PROJECT REPORT

Abdul Wasay	(i21-0834)	QUESTION 1
Hamza Tariq	(i21-0707)	QUESTION 2
Hassan Abbas	(i21-0507)	QUESTION 3

QUESTION 1: STRING

ASYMPTOTIC TIME COMPLEXITY ANALYSIS

CONTAINS FUNCTION:

1. for i from 0 to nums.size() - group.size(): **- O(n - m),** where n is the size of the nums array and m is the size of the group array.
2. found_all = true **- O(1)**
3. for j from 0 to group.size(): **- O(m),** where m is the size of the group array.
4. found = false **- O(1)**
5. for k from i to i + group.size(): **- O(m),** where m is the size of the group array.
6. if not visited[k] and nums[k] == group[j]: **- O(1)**
7. visited[k] = true **- O(1)**
8. found = true **- O(1)**
9. break **- O(1)**
10. if not found: **- O(1)**
11. found_all = false **- O(1)**
12. break **- O(1)**
13. if found_all: **- O(1)**
14. return true **- O(1)**
15. else: **- O(1)**
16. for k from i to i + group.size(): **- O(m),** where m is the size of the group array.
17. visited[k] = false **- O(1)**

18. return false - **O(1)**

- Therefore, the overall time complexity of the function is **O((n-m) * m^2).**

- **Best Case:** The best case time complexity of the contains function would be **O(m),** where m is the size of the group array. This would occur when the first m elements of the nums array match the elements of the group array, so the function can immediately return true without having to iterate over the rest of the nums array or change any values in the visited array.

MATCHING_STRING FUNCTION:

1. count = 0 **- O(1)**
2. for j = 0 to nums.size() - 1 **- O(n),** where n is the number of test cases.
3. count = 0 - O(1)
4. vector<bool>visited with size nums[j].size() initialized to false **- O(m),** where m is the size of the current test case.
5. result.clear() **- O(1)**
6. for i = 0 to groups.size() - 1 **- O(g),** where g is the number of groups.
7. if not contains(nums[j], groups[i], visited) **- O((n-m) * m^2),** where n is the size of the current test case and m is the size of the current group.
8. result.push_back(groups[i]) **- O(1)**
9. else **- O(1)**
10. count = count + 1 **- O(1)**
11. end if **- O(1)**
12. end for **- O(g)**
13. if count == groups.size() **- O(1)**

14. print "TRUE\n" **- O(1)**

15. else **- O(1)**

16. print "FALSE\n" **- O(1)**

17. end if - O(1)

⇨ Therefore, the overall time complexity of the given code is **O(ng(n-m)*m^2).**

❖ **Best Case:** The best case time complexity of the given function is when there is only one group and it matches with the input test case. In this case, the time complexity would be **O(n+m),** where n is the size of the test case and m is the size of the group.

READING_FILE FUNCTION:

1. open "p1_input.txt" for reading as infile **- O(1)**
2. done = false **- O(1)**
3. while not done: **- O(n)**
4. read a line from infile and store it in line **- O(m)**
5. group = [] **- O(1)**
6. num_list = [] **- O(1)**
7. ss = stringstream(line) **- O(1)**
8. while read a value from ss and store it in ingredient and not done: **- O(m)**
9. temp = "" **- O(1)**
10. flag = false **- O(1)**
11. counter = 0 **- O(1)**
12. for i from 0 to length of ingredient - 1: **- O(m)**
13. if ingredient[i] is 'T' and ingredient[i+1] is 'e' and ingredient[i+2] is 's' and ingredient[i+3] is 't': **- O(1)**

```
14.  done = true                        - O(1)
15.  break
16.  if ingredient[i] is "'":               - O(1)
17.  flag = true                        - O(1)
18.  counter = counter + 1              - O(1)
19.  if flag and ingredient[i] is not "'":   - O(1)
20.  append ingredient[i] to temp       - O(1)
21.  if counter is 2:                   - O(1)
22.  break
23.  append temp to group                    - O(1)
24.  if done:                                - O(1)
25.  ss_nums = stringstream(line)               - O(1)
26.  while read a value from ss_nums and store it in num_str: - O(m)
27.  temp = ""                            - O(1)
28.  flag = false                         - O(1)
29.  counter2 = 0                          - O(1)
30.  for i from 0 to length of num_str - 1:  - O(m)
31.  if num_str[i] is "'":            - O(1)
32.  flag = true                   - O(1)
33.  counter2 = counter2 + 1       - O(1)
34.  if counter2 is 2:               - O(1)
35.  break
36.  if flag and num_str[i] is not "'":   - O(1)
37.  append num_str[i] to temp       - O(1)
38.  if counter2 is not 0:              - O(1)
39.  append temp to num_list          - O(1)
40.  if num_list is not empty:              - O(1)
41.  append num_list to nums              - O(1)
```

