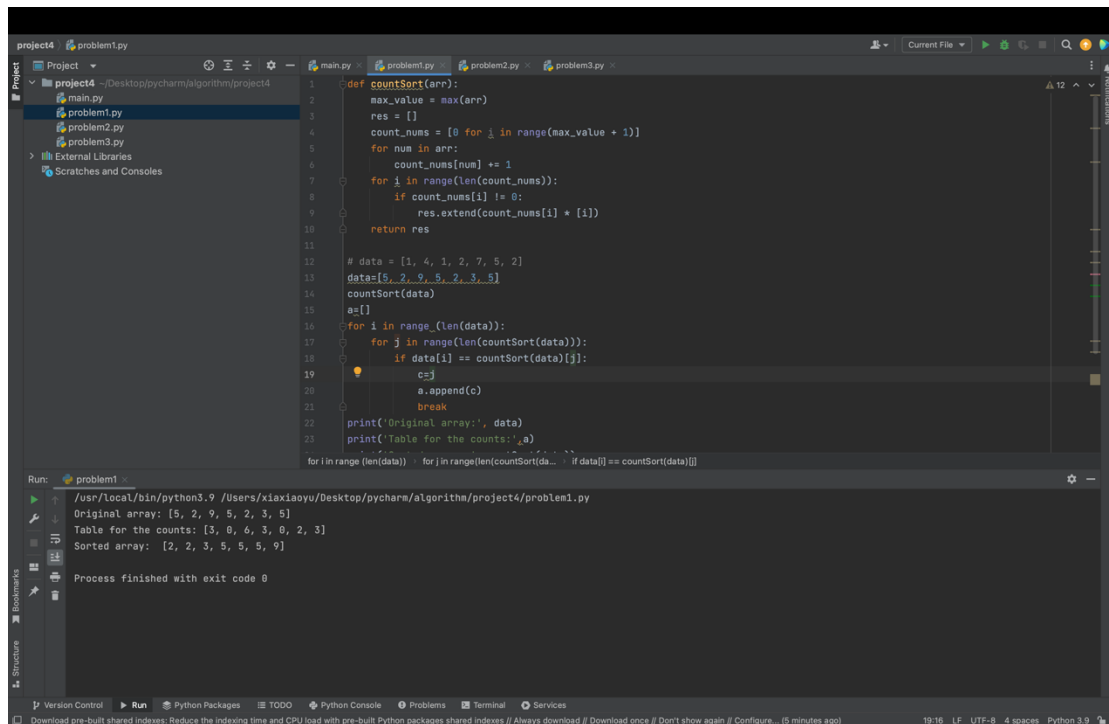


## Problem 1:

### Case1:



The screenshot shows the PyCharm IDE with a project named 'project4'. The file explorer on the left shows a directory structure with 'main.py', 'problem1.py', 'problem2.py', and 'problem3.py'. The 'main.py' file is open in the editor, showing a counting sort implementation. The code defines a 'countSort' function and uses it to sort an array. The output of the program is displayed in the 'Run' console at the bottom.

```
def countSort(arr):
    max_value = max(arr)
    res = []
    count_nums = [0 for i in range(max_value + 1)]
    for num in arr:
        count_nums[num] += 1
    for i in range(len(count_nums)):
        if count_nums[i] != 0:
            res.extend(count_nums[i] * [i])
