General: For project 2 I implemented the Graph and Vertex classes to build the graph. The graph was built in the same way for both q1 and q2 as its characteristics remained the same. The Vertex object has as attributes the flag string, points, coordinates and the visited status (initialized with the False value), as once a flag is picked to be part of the solution for either of the variation of the game, the flag (vertex object) becomes visited.

Q1: Complexity p2q1 – get_route(p,v,flags) O(V^2) – V = total no. of vertices. V1 = O(V^2), V2 = O(V^2)

The first version built into the v1(p,graph_flags) is using random sampling to get the minimum points required by touching the flags(graph nodes), therefore it is a Greedy algorithm. It does so by keeping a variable total_points_gained(int) and a list variable list_solution_flags initialized with one element: the starting point.The algorithms uses a while loop with the condition that the total_points_gained < p. Until that condition is fulfilled, the function generates random samples of unvisited flags with the size of approximately half of number of vertices that the flag_graph consists of. From that random sample it finds the next_flags to be added to the list_solution_flags by using a get_min_distance_vertex(current_flag, random_sample). This function will get the vertex that is closest to the last vertex added to the list_solution_flags and add it to the list itself. Whenever a new vertex is added to the list, its points will also be added to the total_points_gained. Once the total_points_gained is $\geqslant$ p the function removes the starting point from the list_solution_flags and returns the list.

The second version of the game(function v2) is making use of v1 and storing what this returns in a variable partial_soluiton. The algorithm is also greedy because of the use of v1. However, to return back to the SP in the shortest distance possible, the v2 algorithm is calculating the distance from current vertex to the starting point by using the vertex method distance_to_vertex. It then compares this distance to the distance that would take to go to back trough the same vertices and pick one of them to go from directly to the starting point. If the distances show that the return route should be trough previous vertices, the partial_solution will be extended with this specific flags(vertices) and returned.

For both versions, in the main function get_route, the returned lists are returned as a list of flag strings.

Q2: Complexity p2q1 – get_routes(p,v,flags,n) O(V^2) = total no. of vertices. V1 = O(V^2), V2 = O(V^2)

For the second question, to keep a relatively good speed of execution I have chosen to use the same type of algorithms (greedy ones) and adapt them to the scenario of more players. To better find the new vertices for each player I stored the data into a dictionary (picked_flags_by_players_dict) with the players' numbers as keys. The value for each key was a list of vertices that the player has to visit.

For 1st version of the game to find the flags that each player should touch, a while loop was created similar to the one on p1q1 v1. Inside the while loop three local variables were established (min_distance, player, following-vertex. These variables store information about a 'global minimum' vertex. The next step is to go trough a loop for all the players in the picked_flags_by_players_dict. Within a nested for loop is then generating a random sample and the closest next_vertex is got by using the same get_min_distance_vertex function. For the first next_vertex found the three variables (min_distance, player and following_vertex) are updated, but for the next iterations of the for loop the next_vertex will be compared with the following vertex. The next_vertex will become the following_vertex only when the next_vertex < following_vertex, or next_vertex = following_vertex, but the next_vertex points are more. In this way the outcome may show that some players do not have to touch any flags as long as the distances that the other's touch are better. Every time a new following_vertex is found, it is appended to the list located where the key of the dict is the player's number that it was found for. When p $\leqslant$ points the picked_flags_by_player is returned. The dict will be returned as a 2d list of flag strings in the main function get_routes(p,v,flags,n)

The second version combines p2q2_v1 with p2_q2_v2. It therefore starts from the dict_solution_v1(generated by v1 of p2q2) and then for each player's list of flags, it compares the distance between the last touched flag and the starting point with, the distance that would be traversed by going back and picking another flag from which to go directly to the SP. If going back is faster, than the flags that are traversed again will be added to the list of the specific player. The function returns then the dict_solution_v1 with the updated changes. In the main function it will be modified into a 2d list of string.

For p2q1, I chose to use a Greedy algorithm.
Isinstance : https://pynative.com/python-isinstance-explained-with-examples/
HASH https://stackoverflow.com/questions/4901815/object-of-custom-type-as-dictionary-key