42. if not done: **- O(1)**

43. append group to groups **- O(1)**

44. close infile **- O(1)**

⇨ Overall, the time complexity is **O(n + m)** where n is the total number of characters in the input file and m is the total number of characters in all the values on all lines.

❖ **Best Case:** The best case time complexity would occur when the input file "p1_input.txt" is empty or has only one line, which would result in the while loop being executed only once.In that case, the time complexity would be **O(1)**

TOTAL TIME COMPLEXITY OF ALGORITHM

O(g(n-m)*m^2)

Where g=no. of groups

n=number of elements in current group

m=size of nums

TOTAL TIME COMPLEXITY OF CODE

O(Ng(n-m)*m^2)

Where N = no. of test cases

SPACE COMPLEXITY ANALYSIS

1. **VECTOR<VECTOR<STRING>> RESULT :**

BEST CASE:

The best case occurs when all the groups are found during the matching, and hence the size of the RESULT vector will be 0. Therefore, the space complexity will be O(1).

WORST CASE:

The worst case occurs when none of the groups are found during the matching, and hence the size of the RESULT vector will be equal to the size of the GROUPS vector. Therefore, the space complexity will be O(groups.size() * group.size()).

2. **VECTOR<VECTOR<STRING>> GROUPS :**

BEST CASE:

The best case occurs when there are no groups in the input file, and hence the size of the GROUPS vector will be 0. Therefore, the space complexity will be O(1).

WORST CASE:

The worst case occurs when the input file contains the maximum possible number of groups, and the size of each group is the maximum possible size. Therefore, the space complexity will be O(groups.size() * group.size()).

3. **VECTOR<VECTOR<STRING>> NUMS :**

BEST CASE:

The best case occurs when there are no test cases in the input file, and hence the size of the NUMS vector will be 0. Therefore, the space complexity will be O(1).

WORST CASE:

The worst case occurs when the input file contains the maximum possible number of test cases, and the size of each test case is the maximum possible size. Therefore, the space complexity will be O(nums.size() * num_list.size()).

4. **VECTOR<BOOL> VISITED :**

BEST CASE:

The best case occurs when the size of the test case being matched is 0, and hence the size of the VISITED vector will be 0. Therefore, the space complexity will be O(1).

WORST CASE:

The worst case occurs when the size of the test case being matched is the maximum possible size, and hence the size of the VISITED vector will be equal to the size of the test case. Therefore, the space complexity will be O(num_list.size()).

QUESTION 2: GRAPHS

INTRODUCTION:

Main aim of this code is to find the minimum weight Hamiltonian cycle (a path in a graph that passes through every vertex exactly once and returns to the starting vertex, such that the sum of the weights of the edges is minimized) in a graph given as an input file, with some additional restrictions on the weights of certain edges.

