# Risk and Logistics Assignment 2

## 0. Preparation

In [1]:
```
import numpy as np
import pandas as pd
# import xpress
import timeit
```

In [2]:
```
order = pd.read_csv("OrderMatrix.csv")
order = order['Order;Item 1;Item 2;Item 3'].str.split(";", expand=True).drop(0, axis = 1).astype(int)
order = order.rename(columns={1:'Item 1', 2:'Item 2', 3:'Item 3'})
order.index = order.index + 1

dist = pd.read_csv("DistanceMatrix.csv")
dist = dist.iloc[:,0].str.split(";", expand=True).drop(0, axis = 1).astype(int)
# index 97 means entrance
dist.index = dist.index + 1

allocation = pd.read_csv("CurrentAllocation.csv")
allocation = allocation.iloc[:,0].str.split(";", expand=True).drop(0, axis = 1).astype(int)
allocation = allocation.rename(columns={1:'Group'})
allocation.index = allocation.index + 1
```

## 1. Total Walking Distance

In [3]:
```
def dist_two_items(shelf1, shelf2, dist):
    '''
    Function to calculate distance when a picker go to two shelves to collect products
    '''
    distance = dist.loc[97, shelf1] + dist.loc[97, shelf2] + dist.loc[shelf1, shelf2]
    return distance

def dist_three_items(shelf1, shelf2, shelf3, dist):
    '''
    Function to calculate the shortest distance when a picker go to three shelves to collect products
    '''
    # Calculate total distances in three possible conditions
    dist1 = dist.loc[97, shelf1] + dist.loc[shelf1, shelf2] + dist.loc[shelf2, shelf3] + dist.loc[97, shelf3]
    dist2 = dist.loc[97, shelf1] + dist.loc[shelf1, shelf3] + dist.loc[shelf3, shelf2] + dist.loc[97, shelf2]
    dist3 = dist.loc[97, shelf2] + dist.loc[shelf2, shelf1] + dist.loc[shelf1, shelf3] + dist.loc[97, shelf3]
    # Return the shortest one
    distance = min(dist1, dist2, dist3)
    return distance
```

