc[i].append(0)
            inc[a][-1] = 1
            inc[b][-1] = -1
            if matrix[b][a]:
                inc[b][-1] = 1
                handled.add((b, a))
    return inc

def incidence_to_edges(inc):
    edges = set()
    if not inc:
        return edges
    for i in range(len(inc[0])):
        start = end = None
        two_way = False
        for j in range(len(inc)):
            if inc[j][i] > 0:
                if start is not None:
                    two_way = True
                    end = j
                    break
                start = j
            elif inc[j][i] < 0:
                end = j
        edges.add((start, end))
        if two_way:
            edges.add((end, start))
    return edges

def incidence_to_adj(inc):
    adj = {v: set() for v in range(len(inc))}
    if not inc:
        return adj
    for i in range(len(inc[0])):
        start = end = None
        two_way = False
        for j in range(len(inc)):
            if inc[j][i] > 0:
```

```
                if start is not None:
                    two_way = True
                    end = j
                    break
                start = j
            elif inc[j][i] < 0:
                end = j
        adj[start].add(end)
        if two_way:
            adj[end].add(start)
    return adj

def incidence_to_matrix(inc):
    matrix = [[0 for _ in range(len(inc))] for _ in range(len(inc))]
    if not inc:
        return matrix
    for i in range(len(inc[0])):
        start = end = None
        two_way = False
        for j in range(len(inc)):
            if inc[j][i] > 0:
                if start is not None:
                    two_way = True
                    end = j
                    break
                start = j
            elif inc[j][i] < 0:
                end = j
        matrix[start][end] = 1
        if two_way:
            matrix[end][start] = 1
    return matrix

adj1 = {0: {1}, 1: {2}, 2: {1, 0}}
pprint(adj_to_incidence(adj1), width=20)
edges1 = adj_to_edges(adj1)
pprint(edges_to_incidence(edges1), width=20)
matrix1 = adj_to_matrix(adj1)
pprint(matrix_to_incidence(matrix1), width=20)

edges2 = {(4, 0), (4, 1), (4, 2), (4, 3)}
pprint(edges_to_incidence(edges2), width=20)
matrix2 = edges_to_matrix(edges2)
pprint(matrix_to_incidence(matrix2), width=20)
adj2 = matrix_to_adj(matrix2)
print('adj2')
pprint(adj2, width=20)
print()
inc2 = adj_to_incidence(adj2)
pprint(adj_to_incidence(adj2))
```

```
matrix3 = [
    [0, 0, 0, 1],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [1, 0, 0, 0],
]
pprint(matrix_to_incidence(matrix3))
edges3 = matrix_to_edges(matrix3)
pprint(edges_to_incidence(edges3))
adj3 = matrix_to_adj(matrix3)
pprint(adj_to_incidence(adj3))

inc4 = [
    [1, 0, -3],
    [-1, 1, 0],
    [0, 1, 3],
]
pprint(incidence_to_edges(inc4))
pprint(incidence_to_adj(inc4))
print(incidence_to_matrix(inc4))
```

1. Дано N городов и известны расстояния между ними. Не все города связаны друг с другом дорогой. Найти все возможные маршруты из города A в город B (ни в один город не заходить дважды). Определить самый длинный и самый короткий маршрут.

```
def find_routes(edges, start, target):
    adj = {}
    for a, b, cost in edges:
        if a not in adj:
            adj[a] = set()
        adj[a].add((b, cost))

    routes = []
    def dfs(a, curr_route, curr_length):
        if a == target:
            routes.append((curr_route.copy(), curr_length))
            return
        for b, length in adj[a]:
            if b in curr_route: # loop
                continue
            curr_length += length
            curr_route.append(b)
            dfs(b, curr_route, curr_length)
            curr_route.pop()
            curr_length -= length