    return res

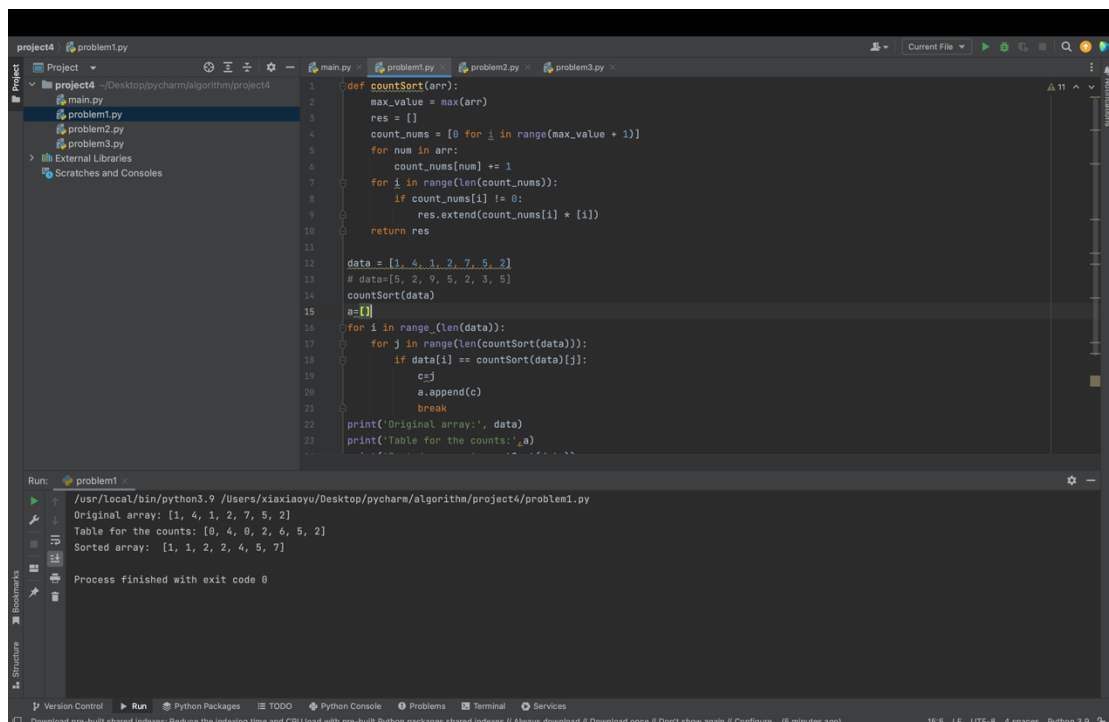
# data = [1, 4, 1, 2, 7, 5, 2]
data = [5, 2, 9, 5, 2, 3, 5]
countSort(data)
a=[]
for i in range(len(data)):
    for j in range(len(countSort(data))):
        if data[i] == countSort(data)[j]:
            c=j
            a.append(c)
            break
print('Original array:', data)
print('Table for the counts:', a)
for i in range(len(data)):
    for j in range(len(countSort(data))):
        if data[i] == countSort(data)[j]:
```

Run: problem1 x

```
/usr/local/bin/python3.9 /Users/xiaiaoyu/Desktop/pycharm/algorithm/project4/problem1.py
Original array: [5, 2, 9, 5, 2, 3, 5]
Table for the counts: [3, 0, 6, 3, 0, 2, 3]
Sorted array: [2, 2, 3, 5, 5, 5, 9]

Process finished with exit code 0
```