In [4]:
```
def walking_dist(order, dist, allocation):
    '''
    Calculate the total walking distance to pick up items in all orders
    '''
    # Product groups stored in two different shelves
    two_shelves = allocation[allocation.duplicated()]['Group']
    tot_dist = 0
    # Orders with ony one item
    order1 = order[(order['Item 1'] != 0) & (order['Item 2'] == 0)]
    # Orders with two items
    order2 = order[(order['Item 2'] != 0) & (order['Item 3'] == 0)]
    # Orders with three items
    order3 = order[order['Item 3'] != 0]

    # Loop over every order including one item
    for i in order1.index:
        group = order1['Item 1'][i]
        shelf = allocation[allocation['Group'] == group].index.values
        if len(shelf) == 1:
            tot_dist += dist.loc[97, shelf[0]]*2
        else:
            tot_dist += min(dist.loc[97, shelf[0]]*2, dist.loc[97, shelf[1]]*2)

    # Loop over every order including two items
    for j in order2.index:
        group1 = order2['Item 1'][j]
        group2 = order2['Item 2'][j]
        shelf1 = allocation[allocation['Group'] == group1].index.values
        shelf2 = allocation[allocation['Group'] == group2].index.values
        # Check if any group is stored in different shelves
        if (len(shelf1) == 1) & (len(shelf2) == 1):
            tot_dist += dist_two_items(shelf1[0], shelf2[0], dist)
        elif (len(shelf1) == 2) & (len(shelf2) == 2):
            tot_dist += min(dist_two_items(shelf1[0], shelf2[0], dist),
                            dist_two_items(shelf1[0], shelf2[1], dist),
                            dist_two_items(shelf1[1], shelf2[0], dist),
                            dist_two_items(shelf1[1], shelf2[1], dist))
        elif len(shelf1) == 2:
            tot_dist += min(dist_two_items(shelf1[0], shelf2[0], dist),
                            dist_two_items(shelf1[1], shelf2[0], dist))
        else:
```

```python
            tot_dist += min(dist_two_items(shelf1[0], shelf2[0], dist),
                            dist_two_items(shelf1[0], shelf2[1], dist))

    # Loop over every order including three items
    for k in order3.index:
        group1 = order3['Item 1'][k]
        group2 = order3['Item 2'][k]
        group3 = order3['Item 3'][k]
        shelf1 = allocation[allocation['Group'] == group1].index.values
        shelf2 = allocation[allocation['Group'] == group2].index.values
        shelf3 = allocation[allocation['Group'] == group3].index.values
        # Check if any group is stored in different shelves
        if (len(shelf1) == 1) & (len(shelf2) == 1) & (len(shelf3) == 1):
            tot_dist += dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist)

        elif (len(shelf1) == 1) & (len(shelf2) == 1):
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[0], shelf3[1], dist))

        elif (len(shelf1) == 1) & (len(shelf3) == 1):
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[1], shelf3[0], dist))

        elif (len(shelf2) == 1) & (len(shelf3) == 1):
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[1], shelf2[0], shelf3[0], dist))

        elif len(shelf1) == 1:
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[1], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[0], shelf3[1], dist),
                            dist_three_items(shelf1[0], shelf2[1], shelf3[1], dist))

        elif len(shelf2) == 1:
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[1], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[0], shelf3[1], dist),
                            dist_three_items(shelf1[1], shelf2[0], shelf3[1], dist))

        elif len(shelf3) == 1:
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[1], shelf2[0], shelf3[0], dist),
```

```python
                            dist_three_items(shelf1[0], shelf2[1], shelf3[0], dist),
                            dist_three_items(shelf1[1], shelf2[1], shelf3[0], dist))

        else:
            tot_dist += min(dist_three_items(shelf1[0], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[0], shelf3[1], dist),
                            dist_three_items(shelf1[0], shelf2[1], shelf3[0], dist),
                            dist_three_items(shelf1[0], shelf2[1], shelf3[1], dist),
                            dist_three_items(shelf1[1], shelf2[0], shelf3[0], dist),
                            dist_three_items(shelf1[1], shelf2[0], shelf3[1], dist),
                            dist_three_items(shelf1[1], shelf2[1], shelf3[0], dist),
                            dist_three_items(shelf1[1], shelf2[1], shelf3[1], dist))

    tot_dist = 3*tot_dist
    return tot_dist
```

In [5]:

```python
start = timeit.default_timer()

dist1 = walking_dist(order, dist, allocation)
print('The total walking distance is ', dist1)

stop = timeit.default_timer()

print('Run time: ', stop - start, 's')
```

```
The total walking distance is  361818
Run time:  0.7294891669999999 s
```

## 2. Random Start

In [6]:

```python
#Q2
# most commonly ordered items
value_counts = order[['Item 1', 'Item 2', 'Item 3']].apply(pd.Series.value_counts).sum(axis=1).drop(0)
top_6_values = value_counts.nlargest(6).index.tolist()
group96 = np.concatenate([np.arange(1, 91), top_6_values])
allocation_rs = allocation.copy()

#Random start heuristic
def RandomStart (iteration, group, order, dist, allocation_rs):
    dist_min = float('inf')

    for k in range(iteration):
        group_rdm = np.random.permutation(group96)
        allocation_rs.loc[:, 'Group'] = group_rdm
        dist_k = walking_dist(order, dist, allocation_rs)

        if dist_k < dist_min:
            dist_min = dist_k
            allocation_final = allocation_rs

    return allocation_final,dist_min

start = timeit.default_timer()

RS2 = RandomStart(20, group96, order, dist, allocation_rs)
allocation_rs2 = RS2[0]
dist_rs2 = RS2[1]

# print("The final allocation for items after random start heuristic: ", allocation_rs2)
print("Total walking distance after random start heuristic: ", dist_rs2)
stop = timeit.default_timer()

print('Run time: ', round(stop - start,1), 's')
```

```
Total walking distance after random start heuristic:  339642
Run time:  15.8 s
```

# 3.Local Search Heuristic

In real life, we often order related products together, such as a mobile phone and a charger. In this case, the two products are likely to appear in the same order, so placing them on adjacent shelves may reduce the total walking distance traveled by the picker.

