Bipartite

```python
def bipartite(G):
    checked = {}

    def _iter_check(node, side):
        if node in checked:
            return
        checked[node] = side
        for neighbor in G[node]:
            _iter_check(neighbor, not side)

    for node in G:
        _iter_check(node, True)

    def _valid(subset):
        for node in subset:
            for neighbor in G[node]:
                if neighbor in subset:
                    return False
        return True

    left_set  = set(filter(lambda x: checked[x], checked))
    right_set = set(G.keys()) - left_set

    if _valid(left_set) and _valid(right_set):
        return left_set
```

Feel the Love

```python
def feel_the_love(G, i, j):
    # return a path (a list of nodes) between `i` and `j`,
    # with `i` as the first node and `j` as the last node,
    # or None if no path exists
    path = dijkstra_path(G, i)
    if not j in path:
        return None

    node_a, node_b = max_weight_edge(G, i)
    path_a = path[node_a]
    path_b = (dijkstra_path(G, node_b))[j]

    return path_a + path_b
```

Weighted Graph

```
def create_weighted_graph(bipartiteG, characters):
    G = defaultdict(dict)
    for char_a, char_b in itertools.combinations(characters, 2):
        a_books = set(bipartiteG[char_a])
        b_books = set(bipartiteG[char_b])

        inter_book_num = float(len(a_books.intersection(b_books)))
        if inter_book_num == 0:
            continue

        prob = inter_book_num / (len(a_books)+len(b_books)-inter_book_num)
        G[char_a][char_b] = prob
        G[char_b][char_a] = prob

    return G
```

Finding the Best Flight

```
def find_best_flights(flights, origin, destination):
    air_graph = build_air_graph(all_flights)

    possible_path = []

    for citys in get_possible_path(air_graph, origin, destination):
        possible_path += [city_to_path(air_graph, citys)]

    feasible = []
    for item in possible_path:
        feasible += filter(lambda x: valid_line(flights, x), item)

    if not feasible:
        return None

    cheap_flights = cost_efficiency(flights, feasible)

    return list(time_efficiency(flights, cheap_flights))

```

Constantly Connected

```python
def process_graph(G):
    global global_G
    global_G = G
    for node in global_G:
        to_visit = global_G[node].keys()

        while to_visit:
            new_node = to_visit.pop()

            global_G[node][new_node] = 1
            global_G[new_node][node] = 1

            for x in global_G[new_node]:
                if x not in global_G[node]:
                    to_visit += [x]
```

Distance Oracle (I)

```python
def create_labels(binarytreeG, root):
    # BFS for the binary tree, meanwhile labeling each node in each level
    labels = {root: {root: 0}}
    frontier = [root]
    while frontier:
        cparent = frontier.pop(0)
        for child in binarytreeG[cparent]:
            if child in labels:
                continue
            labels[child] = {child: 0}
            weight = binarytreeG[cparent][child]
            labels[child][cparent] = weight
            # make use of the labels already computed
            for ancestor in labels[cparent]:
                labels[child][ancestor] = weight + labels[cparent][ancestor]
            frontier += [child]
    return labels
```

Distance Oracle (II)

```python
def count_nodes(treeG, node):
    # count all sub-nodes including itself
    cnts = {}
    visited = {}
    cnts[node] = count_nodes_rec(treeG, node, cnts, visited)
    return cnts

def count_nodes_rec(treeG, node, cnts, visited):
    visited[node] = True
    frontier = [node]
    cnts[node] = 1
    for v in treeG[node]:
        if v not in visited:
            cnts[node] += count_nodes_rec(treeG, v, cnts, visited)
    return cnts[node]
```

Finding a Favor

```python
def maximize_probability_of_favor(G, v1, v2):
    # your code here
    # call either the heap or list version of dijkstra
    # and return the path from `v1` to `v2`
    # along with the probability that v1 will do a favor
    # for v2

    def _count_edges():
        return sum([len(G[v]) for v in G])

    G = reform_graph(G)

    # Theata(dijkstra_list) = Theata(n^2 + m) = Theata(n^2)
    # Theata(dijkstra_heap) = Theata(n * log(n) + m * log(n)) = Theata(m * log(n))
    node_num = len(G.keys())
    edge_num = _count_edges()

    if edge_num * log(node_num) <= node_num ** 2:
        dist_dict = dijkstra_heap(G, v1)
    else:
        dist_dict = dijkstra_list(G, v1)

    path = []
    node = v2
    while True:
        path += [node]
        if node == v1:
            break
        _, node = dist_dict[path[-1]]
```