### Case2:



The screenshot shows the PyCharm IDE with the same project 'project4'. The 'main.py' file is open, showing a counting sort implementation. The output of the program is displayed in the 'Run' console at the bottom.

```
def countSort(arr):
    max_value = max(arr)
    res = []
    count_nums = [0 for i in range(max_value + 1)]
    for num in arr:
        count_nums[num] += 1
    for i in range(len(count_nums)):
        if count_nums[i] != 0:
            res.extend(count_nums[i] * [i])
    return res

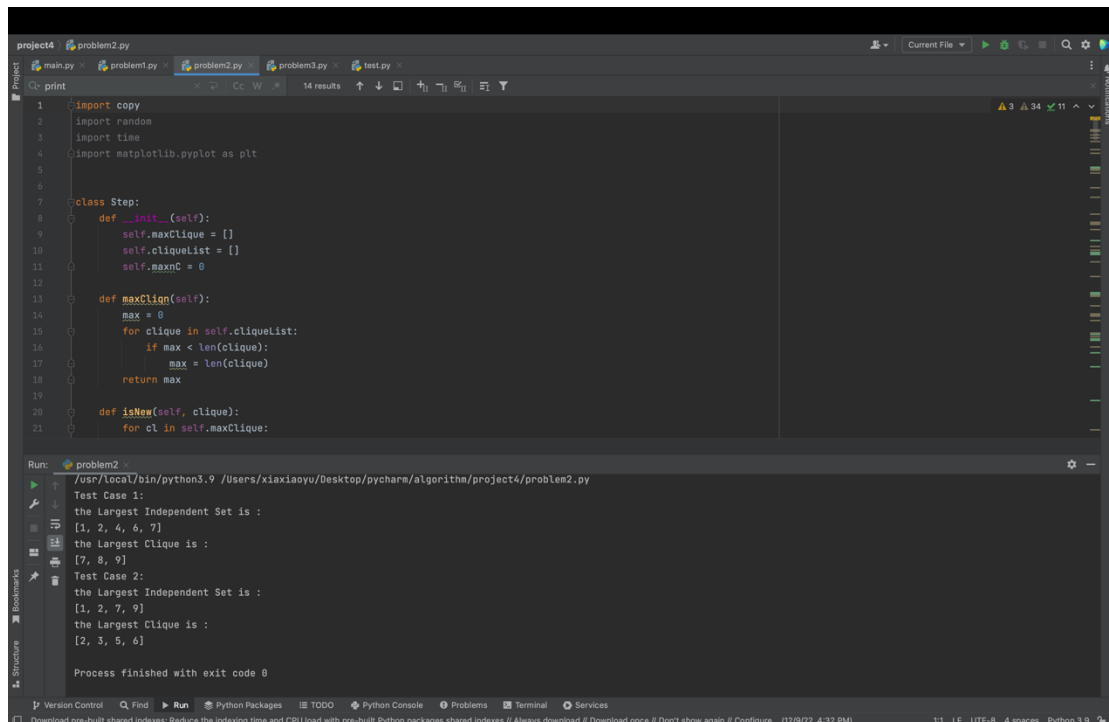
data = [1, 4, 1, 2, 7, 5, 2]
# data=[5, 2, 9, 5, 2, 3, 5]
countSort(data)
a=[]
for i in range(len(data)):
    for j in range(len(countSort(data))):
        if data[i] == countSort(data)[j]:
            c=j
            a.append(c)
            break
print('Original array:', data)
print('Table for the counts:', a)
```

Run: problem1 x

```
/usr/local/bin/python3.9 /Users/xiaiaoyu/Desktop/pycharm/algorithm/project4/problem1.py
Original array: [1, 4, 1, 2, 7, 5, 2]
Table for the counts: [0, 4, 0, 2, 6, 5, 2]
Sorted array: [1, 1, 2, 2, 4, 5, 7]

Process finished with exit code 0
```

## Problem2:



```
1 import copy
2 import random
3 import time
4 import matplotlib.pyplot as plt
5
6
7 class Step:
8     def __init__(self):
9         self.maxClique = []
10        self.cliqueList = []
11        self.maxnC = 0
12
13    def maxClique(self):
14        max = 0
15        for clique in self.cliqueList:
16            if max < len(clique):
17                max = len(clique)
18        return max
19
20    def isNew(self, clique):
21        for cl in self.maxClique:
```

Run: problem2 x  
/usr/local/bin/python3.9 /Users/xiaoya/Desktop/pycharm/algorithm/project4/problem2.py

Test Case 1:  
the Largest Independent Set is :  
[1, 2, 4, 6, 7]  
the Largest Clique is :  
[7, 8, 9]

Test Case 2:  
the Largest Independent Set is :  
[1, 2, 7, 9]  
the Largest Clique is :  
[2, 3, 5, 6]