ALGORITHM:

FUNCTION: MY_NEXT_PERMUTATION

```
1.  n = size of seq                                   O(1)
2.  k = -1                                            O(1)
3.  for i from n-2 to 0 (decrement by 1)
            a. if seq[i] < seq[i+1], set k = i and break  O(n)
4.  if k == -1, return false                          O(1)
5.  l = -1                                            O(1)
6.  for i from n-1 to k+1 (decrement by 1)
            a. if seq[i] > seq[k], set l = i and break    O(n)
7.  swap seq[k] and seq[l]                            O(1)
8.  start = k+1, end = n-1                            O(1)
9.  while start < end
            a. swap seq[start] and seq[end]           O(1)
            b. increment start and decrement end      O(1)
10. return true                                       O(1)
```

TIME COMPLEXITY :

The time complexity of the for loops is O(n) since they iterate through the entire sequence. The while loop in step 9 also has a time complexity of O(n) since it swaps elements from start to end. Therefore, the overall time complexity of this function is **O(n).**

FUNCTION: PROBLEM

```
1.  vertex = {}                                          O(1)
2.  for i = 0 to V-1 do                                  O(V)
        a.  if i != s then
3.  add i to vertex
4.  min_path = MAX_INT                                   O(1)
5.  do {                                                 O(V!)
        a.  current_pathweight = 0                       O(1)
        b.  current_path_order = {}                      O(1)
        c.  add s to current_path_order                  O(1)
        d.  k = s                                        O(1)
        e.  valid = true                                 O(1)
        f.  c = 1                                        O(1)
        g.  for i = 0 to vertex.size()-1 do              O(V)
        h.  if graph[k][vertex[i]] == 0 then             O(1)
6.  valid = false                                        O(1)
7.  break
8.  ii. current_pathweight += graph[k][vertex[i]] + arr[c] O(1)
9.  iii. add vertex[i] to current_path_order              O(1)
10. iv. k = vertex[i]                                     O(1)
11. v. increment c by 1                                   O(1)
12. h. if valid then                                      O(1)
13. if graph[k][s] == 0 then                              O(1)
14. valid = false                                         O(1)
15. ii. else
16. current_pathweight += graph[k][s]                     O(1)
17. add s to current_path_order                           O(1)
18. if valid and current_pathweight < min_path then       O(1)
19. min_path = current_pathweight                         O(1)
20. ii. set min_path_order as current_path_order          O(1)
21. } while (my_next_permutation(vertex))                 O((V-1)!)
22. if min_path <= k then                                 O(1)
23. return min_path                                       O(1)
24. else
25. return -1                                             O(1)
```

TIME COMPLEXITY :

The overall time complexity of the algorithm is **O(V*(V-1)!)** because it in do-while the function creates maximum possible permutations, which should be equal to **(V)!**.