    dfs(start, [start], 0)
    return routes
```

```
def get_min_and_max_route(routes):
    min_i = max_i = 0
    for i in range(len(routes)):
        length = routes[i][1]
        if length < routes[min_i][1]:
            min_i = i
        if length > routes[max_i][1]:
            max_i = i
    return routes[min_i], routes[max_i]

routes = find_routes({
    (1, 2, 10),
    (1, 4, 30),
    (1, 5, 100),
    (2, 3, 50),
    (3, 5, 10),
    (4, 3, 20),
    (4, 5, 60),
}, 1, 5)
print(routes)
print(get_min_and_max_route(routes))
```

1. Дан неориентированный невзвешенный граф. Необходимо посчитать количество его компонент связности и вернуть их в виде двумерного списка, количество строк которого соответствует количеству компонент, а в строках содержится множество вершин каждой компоненты.

```
def components(adj):
    def dfs(adj, start, component, num):
        component[start] = num
        for end in adj[start]:
            if end not in component:
                dfs(adj, end, component, num)

    num = 0
    component = {}
    for a in adj:
        if a not in component:
            dfs(adj, a, component, num)
            num += 1

    return [{a for a, c_num in component.items() if c_num == i} for i
in range(num)]

print(components({1: {4}, 4: {1}, 0: {2}, 2: {5, 3}, 3: {5}, 5: {2}}))
```

1. Дан ориентированный невзвешенный граф. Определить является ли данный граф ациклическим.

```python
from collections import deque

def bfs_has_cycle(adj, start):
    visited = {start}
    queue = deque([start])

    while queue:
        node = queue.popleft()
        for neighbor in adj.get(node, []):
            if neighbor in visited:
                return True
            visited.add(neighbor)
            queue.append(neighbor)
    return False

def is_asyclic(adj):
    for start in adj:
        if bfs_has_cycle(adj, start):
            return False
    return True

assert not is_asyclic({1: {2}, 2: {3}, 3: {1}})
assert is_asyclic({1: {2}, 2: {3}, 3: {4, 5}})
```

1. Даны `N` процессов. Каждый процесс может быть запущен сразу, а может быть только после выполнения некоторого количества предыдущих процессов. Список зависимостей дан в списке `depend`. `depend[i]` - список процессов, от которых зависит процесс `i`. Найти порядок, в котором необходимо выполнять процессы, чтобы зависимый процесс начинался после выполнения предыдущих. Если таких порядков несколько, верните любой. Гарантируется, что существует хотя бы один процесс, который не имеет зависимостей.

```python
def topological_sort(adj):
    if not is_asyclic(adj):
        raise ValueError('graph must be asyclic in order to have a
topological sorting')
    visited = set()
    topo_order = []
    def dfs(a):
        visited.add(a)
        for b in adj.get(a, []):
            if b not in visited:
                dfs(b)
        topo_order.append(a)
    for a in adj:
        if a not in visited:
            dfs(a)
    return topo_order[::-1]
```

```
def inverse_adj(adj):
    inv = {}
    for a, neighbors in adj.items():
        for b in neighbors:
            if b not in inv:
                inv[b] = set()
            inv[b].add(a)

    print(inv)
    return inv

def order_processes(depend):
    return topological_sort(inverse_adj(depend))

print(order_processes({5: {}, 4: {}, 0: {4, 5}, 2: {5}, 1: {3, 4}, 3:
{2}}))
```

1. Дана система двусторонних дорог. N-периферией называется множество городов, расстояние от которых до выделенного города (столицы) больше N. Для данного N определите N-периферию.