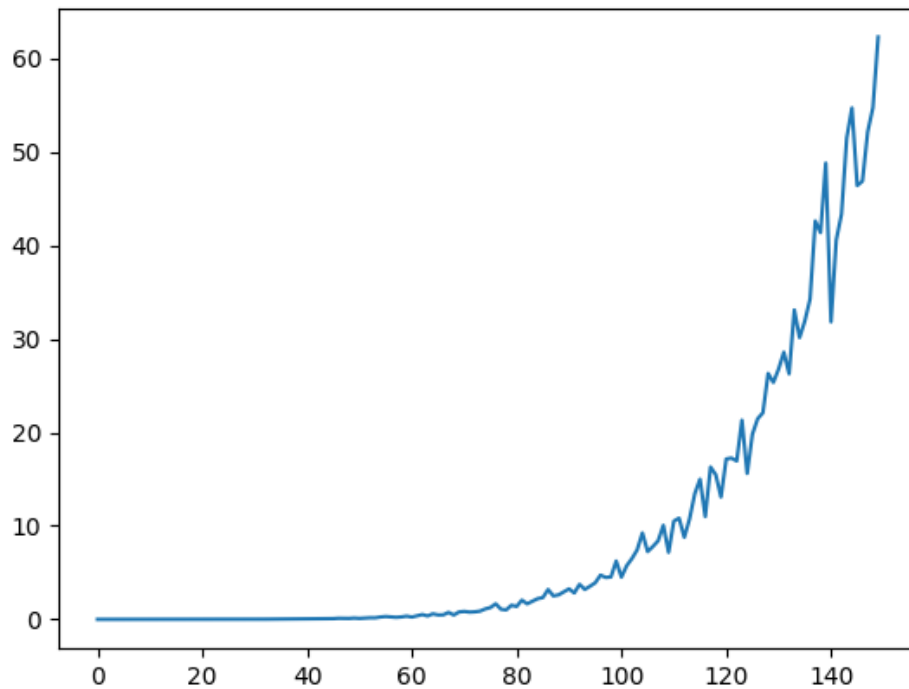
Process finished with exit code 0

Time complexity for both algorithms :  $n^4$

The largest independent set:

```
def process(self):
    n = len(self.V)
    solutions = {}
    for i in range(0, n):
        solutions[i] = Step()
    for v in self.V:
        a = []
        a.append(v)
        solutions[0].cliqueList.append(a)
        solutions[0].updateMaxClique()

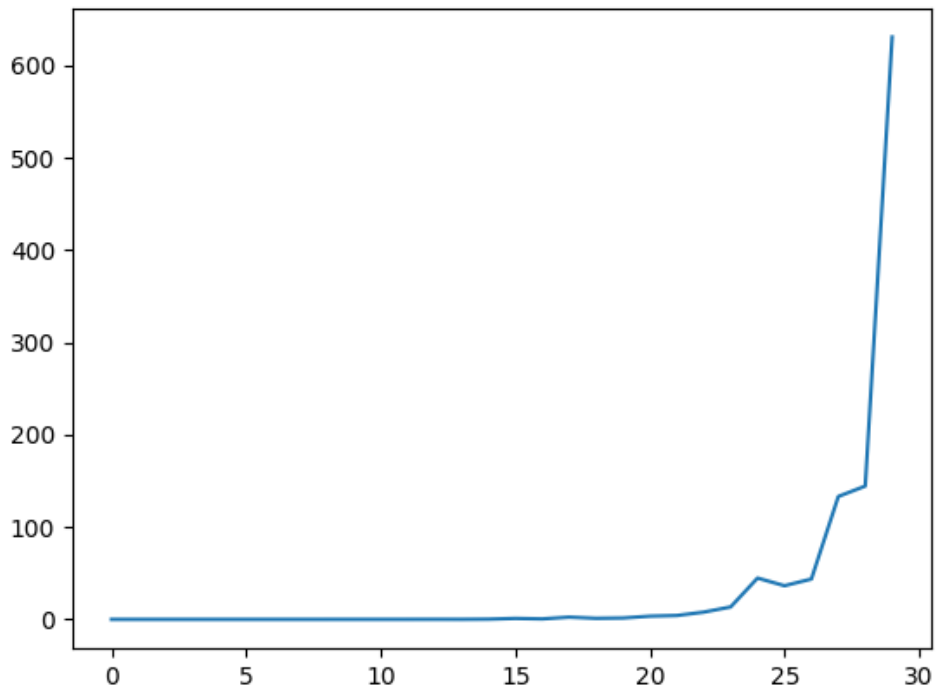
    for i in range(1, n):
        # cliqlist= solutions[i-1].maxClique
        preData = solutions[i - 1]
        cliqlist = preData.maxClique
        preMax = preData.maxnC
        for clique in cliqlist:
            for v in self.V:
                tempclique = copy.deepcopy(clique)
                if not v in tempclique:
                    # if s.isConnectedAll(tempclique,v):
                    if self.isConnectedOne(tempclique, v):
                        tempclique.append(v)
                        solutions[i].cliqueList.append(tempclique)
        solutions[i].updateMaxClique()
        if not len(solutions[i].maxClique):
            break
```