We attempt to use the frequency of the two products appearing in the same order to measure their correlation and then divide the products into n clusters based on the relation. Interchange heuristic is implemented to let products in the same cluster stored closer.

## Allocate 90 products into different clusters based on their relationship in 'order.csv'

In [7]:
```
def cluster(Order,clusterNum):
    '''
    Function tries to partition 90 products into different clusters based on their relationships
    To measure their relationship, we calculate how many times 2 different products are in a same order
    '''
    if 90%clusterNum != 0:
        raise ZeroDivisionError("Cannot choose this cluster number")
    # calculate the relationship between each products based on Order dataset
    OrderDistance = np.zeros((90,90))
    for i in range(0,len(Order)):
        UniOrder = Order.iloc[i][Order.iloc[i]!=0][0:].tolist()
        itemNum = len(UniOrder)

        for j in range(0,itemNum):
            thisItem = UniOrder[j]
            OtherItems = UniOrder[:j] + UniOrder[j+1:]
            for t in range(0,len(OtherItems)):
                OrderDistance[thisItem-1,OtherItems[t]-1] += 1

    # calculate every products' relational degree with others
    row_sums = np.sum(OrderDistance, axis=1)
    last_performance = 0

    # random initialize n center
    center = np.random.randint(0,90,size = clusterNum)


    while True:
        # initialize group and a grouped vector to store item index which are already be clustered
        groups = np.zeros((clusterNum,int(90/clusterNum)))
        grouped = []
        # clustering
        for i in range(0,clusterNum):
            # initialize and find a vector to store other itmes relation to this item
            compare = []
            compare = OrderDistance[center[i]]

            # sort the relation (high to low)
            compare_sort_index = np.argsort(compare)[::-1]

            # delete center's index and grouped index
```

```python
            no_center = compare_sort_index[~np.isin(compare_sort_index, center)]
            no_grouped = no_center[~np.isin(no_center, grouped)]

            # allocate item who have strong relational with this center a group
            groups[i] = np.append(no_grouped[:int(90/clusterNum)-1],center[i])

            # recording groupoed items
            grouped.append(np.append(no_grouped[:int(90/clusterNum)-1],center[i]))

        # measure cluster
        this_performance = 0
        for i in range(0,clusterNum):
            row = int(center[i])
            sumption = 0
            for j in range(0,int(90/clusterNum)):
                col = int(groups[i,j])
                sumption = sumption + OrderDistance[row,col]
            this_performance += sumption

        # if we need cluster again
        if this_performance > last_performance:
            # print(this_performance)
            # print(center)
            last_groups = groups
            last_center = center
            last_performance = this_performance

            # refresh center: find the item who has strongest relational in its group
            # loop over every group
            for i in range(0,clusterNum):
                group_row_sums = np.zeros(int(90/clusterNum))
                # loop over every item in i-th group
                for j in range(0,int(90/clusterNum)):
                    uni_item_sum = 0
                    row = int(groups[i,j])
                    other = np.delete(groups[i], j)
                    for t in other:
                        uni_item_sum = uni_item_sum + OrderDistance[row,int(t)]
                    group_row_sums[j] = uni_item_sum
                center[i] = groups[i,np.where(group_row_sums == max(group_row_sums))[0][0]]
        else:
            break
```

```
    for it in last_groups:
        it += 1
    return last_groups
```

