In [1]:

```
def chunk_list(data_list, chunk_size):
    n = max(1, chunk_size)
    return (data_list[i:i+chunk_size] for i in range(0, len(data_list), chunk_size))


class Graph:

    graph = {}
    current = 0

    def __init__(self, data):
        for edge in data:
            self.add_entry(edge)

        self.reset_current()

    def add_entry(self, edge):
        entry = {'path': int(edge[1]), 'weight': int(edge[2])}
        index = int(edge[0])
        if index in self.graph:
            if list(filter(lambda entry: entry['path'] == int(edge[1]), self.graph[index])):
                return

            self.graph[index].append(entry)
        else:
            self.graph[index] = [entry]

        self.add_entry([edge[1], edge[0], edge[2]])

    def reset_current(self):
        self.current = min(self.graph.keys()) - 1

    def __iter__(self):
        return self

    def __next__(self):
        self.current += 1

        if self.current not in self.graph:
            self.reset_current()
            raise StopIteration

        return self.current

    def breadth_first_until_vertex(self, vertex, counter = []):
        is_loop_found = False
        result_counter = []

        for candidate in counter:
            if not candidate['is_need_to_check']:
                continue
            next_path_list = self.graph[candidate['path'][-1]]
            for path in next_path_list:
                result = dict(
                    path=[],
                    is_need_to_check=candidate['is_need_to_check'],
                    is_loop_found=candidate['is_loop_found']
                )
                if path['path'] == vertex and len(candidate['path']) > 1:
                    is_loop_found = True
                    result['is_loop_found'] = True

                # This path is looping somewhere else before reaching needed vertex
                if path['path'] in candidate['path']:
                    result['is_need_to_check'] = False

                path = candidate['path'] + [path['path']]
                result['path'] = path
                result_counter.append(result)

        if is_loop_found:
            result = filter(lambda path: path['is_loop_found'] and path['is_need_to_check'], result_counter)

            return list(map(lambda x: x['path'], result))

        return self.breadth_first_until_vertex(vertex, result_counter)


    def find_smallest_loop_for_vertex(self, vertex):
        counter = []
        for edge in self.graph[vertex]:
            counter.append(dict(path=[edge['path']], is_need_to_check=True, is_loop_found=False))

        result_loop_list = []
        smallest_loop_list = self.breadth_first_until_vertex(vertex, counter)
        for loop in smallest_loop_list:
            result_loop_list.append([vertex] + loop)

        return result_loop_list

    def find_smallest_loops(self):
        found_loop_list = []
        loop_hash_list = []
        for vertex in graph:
            loop_list = self.find_smallest_loop_for_vertex(vertex)

            for loop in loop_list:
                loop_copy = loop.copy()
                loop_copy.sort()
                loop_hash = hash(tuple(loop_copy))

                if loop_hash in loop_hash_list:
                    continue

                found_loop_list.append(loop)
                found_loop_list.append(loop)
                loop_hash_list.append(loop_hash)

        return found_loop_list

    def get_weight_for_edge(self, edge):
        for entry in self.graph[edge[0]]:
            if entry['path'] == edge[1]:
                return entry['weight']

        return None

    def get_smallest_weight_for_route(self, route):
        start = route[0]
        min_weight = None
        edge = []

        for index, vertex in enumerate(route[1:]):
            weight = self.get_weight_for_edge((start, vertex))

            if not min_weight or weight < min_weight:
                min_weight = weight
                edge = [start, vertex]

            start = vertex

        edge.sort()
        return tuple(edge), min_weight


with open('input.txt', 'r') as input_file:
    data = input_file.read()
    data_list = data.split()

    n = data_list[0]
    graph_data = chunk_list(data_list[1:], 3)

    graph = Graph(graph_data)
    smallest_loop_list = graph.find_smallest_loops()
    marked_edge_list = []
    weight_sum = 0

    for loop in smallest_loop_list:
        edge, weight = graph.get_smallest_weight_for_route(loop)

        if edge in marked_edge_list:
            continue

        marked_edge_list.append(edge)
        weight_sum += weight

    with open('output.txt', 'w') as output_file:
        edge_string_list = map(lambda edge: map(str, edge), marked_edge_list)
        edge_string_list = map(lambda edge: ' '.join(edge), edge_string_list)
        nl = '\n'
        result = f'{weight_sum} {len(marked_edge_list)} {nl}{nl.join(edge_string_list)}'
        output_file.write(result)
```