```
import math

def deijkstra(N, adj, start):
    dist = [float('inf')]*N
    dist[start] = 0
    visited = set()
    while len(visited) != N:
        curr = min(set(range(N)) - visited, key=dist.__getitem__)
        for neighbor, length in adj[curr]:
            new_dist = dist[curr] + length
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
        visited.add(curr)
    return dist

def n_periphery(adj, start, N):
    dist = deijkstra(len(adj), adj, start)
    return [city for city in range(len(adj)) if dist[city] > N]

# def matrix_to_adj(matrix):
#     adj = {}
#     for i in range(len(matrix)):
#         adj[i] = set()
#         for j in range(len(matrix[0])):
#             if matrix[i][j] > 0:
#                 adj[i].add((j, matrix[i][j]))
#     return adj

# n, s, f = map(int, input().split())
```

```
# s -= 1
# f -= 1
# matrix = []
# for i in range(n):
#     matrix.append([int(a) for a in input().split()])
# adj = matrix_to_adj(matrix)
# dist = shortest_paths(n, adj, s)
# print(dist[f] if not math.isinf(dist[f]) else -1)
```

1. Система двусторонних дорог такова, что для любой пары городов можно указать соединяющий их путь. Найдите такой город, сумма расстояний от которого до остальных городов минимальна.

```
def find_city_with_max_neighbor_dist_sum(adj):
    res, max_sum = 0, 0
    for a, neighbors in adj.items():
        s = sum([length for b, length in neighbors])
        if s >= max_sum:
            res, max_sum = a, s
    return res, max_sum
```

1. За проезд по каждой дороге взымается некоторая пошлина. Найдите путь из города A в город B с минимальной величиной S+P, где S - сумма длин дорог пути, а P - сумма пошлин проезжаемых дорог.

```
import math

def deijkstra(N, adj, start):
    dist = [float('inf')]*N
    dist[start] = 0
    visited = set()
    while len(visited) != N:
        curr = min(set(range(N)) - visited, key=dist.__getitem__)
        for neighbor, length in adj[curr]:
            new_dist = dist[curr] + length
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
        visited.add(curr)
    return dist

# edges: (start, end, length, tax)

def edges_to_adj(edges):
    adj = {}
    for a, b, length, tax in edges:
        if a not in adj:
            adj[a] = set()
        total_length = length + tax
        adj[a].add((b, total_length))
```

```
def get_min_total(edges, a, b):
    adj = edges_to_adj(edges)
    dist = deijkstra(len(adj), adj, a)
    return dist[b]

# m, a, b = map(int, input().split())
# edges = [list(map(int, input().split())) for _ in range(m)]
# print(get_min_total(edges, a, b))
```

1. Дан взвешенный ориентированный граф с `n` узлами и `m` ребрами. Узлы пронумерованы от `0` до `n-1`, необходимо проверить, содержит ли граф цикл отрицательного веса.

   Примечание: `edges[i]` состоит из вершин `u`, `v` и веса.

```
def bellman_ford(v, edges, start):
    """Returns an array with min distances to all vertices from start.
    If graph contains a negative weight cycle, return -1"""
    dist = [float('inf')]*v
    dist[start] = 0
    for _ in range(v-1):
        for u, v, length in edges:
            if dist[v] > dist[u] + length:
                dist[v] = dist[u] + length
    for u, v, length in edges:
        if dist[v] > dist[u] + length:
            return -1
    return dist


class Solution:
     def isNegativeWeightCycle(self, n, edges):
         return int(any(
             bellman_ford(n, edges, start) == -1 for start in range(n)
         ))
```

1. Найти минимальное покрывающее дерево для заданного графа. Вывести список его ребер и суммарный вес.

```
class DisjSet:
    def __init__(self, n):
        self.rank = [1] * n
        self.parent = [i for i in range(n)]