# Interchange heuristic based on clusters

In [8]:
```python
def clusterchange(CA,direction,neighbor_define,dist,groups):
    ini = CA
    last_Performance = walking_dist(order,dist,CA)
    CA = CA["Group"].values.tolist()
    last_allocation = CA
    lower_distance = neighbor_define[0]
    upper_distance = neighbor_define[1]

    record = []
    group_record = np.zeros((96,96))
    Performance_record = []

    ## loop over every shelf
    for i in direction:
        current_item = CA[i] # the current item stored in the shelf
        ## check if any product is stored in this shelf
        if current_item == 0:
            continue
        current_group = np.where(groups == current_item)[0]

        # find the adjacent shelves which not include the entrance and i-th shelf
        neighbor = np.array(np.where((dist.iloc[i]<=upper_distance)&(dist.iloc[i]>=lower_distance))).transpose()
        neighbor = neighbor[(neighbor != i) & (neighbor != 96)]
        neighbor_item = []
        neighbor_group = []
        # find the product stored in adjacent shelves and cluster of it
        for j in neighbor:
            neighbor_item.append(CA[int(j)])
            neighbor_group.append(np.where(groups == neighbor_item[-1])[0])

        # find products in the same cluster but not stored in adjacent shelves
        group_member = groups[int(current_group)][~np.in1d(groups[int(current_group)], neighbor_item)]
        group_member = group_member[~np.in1d(group_member, current_item)]

        # find shleves 'group_member' are stored in
        group_member_shelf = []
        for j in group_member:
            group_member_shelf.append(CA.index(j))

        # collect the distance between each shelf in 'group_member_shelf' and the i-th shelf
        group_member_dist = []
```

```
        for j in group_member_shelf:
            group_member_dist.append(dist.iloc[i][j+1])

        # Sort distance in descending order and return the corresponding index
        member_dist_sortindex = np.argsort(group_member_dist)[::-1].tolist()

        change_item_in = 0
        change_item_out = 0
        # loop over every adjacent shelf
        for j in range(len(neighbor)):
            # check if the product stored in j-th neighbor is in the same cluster as that stored in i-th shelf
            if ((len(neighbor_group[j]) > 0) and (neighbor_group[j] == current_group)):
                continue
            else:
                # The farthest product in the same cluster from i-th shelf
                change_shelf = member_dist_sortindex.pop(0)
                change_item_in = CA[int(change_shelf)]
                # exchange shelves
                change_item_out = CA[neighbor[j]]
                CA[neighbor[j]] = change_item_in
                CA[int(change_shelf)] = change_item_out

        this_allocation = pd.DataFrame({'Group': CA})
        this_allocation.index = this_allocation.index + 1
        this_Performance = walking_dist(order, dist, this_allocation)

        # if the total distance decrease after exchange
        if this_Performance < last_Performance:
            last_Performance = this_Performance
            last_allocation = CA
            record.append(i)
            Performance_record.append(this_Performance)
        else:
            CA = last_allocation
            this_Performance = last_Performance
        group_record[i] = CA

    if len(Performance_record) == 0:
        return  walking_dist(order,dist,ini),ini
    else:
        min_Per_index = Performance_record.index(min(Performance_record))
        min_i = record[min_Per_index]
```

```
        final_allocation = pd.DataFrame({"Group":group_record[min_i]})
        final_allocation.index = final_allocation.index + 1

        return min(Performance_record),final_allocation
```

## An Example to Use Cluster Finding Better Solution

1. obtaining a different clusters
2. determine how to define neighbor(how long the distance is)
3. determine the changing center's range/direction
4. doing cluster change

In [9]:
```
groups3 = cluster(order,3)
print(groups3)
```

```
[[59. 42.  9. 82. 33. 40. 38.  1. 36. 15. 37. 77. 50. 20. 34. 18. 72.  4.
  61. 84.  7. 48. 74. 49.  5. 28. 64. 63. 85. 52.]
 [76. 43.  6. 29. 75. 78.  8. 35. 17. 30. 89.  3. 47. 10. 12. 31. 90. 51.
  54. 57. 58. 86. 19. 21. 23. 25. 80. 81. 66. 88.]
 [32. 67. 14. 26. 44. 87. 62. 70. 60. 13.  2. 11. 24. 16. 45. 27. 83. 79.
  73. 71. 69. 68. 65. 56. 55. 53. 46. 41. 39. 22.]]
```

