```python
from collections import deque, defaultdict
from copy import deepcopy
from heapq import *

# graph = defaultdict(dict, {})
# graph['jmeno uzlu'] = ['jmenu kam vede hrana']
# graph['jmeno uzlu'] = {'jmenu kam vede hrana': cena}

def BFS(gr, s):
    """ Breadth first search
    Returns a list of nodes that are "findable" from s """
    prev = {s:None}
    q = deque([s])
    while q:
        node = q.popleft()
        for each in gr[node]:
            if each not in prev:
                prev[each] = node
                q.append(each)
    return prev

def shortest_hops(gr, s):
    """ Finds the shortest number of hops required
    to reach a node from s. Returns a dict with mapping:
    destination node from s -> no. of hops
    """
    dist = {}
    q = deque([s])
    explored = {s}
    for n in gr.nodes():
        if n == s: dist[n] = 0
        else: dist[n] = float('inf')
    while len(q) != 0:
        node = q.popleft()
        for each in gr[node]:
            if each not in explored:
                explored.add(each)
                q.append(each)
                dist[each] = dist[node] + 1
    return dist

def undirected_connected_components(gr):
    """ Returns a list of connected components
    in an undirected graph """
    explored = set([])
    con_components = []
    for node in gr:
        if node not in explored:
            reachable = BFS(gr, node).keys()
            con_components.append(reachable)
            explored |= reachable
    return con_components

def DFS(gr, s, path=None):
    """ Depth first search
    Returns a list of nodes "findable" from s """
    if path is None:
        path = set()
    if s in path:
        return path
    path.add(s)
    for each in gr[s]:
        if each not in path:
            DFS(gr, each, path)
    return path

def topological_ordering(digr_ori):
    """ Returns a topological ordering for a
    acyclic directed graph """

    def find_sink_node(digr):
        """ Finds a sink node (node with all incoming arcs)
        in the directed graph. Valid for a acyclic graph only """
        # first node is taken as a default
        node = next(iter(digr.keys()))
        while digr[node]:
            node = digr[node][0]
```

```python
        return node

    digr = deepcopy(digr_ori)
    ordering = []
    n = len(digr)
    while n > 0:
        sink_node = find_sink_node(digr)
        ordering.append((sink_node, n))
        del digr[sink_node]
        n -= 1
    return ordering


def transposed(digr):
    """ Returns the transpose of directed graph
    with edges reversed and nodes same """
    trans = {}
    for n in digr:
        for edge in n:
            trans[edge] = trans.get(edge, []) + [n]
    return trans

def directed_connected_components(digr):
    """ Returns a list of strongly connected components
    in a directed graph using Kosaraju's two pass algorithm """

    def outer_dfs(digr, node, explored, path):
        if node in path or node in explored:
            return False
        path.append(node)
        for each in digr[node]:
            if each not in path or each not in explored:
                outer_dfs(digr, each, explored, path)

    def DFS_loop(digr):
        """ Core DFS loop used to find strongly connected components
        in a directed graph """
        explored = set([]) # list for keeping track of nodes explored
        finishing_times = [] # list for adding nodes based on their finishing times
        for node in digr:
            if node not in explored:
                leader_node = node
                inner_DFS(digr, node, explored, finishing_times)
        return finishing_times

    def inner_DFS(digr, node, explored, finishing_times):
        """ Inner DFS used in DFS loop method """
        explored.add(node) # mark explored
        for each in digr[node]:
            if each not in explored:
                inner_DFS(digr, each, explored, finishing_times)
        # adds nodes based on increasing order of finishing times
        finishing_times.append(node)

    finishing_times = DFS_loop(transposed(digr))
    # use finishing_times in descending order
    explored, connected_components = [], []
    for node in finishing_times[::-1]:
        component = []
        outer_dfs(digr, node, explored, component)
        if component:
            explored += component
            connected_components.append(component)
    return connected_components

def shortest_path(gr, fro, to):
    """ Finds the shortest path from s to every other vertex findable
    from s using Dijkstra's algorithm in O(mlogn) time. Uses heaps
    for super fast implementation """

    closed = set()
    dist = {fro:0}
    heap = [(0, fro)]
    prev = {}

    while heap:
        curd, cur = heappop(heap)
```

```
        if cur == to:
            return curd, prev
        if cur in closed:
            continue
        closed.add(cur)

        for n, cost in gr[cur].items():
            if n in closed:
                continue
            if curd + cost < dist.get(n, float('inf')):
                dist[n] = curd + cost
                prev[n] = cur
                heappush(heap, (dist[n], n))

# g = defaultdict(dict, {
#     0: {1: 2, 2: 1, 3: 10},
#     1: {3: 1},
#     2: {3: 4},
# })
# print(shortest_path(g, 0, 3))

def minimum_spanning_tree(gr):
    weight = 0
    heap = []
    tree = []

    start = next(iter(gr.keys()))
    connected = {start}

    for to, cost in gr[start].items():
        heap.append((cost, start, to))
    heapify(heap)

    while heap:
        cost, fro, to = heappop(heap)
        if to in connected:
            continue
        connected.add(to)
        tree.append((fro, to))
        weight += cost
        for n, cost in gr[to].items():
            heappush(heap, (cost, to, n))
    return weight, tree

# g = defaultdict(dict, {
#     0: {1: 2, 2: 1},
#     1: {3: 1, 0: 2},
#     2: {3: 4, 0: 1},
#     3: {2: 4, 1: 1},
# })
# print(minimum_spanning_tree(g))
```

```python
def edmonds_karp(C, source, sink):
    """Max flow"""

    def bfs(C, F, source, sink):
        queue = [source]
        prev = {source: None}
        while queue:
            u = queue.pop(0)
            for v in range(len(C)):
                if C[u][v] - F[u][v] > 0 and v not in prev:
                    prev[v] = u
                    if v == sink:
                        ret = [v]
                        while ret[-1]:
                            ret.append(prev[ret[-1]])
                        ret.reverse()
                        return list(zip(ret[:-1], ret[1:]))
                    queue.append(v)
        return None

    n = len(C) # C is the capacity matrix
    F = [[0] * n for i in range(n)]
    # residual capacity from u to v is C[u][v] - F[u][v]

    while True:
        path = bfs(C, F, source, sink)
        if not path:
            break
        # traverse path to find smallest capacity
        flow = min(C[u][v] - F[u][v] for u,v in path)
        # traverse path to update flow
        for u,v in path:
            F[u][v] += flow
            F[v][u] -= flow
    return sum(F[source][i] for i in range(n)), F

def min_cut(C, F, source):
    queue = [source]
    cut = []
    visited = {source}
    while queue:
        u = queue.pop(0)
        for v in range(len(C)):
            if v not in visited:
                if C[u][v] - F[u][v] > 0:
                    queue.append(v)
                elif C[u][v] != 0:
                    cut.append((u, v))
    return cut

# g = [
#     [0, 2, 1, 0],
#     [0, 0, 0, 1],
#     [0, 0, 0, 4],
#     [0, 0, 0, 0],
# ]
# print(edmonds_karp(g, 0, 3))
# print(min_cut(g, edmonds_karp(g, 0, 3)[1], 0))
```

```python
from math import *

# vec-mult of two vectors
# The area of the parallelogram
# negative for clockwise turn, and zero if the points are collinear.
def cross(a, b):
    return a[0]*b[1] - a[1]*b[0]

# equals |a|*|b|*cos(alf)
def dot(a, b):
    return a[0]*b[1] + a[1]*b[0]

def angle(a, b):
    return acos(dot(a, b)/hypot(*a)/hypot(*b))

# perpendicular vector of unit length
def normal(a):
    h = hypot(*a)
    return [-a[1]/h, a[0]/h]

def area_regular_polygon(n, s):
    return 0.25 * n * s**2 / tan(pi/n)

def area_polygon(vx):
    return abs(sum(cross(vx[i-1], vx[i]) for i in range(len(vx)))) / 2
```

```python
from .basic import *

def convex_hull(points):
    """Computes the convex hull of a set of 2D points.
    Input: an iterable sequence of (x, y) pairs representing the points.
    Output: a list of vertices of the convex hull in counter-clockwise order,
      starting from the vertex with the lexicographically smallest coordinates.
    Implements Andrew's monotone chain algorithm. O(n log n) complexity.
    """
    # Sort the points lexicographically (tuples are compared lexicographically).
    # Remove duplicates to detect the case we have just one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly repeated multiple times.
    if len(points) <= 1:
        return points

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenation of the lower and upper hulls gives the convex hull.
    # Last point of each list is omitted because it is repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1]

# 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
# Returns a positive value, if OAB makes a counter-clockwise turn,
# negative for clockwise turn, and zero if the points are collinear.
def cross(o, a, b):
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

# Example: convex hull of a 10-by-10 grid.
print(convex_hull([(i//10, i%10) for i in range(100)]) == [(0, 0), (9, 0), (9, 9), (0, 9)])
```