Part1

```python
def pq(data, P, init_centroids, max_iter):
    data = np.array(data, dtype='float32')
    init_centroids = np.array(init_centroids, dtype='float32')
    N, M = data.shape
    K = 256
    MP = int(M / P)
    result_centroids = np.array([])

    for i in range(P):
        sliced_data = data[:, i * MP:(i + 1) * MP]  # sliced data by P
        centroids = init_centroids[i,].copy() #copy another centroids

        for _ in range(max_iter):
            ### change to L1 distance by using distance_matrix methol with parameter 1
            row_dis = distance_matrix(sliced_data, centroids, 1)
            row_label = np.argmin(row_dis, axis=1) # then sort the distance

            for j in range(K):
                index_list = np.where(row_label == j)[0] # search the point that belong to centroid

                temp_data = [sliced_data[index] for index in index_list]
                if len(temp_data) != 0: # if no point belongs to centroid, centroid should stay the same
                    ### because change to L1 distance, kmedian should be the beat methol instead of kmeans
                    centroids[j] = np.median(temp_data, axis=0) #updata the centroids


        row_dis = distance_matrix(sliced_data, centroids, 1)
        row_label = np.argmin(row_dis, axis=1)

        if i == 0:
            result_label = np.array([row_label]).T.copy()
            result_centroids = np.array([centroids.copy()])
        else:
            temp = np.append(result_centroids, centroids)
            dim = result_centroids.shape
            result_centroids = temp.reshape(dim[0] + 1, dim[1], dim[2])
            result_label = np.column_stack((result_label, row_label)).astype(np.uint8)

    return result_centroids, result_label
```

The implementation of part 1 already written on the annotations.

About change to L1 dsitance, there are two point:

1. use scipy.spatial.distance_matrix to caculate L1 distance

2. usw k-median instead of k-means

Part2

```python
def query(queries, codebooks, codes, T):
    # print(codes)
    Q, M = queries.shape
    P, K, _ = codebooks.shape
    N, P = codes.shape
    MP = int(M / P)
    candidates = []

    inverted_index = {}
    for i in range(N):  # use a dict with tuple as key to search index instead of np.where
        if tuple(codes[i]) not in inverted_index.keys():
            inverted_index[tuple(codes[i])] = [i]
        else:
            inverted_index[tuple(codes[i])].append(i)

    one_list = np.array([np.zeros((P)).astype(np.int32)] * P)
    for i in range(P):  # one_list = [[0,1],[1,0]] if dim == 2, [[1,0,0],[0,1,0],[0,0,1]] if dim==3......dim==4
        one_list[i][i] = 1

    for q in range(Q):
        for i in range(P):
            sliced_data = queries[q, i * MP:(i + 1) * MP]
            row_dis = distance_matrix([sliced_data], codebooks[i], 1)

            ### sort distance
            sort_index = row_dis[0].argsort()
            if i == 0:
                sorted_matrix = np.array([sort_index])
                dis_matrix = np.array(row_dis)
            else:
                sorted_matrix = np.append(sorted_matrix, [sort_index], axis=0)
                dis_matrix = np.append(dis_matrix, row_dis, axis=0)

        distance_dict = {(0,) * P: distance(index_conv([0] * P, sorted_matrix), dis_matrix)}  # add(0..0) to dis_dict
        used_index = {}  # store used_index in dict.keys(), because tuple can be hashed and it's fast
        temp_candidate = set()

        while len(temp_candidate) < T:
            minimal_index = min(distance_dict, key=distance_dict.get)  # find the minimal distance
            ###calculate the invert index
```

```
            minimal_invert_index = tuple([sorted_matrix[i][minimal_index[i]] for i in range(len(minimal_index))])

            if minimal_invert_index in inverted_index.keys():  # use dict.key to check if used because it's fast
                add_set = tuple(inverted_index[minimal_invert_index])
                temp_candidate = temp_candidate.union(add_set)  # add to temp_candidate

            distance_dict.pop(minimal_index)  # delete this cell after used
            used_index[minimal_index] = True  # add the used cell into used_index

            for one in one_list:
                new_index = tuple(one + list(minimal_index)) # get the neighborhood cell index
                if new_index not in used_index.keys() and max(new_index) < 256: # check if used or out of index
                    distance_dict[new_index] = distance(index_conv(new_index, sorted_matrix), dis_matrix) # if not, add it

        candidates.append(temp_candidate)

    return candidates
```

The implementation of part 1 already written on the annotations.

About extended the algorithm 3.1 to a more general case with P>2:

Just extend the index length, if p == 4, the first index == [0,0,0,0], and the other work will check every dim in thid index.

About how you efficiently retrieve the candidates:

Store most of data in tuple as key in dict because it can be hashed and search it in dict will be fast and efficiently in O(1) time