In [10]:
```
totaldist0 = dist1
allo0 = allocation
print("Total distance before is",totaldist0)

start = timeit.default_timer()
totaldis1_1,allo1_1 = clusterchange(allo0,range(0,50,1),neighbor_define = [0,1], dist = dist, groups = groups3)
totaldis1_2,allo1_2 = clusterchange(allo1_1,range(95,50,-1),neighbor_define = [0,1],dist = dist, groups = groups3)
stop = timeit.default_timer()
print('Run time: ', stop - start, 's')
print("Total distance is",totaldis1_2)
```

```
Total distance before is 361818
Run time:  63.795832334 s
Total distance is 349062
```

In [11]:
```
groups9 = cluster(order,9)
print(groups9)
```

```
[[ 9. 82. 33. 40. 38.  1. 36. 88. 77. 52.]
 [74. 50. 72. 84. 76. 49. 48.  7. 63. 42.]
 [28. 18.  5. 55. 64. 43. 29. 70. 78. 59.]
 [34. 20. 22. 14. 25. 26. 31. 30. 21. 15.]
 [17. 83. 58. 67. 61. 62. 12. 65. 10. 37.]
 [ 3. 85.  6. 13. 41. 89. 47. 35.  4. 32.]
 [66. 81. 23. 57. 24. 11. 80. 87. 75.  8.]
 [39.  2. 27. 19. 16. 90. 68. 69. 71. 46.]
 [56. 44. 60. 45. 79. 73. 54. 53. 51. 86.]]
```

In [12]:
```
# totaldist0 = walking_dist(order, dist, allocation)
# allo0 = allocation
print("Total distance before is",totaldis1_2)

start = timeit.default_timer()
totaldis2_1,allo2_1 = clusterchange(allo1_2,range(0,50,1),neighbor_define = [0,1],dist = dist, groups = groups9)
totaldis2_2,allo2_2 = clusterchange(allo2_1,range(95,50,-1),neighbor_define = [0,1],dist = dist, groups = groups9)
stop = timeit.default_timer()
print('Run time: ', stop - start, 's')
print("Total distance is", totaldis2_1)
```

```
Total distance before is 349062
Run time:  64.41828054100002 s
Total distance is 349062
```

## Implement the heuristic for the shelf allocation given in csv files

In [13]:
```
start_dist = dist1
print("Total distance before interchange heuristic is", start_dist)
allo3_0 = allocation
better_dist = start_dist - 1
cluster_num = [10,3]
n = 0
itera = 0
start = timeit.default_timer()
while better_dist < start_dist:
    itera = itera + 1
#     print(itera)
    start_dist = better_dist

    groups_q3 = cluster(order,cluster_num[n])
    totaldis3_1,allo3_1 = clusterchange(allo3_0,range(0,50,1),neighbor_define = [0,1],dist = dist,groups = groups_q3)
#     print(totaldis3_1)
    totaldis3_2,allo3_2 = clusterchange(allo3_1,range(95,50,-1),neighbor_define = [0,1],dist = dist,groups = groups_q3)
#     print(totaldis3_2)

    allo3_0 = allo3_2
    better_dist = totaldis3_2
    if n < 1:
        n = n+1
    else:
        n = 0
stop = timeit.default_timer()

print("Total distance after interchange heuristic is", start_dist)
print('Run time: ', stop - start, 's')
```

```
Total distance before interchange heuristic is 361818
Total distance after interchange heuristic is 319746
Run time:  446.62663299999997 s
```