FUNCTION: MAIN

```
1.  for i = 1 to 3 do                                  O(1)
2.  filename = "P2_test" + i                           O(1)
3.  infile = open(filename + ".txt")                   O(1)


4.  if infile does not exist then                      O(1)
5.  return 1


6.  vertices = read a line from infile                 O(1)
7.  V = count vertices by counting the number of commas and incrementing by one
    O(vertices)


8.  graph = create a V x V matrix of zeros             O(V^2)


9.  edges = read a line from infile                    O(1)
10. weight = read a line from infile                   O(1)


11. sizee = length of weight minus 10                  O(1)
12. for each character c in weight do                  O(weight)
13. if c is a comma then
        a. sizee = sizee - 1


14. w_arr = create an integer array of sizee           O(1)
15. k = 0                                              O(1)
16. temp_str = ""                                      O(1)
```

```
17. flag = false                                        O(1)
18. for each character c in weight do                   O(weight)
19. if c is '}' then
        a. flag = false
20. if c is '{' then
        a. flag = true
21. else if flag is true then
        a. if c is a digit then
        b. temp_str = temp_str + c
        c. else if c is a comma then
        d. w_arr[k] = convert temp_str to an integer    O(log(temp_str))
        e. k = k + 1
        f. temp_str = ""


22. if temp_str is not an empty string then             O(1)
23. w_arr[k] = convert temp_str to an integer           O(log(temp_str))
24. k = k + 1


25. k = 0                                               O(1)
26. flag = false                                        O(1)
27. for each character c in edges do                    O(edges)
28. if c is '}' then
        a. flag = false
29. if c is '{' then
        a. flag = true
30. else if flag is true then
        a. if c is 'h' or (c is a capital letter between 'A' and 'Z') then
        b. if c is 'h' then
                i. l = integer value of the character two indices after c minus 64   O(1)
               ii. graph[0][l] = w_arr[k]               O(1)
              iii. graph[l][0] = w_arr[k]               O(1)
               iv. k = k + 1                            O(1)
                v. i = i + 2                            O(1)
        c. else
                i. m = integer value of c minus 64      O(1)
               ii. l = integer value of the character two indices after c minus 64   O(1)
```

```
            iii. graph[m][l] = w_arr[k]                   O(1)
            iv.  graph[l][m] = w_arr[k]                   O(1)
            v.   k = k + 1                                O(1)
            vi.  i = i + 2                                O(1)
31. s = 0                                   O(1)
32. min_path_order = new vector of integers               O(1)
33. arr = new array of integers with size V               O(V)
34. read a line from input file and store it in delivery  O(1)
35. arr[0] = 0                                            O(1)
36. k = 1                                                 O(1)
37. for i from 0 to length of delivery - 1 do             O(delivery)
        a. if delivery[i] == "=" then
        b. token = ""
        c. for j from i+1 to length of delivery do        O(delivery)
            i.   if delivery[j] == "," or delivery[j] == null then
            ii.  arr[k] = integer value of token          O(1)
            iii. k = k + 1                                O(1)
            iv.  token = ""
            v.   break out of loop
        d. else
            i.   concatenate delivery[j] to token         O(1)
        e. end if
        f. end for
38. end if
39. end for


40. read a line from input file and store it in limit     O(1)
41. flag = False                                          O(1)
42. delivery = ""                                         O(1)
43. k = 0                                                 O(1)
44. for i from 0 to length of limit - 1 do                O(limit)
        a. if limit[i] == "=" then
            i.   flag = True                              O(1)
        b. else if flag == True then
            i.   concatenate limit[i] to delivery         O(1)
        c. end if
45. end for
46. num = integer value of delivery                       O(1)
```

```
47. min_path = result of calling function Problem with arguments graph, s, min_path_order,
    arr, and num                                                            O(1)


48. if min_path > -1 then
        a.  for i from 0 to length of min_path_order - 1 do
                i.  if min_path_order[i] == 0 then
                        1.  print starting vertex                           O(1)
        b.  else
                        1.  print other vertices                            O(1)
        c.  end if
        d.  end for
49. else
50.         print Not feasible                                              O(1)
51. end if
52. close input file                                                        O(1)
53. V = 0                                                                   O(1)
54. return 0                                                                O(1)
```

TIME COMPLEXITY :

The overall time complexity is V^2 because it uses while reading the graph from the file

Asymptotic Time complexities:

Worst case: The worst-case time complexity of the given code is **O(V)!** where n is the number of vertices in the graph.

QUESTION 3: DYNAMIC PROGRAMMING

PART A

INTRODUCTION:

The aim of this project is to find the total number of ways of email-delivery to Aamir given a maximum number of emails that he can receive. The algorithm used for this purpose is the Count Email Ways algorithm.

ALGORITHM:

```
CountEmailWays(n):
  // Initialize f[0] and f[1]
  f[0] = 1
  f[1] = 1

  // Compute f[i] for i from 2 to n
  for i = 2 to n:
    f[i] = f[i-1] + f[i-2]

  // Return the final value of f[n]
  return f[n]
```

TABLE:

"n" e-mails	Number of ways
3	3
8	34
75	3416454622906707
1225	740221924612838153551556980536349050965665573918977317062028979849855520216 8614220564123097306186533878335573209493172256624481282050263922108038105180 6039083409995678958872442670188369172105693688882153901354981O 7060634049645656705056309965305780860046993

TIME COMPLEXITY ANALYSIS:

The time complexity of the Count Email Ways algorithm can be analyzed as follows:

Step 1 takes constant time.