    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):
        xset = self.find(x)
        yset = self.find(y)
        if xset == yset:
            return
        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset
        else:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + 1


def kruskalMST(n, edges):
    disj_set = DisjSet(n)
    edges = sorted(edges, reverse=True, key=lambda e: e[2])
    n_included = 0
    result = []
    while n_included < n - 1:
        a, b, w = edges.pop()
        if disj_set.find(a) != disj_set.find(b):
            n_included += 1
            result.append([a, b, w])
            disj_set.union(a, b)
    return result, sum(map(lambda e: e[2], result))


n, m = map(int, input().split())
edges = []
for _ in range(m):
    a, b, w = map(int, input().split())
    edges.append([a-1, b-1, w])

mst, total_w = kruskalMST(n, edges)
print(total_w)
```

1. В неориентированный взвешенный граф добавляют ребра. Напишите программу, которая, после добавления ребер, находит сумму весов ребер в компоненте связности.

   На вход подаются два числа `n` и `m` - количество вершин в графе и количество производимых добавлений и запросов. Далее следует список `add` из m строк. Каждая строка состоит из трех чисел `x, y, w`. Это означает, что в граф добавляется ребро из вершины x в вершину y веса w. Кратные ребра допустимы. И число `A` - вершина, для компоненты связности которой необходимо найти суммарный вес ребер.

```
# ejudge: https://informatics.msk.ru/mod/statements/view.php?
chapterid=1376#1

# простое решение за O(n^2)
# на информатиксе не проходит последние три теста по времени.
# n, ops = map(int, input().split())

# component = [0]*n # key is vertex number, value is its component
number
# size = [0]*n # key is component number, value is sum of its members
# for i in range(n):
#     component[i] = i # currently all vertexes are separated (there
is no edges), so each vertex has its own component
#     size[i] = 0 # no edges, so sum is zero

# for _ in range(ops):
#     query = [int(q) for q in input().split()]
#     op, args = query[0], query[1:]
#     if op == 2:
#         x = args[0] - 1
#         print(size[component[x]])
#         continue
#     x, y, w = args
#     x -= 1
#     y -= 1
#     comp_x, comp_y = component[x], component[y]
#     if comp_x == comp_y:
#         size[comp_x] += w
#         continue
#     # these vertexes were from different components, let's now merge
these components
#     for i in range(n):
#         if component[i] == comp_y:
#             component[i] = comp_x
#     size[comp_x] += w + size[comp_y]

# эффективное решение через Disjoint Set.
# на информатиксе не проходит последние 2 теста по времени, но это из-
за медленности Питона
# переписал на C++ этот же алгоритм и получил OK.

class DisjSet: # source: Lection 9
    def __init__(self, n):
        self.rank = [1] * n
        self.parent = [i for i in range(n)]
        self.size = [0]*n

    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, weight):
        xset = self.find(x)
        yset = self.find(y)
        if xset == yset:
            self.size[xset] += weight
            return
        if self.rank[xset] < self.rank[yset]:
            self.size[yset] += self.size[xset] + weight
            self.parent[xset] = yset
        elif self.rank[xset] > self.rank[yset]:
            self.size[xset] += self.size[yset] + weight
            self.parent[yset] = xset
        else:
            self.size[xset] += self.size[yset] + weight
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + 1


# n, ops = map(int, input().split())

# disj_set = DisjSet(n)

# for _ in range(ops):
#     query = [int(q) for q in input().split()]
#     op, args = query[0], query[1:]
#     if op == 2:
#         x = args[0] - 1
#         print(disj_set.size[disj_set.find(x)])
#         continue
#     x, y, w = args
#     x -= 1
#     y -= 1
#     disj_set.union(x, y, w)
```

1. Дан граф представляющий собой транспортную сеть с N вершина пронумерованными 1 до N и M ребрами. Найти максимальный поток из вершины 1 до вершины N.

```
from collections import defaultdict
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self. ROW = len(graph)
    def BFS(self, s, t, parent):
        visited = [False]*(self.ROW)
        queue = []
        queue.append(s)
        visited[s] = True
```

```
        while queue:
            u = queue.pop(0)
            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u
                    if ind == t:
                        return True
        return False
    def FordFulkerson(self, source, sink):
        parent = [-1]*(self.ROW)
        max_flow = 0
        while self.BFS(source, sink, parent):
            path_flow = float("Inf")
            s = sink
            while s != source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]
            max_flow += path_flow

            v = sink
            while v != source:
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]
        return max_flow
```

1. Дано бинарное дерево. Выполнить прямой/центрированный/обратный/уровневый обход дерева.