### Implement the heuristic for the shelf allocation provided by previous question

In [14]:

```
start_dist = dist_rs2
print("Total distance before interchange heuristic is", start_dist)
allo3_0_ = allocation_rs2
better_dist = start_dist - 1
cluster_num = [10,3]
n = 0
itera = 0
start = timeit.default_timer()
while better_dist < start_dist:
    itera = itera + 1
#     print(itera)
    start_dist = better_dist

    groups_q3_ = cluster(order,cluster_num[n])
    totaldis3_1_,allo3_1_ = clusterchange(allo3_0_,range(0,50,1),neighbor_define = [0,1],dist = dist,groups = groups_q3_
#     print(totaldis3_1_)
    totaldis3_2_,allo3_2_ = clusterchange(allo3_1_,range(95,50,-1),neighbor_define = [0,1],dist = dist,groups = groups_q
#     print(totaldis3_2_)

    allo3_0_ = allo3_2_
    better_dist = totaldis3_2_
    if n < 1:
        n = n+1
    else:
        n = 0
stop = timeit.default_timer()

print("Total distance after interchange heuristic is",start_dist)
print('Run time: ', stop - start, 's')
```

```
Total distance before interchange heuristic is 339642
Total distance after interchange heuristic is 339641
Run time:  74.72714791699991 s
```

# 4 Improved Layout

## Calculate Distance

We consider the map as a Cartesian coordinate system so every shelf and entrance has a coordinate. Based on this, breadth-first search algorithm is implemented to calculate the distance.

In [15]:
```python
from queue import Queue
def init_data():
    '''
    Define the improved layout
    '''
    ware_map = np.zeros((30, 13), dtype=int)
    ware_list = [0 for i in range(0, 98)]

    # define the entrance
    ware_map[0, 3] = 97
    ware_list[97] = (0, 3)

    # define the layout of all shelves
    for i in range(1,18):
        ware_map[i, 0] = i
        ware_map[i, 8] = i + 73
        ware_list[i] = (i, 0)
        ware_list[i + 73] = (i, 8)
        if i <= 4:
            ware_map[i, 2] = i + 17
            ware_map[i, 3] = i + 31
            ware_map[i, 5] = i + 45
            ware_map[i, 6] = i + 59
            ware_list[i + 17] = (i, 2)
            ware_list[i + 31] = (i, 3)
            ware_list[i + 45] = (i, 5)
            ware_list[i + 59] = (i, 6)
        elif 6 <= i <= 10:
            ware_map[i, 2] = i + 16
            ware_map[i, 3] = i + 30
            ware_map[i, 5] = i + 44
            ware_map[i, 6] = i + 58
            ware_list[i + 16] = (i, 2)
            ware_list[i + 30] = (i, 3)
            ware_list[i + 44] = (i, 5)
            ware_list[i + 58] = (i, 6)
        elif 12 <= i <= 16:
            ware_map[i, 2] = i + 15
            ware_map[i, 3] = i + 29
            ware_map[i, 5] = i + 43
            ware_map[i, 6] = i + 57
```

```
            ware_list[i + 15] = (i, 2)
            ware_list[i + 29] = (i, 3)
            ware_list[i + 43] = (i, 5)
            ware_list[i + 57] = (i, 6)
    for j in range(1,7):
        ware_map[18, j] = j + 90
        ware_list[j + 90] = (18, j)

    return ware_map, ware_list
```

In [16]:
```python
def bfs(ware_map, src, dst):
    '''
    Implement a breadth-first search algorithm to find the shortest distance between pairs of locations
    '''
    dis = -np.ones((ware_map.shape[0], ware_map.shape[1]), dtype=int)
    # the possible movements on the map
    dx = [-1, 0, 1, 0]
    dy = [0, 1, 0, -1]

    q = Queue()
    q.put(src)
    dis[src] = -1
    while not q.empty():
        pos = q.get()
        for i in range(0, 4):
            x = pos[0] + dx[i]
            y = pos[1] + dy[i]
            # check if neighbor is within the boundaries and has not been visited
            if 0 <= x < ware_map.shape[0] and 0 <= y < ware_map.shape[1] and dis[x, y] == -1:
                if x == dst[0] and y == dst[1]:
                    # if this is the entrance
                    if ware_map[x, y] == 97:
                        return dis[pos] + 1
                    elif ware_map[pos] == 0:
                        return dis[pos]
                if ware_map[x, y] == 0 or ware_map[x, y] == 97:
                    dis[x, y] = dis[pos] + 1
                    q.put((x, y))
    return -1


def search(ware_map, ware_list):
    dis_mat = np.zeros((98, 98), dtype=int)
    # loop over every location as starting point
    for i in range(1, 98):
        src = ware_list[i]
        # loop over every location as destination
        for j in range(1, 98):
            if i == j:
                continue
            dst = ware_list[j]
```

```python
                dis_mat[i, j] = bfs(ware_map, src, dst)
    return dis_mat
```