## The largest clique:

```
def process(self):
    n = len(self.V)
    solutions = {}
    for i in range(0, n):
        solutions[i] = Step()
    for v in self.V:
        a = []
        a.append(v)
        solutions[0].cliqueList.append(a)
        solutions[0].updateMaxClique()

    for i in range(1, n):
        # cliqList= solutions[i-1].maxClique
        preData = solutions[i - 1]
        cliqList = preData.maxClique
        preMax = preData.maxnC
        for clique in cliqList:
            for v in self.V:
                tempclique = copy.deepcopy(clique)
                if not v in tempclique:
                    if self.isConnectedAll(tempclique, v):
                        tempclique.append(v)
                        solutions[i].cliqueList.append(tempclique)
        solutions[i].updateMaxClique()
        if not len(solutions[i].maxClique):
            break
```

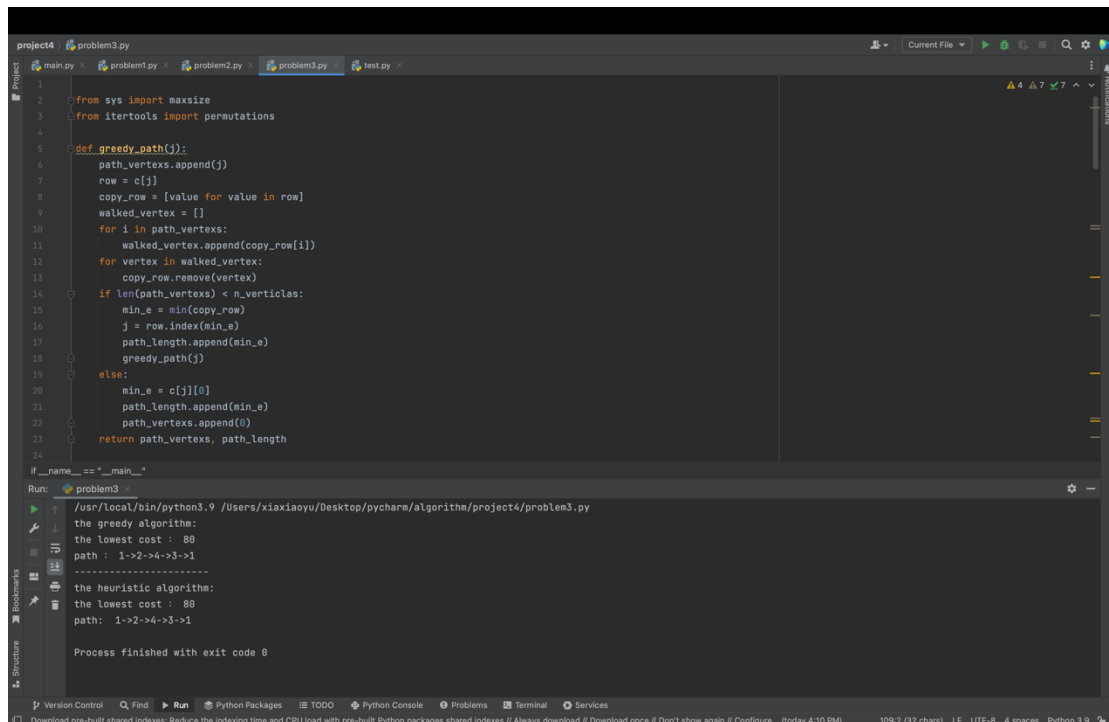


## Problem3:

### Case1:

```
project4 problem3.py
main.py problem1.py problem2.py problem3.py test.py
1
2 from sys import maxsize
3 from itertools import permutations
4
5 def greedy_path(j):
6     path_vertices.append(j)
7     row = c[j]
8     copy_row = [value for value in row]
9     walked_vertex = []
10    for i in path_vertices:
11        walked_vertex.append(copy_row[i])
12    for vertex in walked_vertex:
13        copy_row.remove(vertex)
14    if len(path_vertices) < n_vertices:
15        min_e = min(copy_row)
16        j = row.index(min_e)
17        path_length.append(min_e)
18        greedy_path(j)
19    else:
20        min_e = c[j][0]
21        path_length.append(min_e)
22        path_vertices.append(0)
23    return path_vertices, path_length
24
25 if __name__ == "__main__":
26     # Run: problem3
27     /usr/local/bin/python3.9 /Users/xiaxiaoyu/Desktop/pycharm/algorithm/project4/problem3.py
28
29 the greedy algorithm:
30 the lowest cost : 275
31 path : 1->4->2->5->3->1
32 -----
33 the heuristic algorithm:
34 the lowest cost : 268
35 path: 1->2->5->3->4->1
36
37 Process finished with exit code 0
```

## Case2:

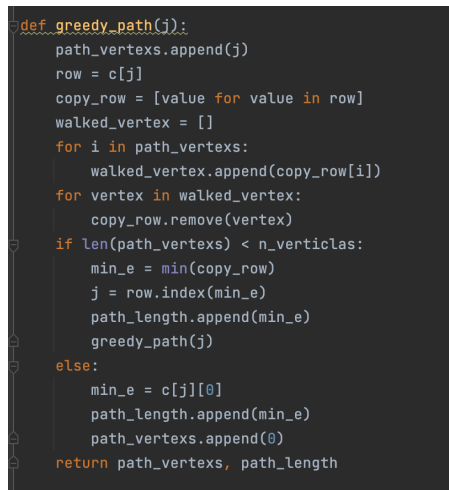


```
1 from sys import maxsize
2 from itertools import permutations
3
4 def greedy_path(j):
5     path_vertices.append(j)
6     row = c[j]
7     copy_row = [value for value in row]
8     walked_vertex = []
9     for i in path_vertices:
10         walked_vertex.append(copy_row[i])
11     for vertex in walked_vertex:
12         copy_row.remove(vertex)
13     if len(path_vertices) < n_vertices:
14         min_e = min(copy_row)
15         j = row.index(min_e)
16         path_length.append(min_e)
17         greedy_path(j)
18     else:
19         min_e = c[j][0]
20         path_length.append(min_e)
21         path_vertices.append(0)
22     return path_vertices, path_length
23
24 if __name__ == "__main__":
25     # Run: problem3
26     /usr/local/bin/python3.9 /Users/xiaoya/Desktop/pycharm/algorithm/project4/problem3.py
27     the greedy algorithm:
28     the lowest cost : 80
29     path : 1->2->4->3->1
30     -----
31     the heuristic algorithm:
32     the lowest cost : 80
33     path: 1->2->4->3->1
34
35     Process finished with exit code 0
```

compare the results of A and B and discuss how good the results of B are by calculating: in the first test case, the calculating result is  $260 / 275 = 0.945$ . In the second test case, the result is  $80 / 80 = 1$ . There is a difference in the results in the first test case because the TSP problem is an NP-hard problem so we cannot promise that every different algorithm, which is used in the same graph, could get the same answer.

the time complexity of each using Big O :

greedy algorithm:  $O(n^2)$ :



```
def greedy_path(j):
    path_vertices.append(j)
    row = c[j]
    copy_row = [value for value in row]
    walked_vertex = []
    for i in path_vertices:
        walked_vertex.append(copy_row[i])
    for vertex in walked_vertex:
        copy_row.remove(vertex)
    if len(path_vertices) < n_vertices:
        min_e = min(copy_row)
        j = row.index(min_e)
        path_length.append(min_e)
        greedy_path(j)
    else:
        min_e = c[j][0]
        path_length.append(min_e)
        path_vertices.append(0)
    return path_vertices, path_length
```

A heuristic algorithm:  $O(n^2)$ :

```
def heuristic(graph, s):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    min_path = maxsize
    next_permutation = permutations(vertex)
    for i in next_permutation:
        # store current Path weight(cost)
        current_pathweight = 0
        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        # update minimum
        min_path = min(min_path, current_pathweight)
    return min_path
```