```
from collections import deque

class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
    def __str__(self):
        return f'Value: {val}\nLeft: {left}\nRight: {right}'

def nlr(node): # прямой
    if node is None: return
    print(node.val, end='')
    nlr(node.left)
    nlr(node.right)

def lnr(node): # центрированный
    if node is None: return
```

```
    lnr(node.left)
    print(node.val, end='')
    lnr(node.right)

def lrn(node): # обратный
    if node is None: return
    lrn(node.left)
    lrn(node.right)
    print(node.val, end='')

def dfs_level(root): # уровневый, через обход в ширину
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node is None: continue
        print(node.val, end='')
        queue.append(node.left)
        queue.append(node.right)


t = Node('F', Node('B', Node('A'), Node('D', Node('C'), Node('E'))),
Node('G', None, Node('I', Node('H'), None)))
nlr(t)
print()
lnr(t)
print()
lrn(t)
print()
dfs_level(t)
```

1. Дано бинарное дерево. Проверить является ли данное дерево сбалансированным.

```
class Node:
    def __init__(self, val, left, right):
        self.val = val
        self.left = left
        self.right = right
    def __str__(self):
        return f'Value: {val}\nLeft: {left}\nRight: {right}'

def bin_tree_height(root) -> int: # from task #40
    if root is None:
        return 0
    return 1 + max(bin_tree_height(root.left),
bin_tree_height(root.right))

def is_balanced_bin_tree(root) -> bool:
    if root is None:
        return True
    return abs(bin_tree_height(root.left) -
```

```
bin_tree_height(root.right)) <= 1

t1 = Node(1, Node(1, None, None), Node(1, None, None))
assert is_balanced_bin_tree(t1) == True
t2 = Node(1, t1, t1)
assert is_balanced_bin_tree(t2) == True
t3 = Node(1, t1, None)
assert is_balanced_bin_tree(t3) == False
t4 = Node(1, t3, t1)
assert is_balanced_bin_tree(t4) == True
t5 = Node(1, t4, t1)
assert is_balanced_bin_tree(t5) == False
```

1. Дано бинарное дерево. Проверить является ли данное дерево бинарным деревом поиска.

```
def isBST(root, less=float('inf'), greater=float('-inf')):
    if root is None:
        return True
    if not (greater <= root.data < less):
        return False
    return isBST(root.left, root.data, greater) and isBST(root.right,
less, root.data)

class Solution:
    def isBST(self, root):
        return isBST(root)
```

1. Дано бинарное дерево. Найти высоту дерева.

```
class Node:
    def __init__(self, val, left, right):
        self.val = val
        self.left = left
        self.right = right
    def __str__(self):
        return f'Value: {val}\nLeft: {left}\nRight: {right}'

def bin_tree_height(root) -> int:
    if root is None:
        return 0
    return 1 + max(bin_tree_height(root.left),
bin_tree_height(root.right))

t1 = Node(5, Node(6, None, None), Node(7, None, None))
assert bin_tree_height(t1) == 2
t2 = Node(5, Node(6, Node(7, Node(8, Node(9, None, None), None), None,
), None), None)
assert bin_tree_height(t2) == 5
t3 = Node(5, Node(6, Node(7, Node(8, Node(9, None, None), None),
```

```
None), None), t2)
assert bin_tree_height(t3) == 6
t4 = Node(5, Node(6, Node(7, t3, t3), None), t3)
assert bin_tree_height(t4) == 9
```

1. Дано бинарное дерево. Найти ширину дерева.

```
def get_max_width(root):
    max_width = 0
    level = [root]
    while level:
        max_width = max(max_width, len(level))
        new_level = []
        while level:
            node = level.pop()
            if node.right is not None:
                new_level.append(node.right)
            if node.left is not None:
                new_level.append(node.left)
        level = new_level[::-1]
    return max_width
```

1. Дано бинарное дерево поиска. Реализовать функцию поиска/вставки/удаления узла.