In [17]:

```python
ware_map, ware_list = init_data()
dis_mat = search(ware_map, ware_list)
```

In [18]:

```python
dist_new = pd.DataFrame(dis_mat)
dist_new = dist_new.drop(0,axis =1).drop(0,axis=0)
# the new distance matrix
dist_new
```

Out[18]:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 20 | 21 | 22 | 16 | 17 | 18 | 19 | 20 | 21 | 3 |
| **2** | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | 19 | 20 | 21 | 15 | 16 | 17 | 18 | 19 | 20 | 4 |
| **3** | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 18 | 19 | 20 | 14 | 15 | 16 | 17 | 18 | 19 | 5 |
| **4** | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | 17 | 18 | 19 | 13 | 14 | 15 | 16 | 17 | 18 | 6 |
| **5** | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | ... | 16 | 17 | 18 | 12 | 13 | 14 | 15 | 16 | 17 | 7 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **93** | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | ... | 6 | 5 | 4 | 2 | 1 | 0 | 1 | 2 | 3 | 19 |
| **94** | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | ... | 5 | 4 | 3 | 3 | 2 | 1 | 0 | 1 | 2 | 18 |
| **95** | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | ... | 4 | 3 | 2 | 4 | 3 | 2 | 1 | 0 | 1 | 19 |
| **96** | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | ... | 3 | 2 | 1 | 5 | 4 | 3 | 2 | 1 | 0 | 20 |
| **97** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... | 18 | 19 | 20 | 18 | 19 | 18 | 17 | 18 | 19 | 0 |

In [19]:
```
#Q4 Random start
start = timeit.default_timer()
RS = RandomStart(20, group96, order, dist_new, allocation_rs)
allocation_rs4 = RS[0]
dist_rs4 = RS[1]
# print("The final allocation for items after random start heuristic: ", allocation_rs4)
print("Total walking distance after random start heuristic: ", dist_rs4)
stop = timeit.default_timer()

print('Run time: ', round(stop - start,1), 's')
```

```
Total walking distance after random start heuristic:  247770
Run time:  15.5 s
```

In [20]:

```
start_dist = dist_rs4
print("Total distance before interchange heuristic is", start_dist)
allo4_0 = allocation_rs4
better_dist = start_dist - 1
cluster_num = [10,3]
n = 0
itera = 0
start = timeit.default_timer()
while better_dist < start_dist:
    itera = itera + 1
#     print(itera)
    start_dist = better_dist

    groups_q4 = cluster(order,cluster_num[n])
    totaldis4_1,allo4_1 = clusterchange(allo4_0,range(0,50,1),neighbor_define = [0,1],dist = dist_new,groups = groups_q4
#     print(totaldis4_1)
    totaldis4_2,allo4_2 = clusterchange(allo4_1,range(95,50,-1),neighbor_define = [0,1],dist = dist_new,groups = groups_
#     print(totaldis4_2)

    allo4_0 = allo4_2
    better_dist = totaldis4_2
    if n < 1:
        n = n+1
    else:
        n = 0
stop = timeit.default_timer()
print('Run time: ', stop - start, 's')
print("Total distance after interchange heuristic is",start_dist)
```

```
Total distance before interchange heuristic is 247770
Run time:  74.39835925 s
Total distance after interchange heuristic is 247769
```