Step 2 is executed n-1 times, and each iteration involves three primitive operations: two additions and one assignment. Therefore, the total number of primitive operations in Step 2 is 3(n-1).

Step 3 takes constant time.

Therefore, the total number of primitive operations in the algorithm is 3(n-1) + O(1), which is O(n). Hence, the asymptotic time complexity of the algorithm is O(n).

CONCLUSION:

The Count Email Ways algorithm is an efficient algorithm for computing the total number of ways of email-delivery to Aamir given a maximum number of emails that he can receive. The algorithm has a linear time complexity, which makes it suitable for large values of n.

PART B

ALGORITHM FOR OPTIMAL COST:

```
1. opt[1] = 0
2. for i in range(2, n+1):
3.     opt[i] = infinity
4.     for j in range(1, i):
5.         opt[i] = min(opt[i], opt[j] + c[j, i])
6. return opt[n]
```

TIME COMPLEXITY:

The time complexity of this algorithm is O(n^2) and the space complexity is O(n).

ANALYSIS:

SPACE COMPLEXITY:

The algorithm uses an array opt of size n+1 to store the minimum cost of partitioning a subsequence of length up to i, for i ranging from 1 to n. Therefore, the space complexity of the algorithm is O(n).

TIME COMPLEXITY:

The outer loop executes n-1 times, and the inner loop executes i-1 times, where i ranges from 2 to n. Therefore, the total number of iterations of the inner loop is 1 + 2 + ... + (n-1), which is equal to n*(n-1)/2.

The cost of each iteration of the inner loop is O(1) since it involves a constant amount of computation to update the value of opt[i].

Therefore, the time complexity of the algorithm is O(n^2).

ALGORITHM FOR OPTIMAL PATH:

Let path[i] be the point visited before i on the optimal path from 1 to i. We can find the optimal path by tracing back from n to 1 using path.

```
1. i = n
2. path[i] = None
3. while i > 1:
4.     for j in range(1, i):
5.         if opt[i] == opt[j] + c[j, i]:
6.             path[j] = j
7.             path[i] = path[j]
8.             i = j
9.             break
10. return path
```

TIME COMPLEXITY:

The time complexity of this algorithm is O(n) and the space complexity is O(n).

ANALYSIS:

The given algorithm uses dynamic programming to solve a problem where we want to find the minimum cost of partitioning an input sequence of length n into non-empty subsequences, and also find the path that corresponds to the minimum cost partition. Here, c[j, i] represents the cost of partitioning the subsequence from index j to index i.

Let's analyze the space and time complexity of the given algorithm:

SPACE COMPLEXITY:

The algorithm uses two arrays opt and path, both of size n+1. The array opt stores the minimum cost of partitioning a subsequence of length up to i, for i ranging from 1 to n. The array path stores the index of the optimal partition point for each subsequence of length up to i, for i ranging from 1 to n. Therefore, the space complexity of the algorithm is O(n).

TIME COMPLEXITY:

The outer loop executes at most n-1 times, since i is initialized to n and it decreases by at least 1 in each iteration until it reaches 1.

The inner loop executes i-1 times, where i ranges from 2 to n. Therefore, the total number of iterations of the inner loop is 1 + 2 + ... + (n-1), which is equal to n*(n-1)/2.

The cost of each iteration of the inner loop is O(1) since it involves a constant amount of computation to update the values of path[j] and path[i].

Therefore, the time complexity of the algorithm is O(n^2).

EXAMPLE:

Suppose we have n = 5 and the cost matrix is as follows

```
   1 2 3 4 5
  ---------
1| 0 1 5 6 8
2|   0 3 2 4
3|     0 6 3
4|       0 1
5|         0
```

Using the above algorithms, we can find the optimal cost to be 8 and the optimal path to be

1->2->4->5.

RESULTS OF SAMPLE TEST CASES:

CASE 1:

Optimal cost = 150

Optimal path: 1->3->7->8

CASE 2:

Optimal cost = 320

Optimal path: 1->2->3->7->10