```
def find(node, x):
    if node is None:
        return False
    if node.val == x:
        return True
    return find(node.left, x) if x < node.val else find(node.right, x)


def insert(node, x):
    if x <= node.val
        if node.left is None:
            node.left = x
        else:
            insert(node.left, x)
    else:
        if node.right is None:
            node.right = x
        else:
            insert(node.right, x)

def delete(node, x):
    if node is None:
        return None
    if x < node.val:
        node.left = delete(node.left, x)
```

```
        return
    elif x > node.val:
        node.right = delete(node.right, x)
        return
    if node.left is None:
        temp = root.right
        root = None
        return temp
    elif node.right is None:
        temp = root.left
        root = None
        return temp
    next = root.right
    while next.left is not None:
        next = next.left
    node.val = next.val
    node.right = delete(node.right, node.val)
```

1. Дано бинарное дерево. Выполнить прямой/центрированный/обратный обход дерева не используя рекурсию.

```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
    def __str__(self):
        return f'Value: {self.val}\nLeft: {self.left}\nRight:
{self.right}'

def nlr(node): # прямой
    if node is None: return
    stack = [node]
    while stack:
        node = stack.pop()
        if node.right is not None:
            stack.append(node.right)
        if node.left is not None:
            stack.append(node.left)
        if not node.left and not node.right:
            print(node.val, end='')
        else:
            stack.append(Node(node.val))

def lnr(node): # центрированный
    if node is None: return
    stack = [node]
    while stack:
        node = stack.pop()
        if node.right is not None:
```

```
            stack.append(node.right)
        if not node.left and not node.right:
            print(node.val, end='')
        else:
            stack.append(Node(node.val))
        if node.left is not None:
            stack.append(node.left)

def lrn(node): # обратный
    stack = [node]
    while stack:
        node = stack.pop()
        if not node.left and not node.right:
            print(node.val, end='')
        else:
            stack.append(Node(node.val))
        if node.right is not None:
            stack.append(node.right)
        if node.left is not None:
            stack.append(node.left)

t = Node('F', Node('B', Node('A'), Node('D', Node('C'), Node('E'))),
Node('G', None, Node('I', Node('H'), None)))
nlr(t)
print()
lnr(t)
print()
lrn(t)
```

1. Дано бинарное дерево поиска. Найти все тупиковые узлы. Под тупиковым узлом понимается узел, добавление потомков к которому невозможно.

```
class Solution:
    def isDeadEnd(self, root):
        deadends = []
        def dfs(a, smaller, greater):
            if a is None:
                return
            if a.data - 1 == greater and a.data + 1 == smaller:
                deadends.append(a.data)
            dfs(a.left, a.data, greater)
            dfs(a.right, smaller, a.data)
        dfs(root, float('inf'), 0)
        return deadends
```

1. Дан корень бинарного дерева. Необходимо преобразовать дерево в односвязный список. Список должен использовать тот же класс `Node`, правый указатель должен ссылаться на следующий элемент в списке, левый всегда `None`. Список должен иметь тот же порядок, что и прямой обход бинарного дерева.

```
def tree_to_list(root):
    ll_start = Node(None, None, None)
    ll_end = ll_start
    stack = [root]
    while stack:
        node = stack.pop()
        if node.right is not None:
            stack.append(node.right)
        if node.left is not None:
            stack.append(node.left)
        if not node.left and not node.right:
            ll_end.right = Node(node.val, None, None)
            ll_end = ll_end.right
        else:
            stack.append(Node(node.val))
    return ll_start.right

t = Node('F', Node('B', Node('A'), Node('D', Node('C'), Node('E'))),
Node('G', None, Node('I', Node('H'), None)))
print(tree_to_list(t))
```

1. Дано представление полного бинарного дерева в виде списка. Необходимо построить бинарное дерево.

