Design and Analysis of Algorithms (Project)

By (21i-0749, 21i-2990, 21i-2992) of section B

Question 1:

Solved by Harras Mansoor (21I-0749)

```
d = 10
p = 0
q = 13
file = open("p1_input.txt")
group = ""
group_text = ""
gcount = 0
i = 0
gmax = 0
group = read(file)

while (group[0] != 'T') do        //O(m)
    group = read(file)
    if (group[0] == 'T') do
       break
    group_text += group
    gcount += 1
    j = 0
    while group[i] != '\0' do    //O(m)
        if group[i] == '\''
            j += 1
        i += 1
     if j > gmax
        gmax = j
            i = 0

 gmax = gmax / 2
 string pattern[gcount][gmax];


bool gcheck[gcount]
i = 0
a = ""
x = 0
y = 0
first = true
flag = false

while (group_text[i] != '\0') do    //O(m)
    while (group_text[i] >= 'a' && group_text[i] <= 'z')
        || (group_text[i] >= 'A' && group_text[i] <= 'Z') do  //O(m)
        a = a + group_text[i]
        i++
        flag = true
    if flag == true
        if first != true
            pattern[x][y] = a
            a = ""
            y++
            if group_text[i + 1] == '}'
```

```
                x++
                y = 0
            else
                first = false
                a = ""
        flag = false
        i++

print "Pattern"
int g[gcount]

for k --> 0 to gcount           //O(m)
    for l --> 0 to gmax         //O(m^2)
        if pattern[k][l] != ""
            print pattern[k][l]
            g[k] = l


int grouphash[gcount]

for i --> 0 to (gcount) do      //O(m)
    grouphash[i] = 0
for j --> 0 to (g[i] + 1) do    //O(m)
    p = 0
    for k --> 0 to (pattern[i][j].length()) :  //O(m^2)
        p = (d * p + pattern[i][j][k]) % q
    grouphash[i] += p

string text[gcount]



for i --> 0 to (gcount) do      //O(m)
    g[i] = g[i] + 1
    print "G[", i, "]:", g[i]

while (not end of file) do      //O(m)
    i = 0
    if (group[0] == 'T') then
    group = read line from file
    end if

    while (group[i] != '[') do     //O(m)
    i = i + 1
    end while

    i = i + 1
    t = ""
    c = 0

    while (group[i] != end of string) do   //O(m)
        while ((group[i] >= 'a' and group[i] <= 'z')
            or (group[i] >= 'A' and group[i] <= 'Z')) do //O(m)
            t = t + group[i]
            i = i + 1
            end while

        while (group[i] != end of string and not
            ((group[i] >= 'a' and group[i] <= 'z') or
                (group[i] >= 'A' and group[i] <= 'Z'))) do //O(m)
            i = i + 1
            end while
```

```
    c = c + 1
    t = t + ","
    end while

c = c - 1
string text[c]
i = 1
j = 0
a = ""

while (t[i] != end of string) do  //O(m)
a = ""
    while (t[i] != ',') do            //O(m^2)
    a = a + t[i]
    i = i + 1
    end while

text[j] = a
j = j + 1
i = i + 1
end while

for i --> 0 to gcount do          //O(m)
    gcheck[i] = false
    end for

f = false
bool tcheck[c]

for i --> 0 to c do          //O(n)
    tcheck[i] = true
    end for

int texthash[c]

for i --> 0 to c do          //O(n)
    texthash[i] = 0
    end for

        p = 0

for i --> 0 to c do          //O(n)
    p = 0
    for k --> 0 to (text[i].length()) do  //O(n^2)
        p = (d * p + text[i][k]) % q
        end for

    texthash[i] = p
    end for

bool mcheck[gcount][g]



for i --> 0 to gcount do            //O(m)
    for j --> 0 to g[i] do
        mcheck[i][j] = true
        end for


hash = 0
x = 0
```

```
for i --> 0 to gcount do              //O(m)
    bool check[g[i]]
    hash = 0
    x = 0

    for j --> 0 to g[i] do            //O(m)
        hash += texthash[j]
        end for

    y = j

    for k --> 0 to c do                   //O(m*n)
        if (hash == grouphash[i]) then
            for l --> 0 to g[i] do        //O(m*n*m)
                for m --> x to (x+g[i]) do  //O(m*n*m*m)
                    if ((text[m] == pattern[i][l]) && tcheck[m] == true)
                        //O(m*n*m*m*m)
                        check[l] = true
                        break loop
                    else
                        check[l]=false

            for v --> 0 to g[i] do         //O(m)
                if (check[v] == true)
                    f = true
                else
                    f = false
                    break;

            if (f == true) {
                print "Group[", i + 1,"] : "," is checked"
                for l --> k to (k+g[i])
                    tcheck[l] = false

                gcheck[i] = true

            hash = hash - texthash[k] + texthash[y]
            y++
            x = x + 1

            if gcheck[i] == true
                break loop

for j --> 0 to gcount do     //O(m)
    if (gcheck[j] == true)
        f = true;
    else
        f = false;
        break;
    end for

if f == true
    print "TRUE"
else
    print "FALSE"
getline from file(group)
```