```
from collections import deque
def convert(head):
    tree = Tree(head.data)
    level = deque([tree])
    while level:
        node = level.popleft()
        if head.next:
            node.left = Tree(head.next.data)
            head = head.next
            level.append(node.left)
        if head.next:
            node.right = Tree(head.next.data)
            head = head.next
            level.append(node.right)
    return tree
```

1. Дано бинарное дерево поиска с целочисленными ключами. Найти преемника и предшественника данного ключа `key`. Если какого-то значения не существует, верните вместо него `None`.

   Примечание: преемник и предшественник - ближайшие значения после и до указанного ключа.

```
def get_pre_suc(root, key):
    pre, suc = Node(-1), Node(-1)
```

```
    while root:
        if root.key == key:
            if root.left:
                pre = root.left
                while pre.right:
                    pre = pre.right
            if root.right:
                suc = root.right
                while suc.left:
                    suc = suc.left
            return pre, suc

        if root.key > key:
            suc = root
            root = root.left
        else:
            pre = root
            root = root.right
    return pre, suc

class Solution:
    def findPreSuc(self, root, pre, suc, key):
        got_pre, got_suc = get_pre_suc(root, key)
        pre.key, suc.key = got_pre.key, got_suc.key
```

1. Найти Z-функцию для строки.

```
def z_func(s):
    n = len(s)
    z = [0]*n
    z[0] = n
    l, r = 0, 0
    for i in range(1, n):
        if i <= r and z[i-l] < r-i+1:
            z[i] = z[i-l]
            continue
        l = i
        if i > r:
            r = l-1
        while r+1 < n and s[r+1] == s[r+1-l]:
            r += 1
        z[i] = r-l+1
    return z

s = input()
print(z_func(s))
```

1. Найти префикс-функцию для строки.

```
def prefix(s):
    n = len(s)
    pref = [0]*n
    for i in range(1, n):
        p = pref[i-1]
        while p > 0 and s[p] != s[i]:
            p = pref[p - 1]
        if s[p] == s[i]:
           p += 1
        pref[i] = p
    return pref

# s = input()
# print(*prefix(s))
```

1. Реализовать полиномиальную хеш-функцию для строк.

```
def hash_str(s, p=31, m=2**32):
    hash = 0
    p_pow = 1
    for i in range(len(s)):
        hash = (hash + (1 + ord(s[i]) - ord('a')) * p_pow) % m
        p_pow = (p_pow * p) % m
    return hash

print(hash_str('abracadabra'))
print(hash_str('abracadabre'))
```

1. Найти и вернуть все индексы начала вхождения строки `pat` в строку `text`.

```
def find_substr(text, pat):
    n = len(text)
    m = len(pat)
    matches = []
    z = z_func(pat+'#'+text)
    for i in range(len(text)):
        if z[i+m+1] == m:
            matches.append(i)
    return matches

s = input()
pat = input()
print(*find_substr(s, pat))
```

1. Реализовать сортировку списка строк на основе хеширования.

```
def sortUsingHash(a, n):
     Max = max(a)
     Min = abs(min(a))
```

```
      hashpos = [0] * (Max + 1)
      hashneg = [0] * (Min + 1)
      for i in range(0, n):
            if a[i] >= 0:
                  hashpos[a[i]] += 1
            else:
                  hashneg[abs(a[i])] += 1
      for i in range(Min, 0, -1):
            if hashneg[i] != 0:
                  for j in range(0, hashneg[i]):
                        print((-1) * i, end=" ")
      for i in range(0, Max + 1):
            if hashpos[i] != 0:
                  for j in range(0, hashpos[i]):
                        print(i, end=" ")

a = [-1, -2, -3, -4, -5, -6, 8, 7, 5, 4, 3, 2, 1, 0]
n = len(a)
sortUsingHash(a, n)
```