Report:

I used hashing to solve the question.(modified Rabin Karp) which gave me less time complexity as compared to naive or bruteforce approach. First I solved this problem by naive algorithm which gave me timecomplexity of around O(n^5), then I solved the question using modified rabin karp and got the following result.

Time complexity for file reading:

m = size of total groups and their values

n = Size of test case array.

n > m

O(m+m+m++m+n+n+n+n) → O(4m+4n) → O(n)

Time complexity:

m = size of total groups and their values

n = Size of test case array.

As I used hashing so the average case time complexity will be: **O(m^3 + m*n)** → **O(n^3)**

Worst case time complexity will be: **O(m*n*m*m*m)** → **O(n^5)**

Space Complexity:

m = size of total groups and their values

n = Size of test case array.

string array for all groups and ingredients = m

String array for all the ingredients of test cases = n

Bool array for all groups and ingredients = d

Bool array for all the ingredients of test cases = e

Space Complexity: **O(m+n+d+e)** → **O(n)**

Question 2:

Solved by: Abtaal Aatif (21i-2990) (B)

Introduction:

The problem presented to us is a modified traveling salesman problem. Thus, the help of the held-karp algorithm is employed to help us complete it for the requirements of this project.

Note: in pseudocode, complexity is only mentioned where code gives non-constant time complexity. Complexities are written in **bold** next to the relevant line. Function head is written in **bold.**

Pseudocode:

```
adj, memo as int**
N, time_limit, feasible, pindex as int
vname as char*
INF = 1*e^9
vcost, pathway as int*

//file reading function
test_read(fname)
        Ifstream obj
        obj.open(fname)
        obj modified to not skip white-space characters
        temp = 0
        line_point = 0
        while( temp != '{' )                //O(n)
                Obj >> temp
                line_point++
        while( temp != '}' )                //O(n)
                Obj >> temp
                If (temp is an upper-case letter or 'h' )
                        N++
        vname = 0
        vname = new char[N] //to store names of vertices
        move file ptr to beginning of line using line_point as index    //O(n)
        Obj >> temp
        vindex = 0
        while( temp != '}' )                //O(n)
                Obj >> temp
                If (temp is an upper-case letter or 'h' )
                        vname[vindex] = temp
                        vindex++
        Edgelist, weights, waitlist //strings to store relevant data from files
        getline(obj, edgelist) //done twice to move file ptr first and empty buffer
        getline(obj, weights)
        getline(waitlist)
        Time_limit = 0;
        while( reading file )               //O(n)
```

```
            If( temp is a number character )
                time_limit = time_limt + temp - '0'
        waitlen = waitlist.length()     //O(1)
        for( x2 = 1 to waitlen )        //O(n)
            Store waiting times in vcost[]
        while( edgelist[index] != '{' )     //O(n)
            Index++
        while( edgelist[index] != '}' )     //O(n)
            while( edgelist[index] != ')' )  //O(n) as it moves index variable for outer loop
                Store edge vertices and update index
            for( l = 1 to N )           //O(n^2)
                find index of edgelist vertex pairs
            while(weights[index2] != ',')   //O(n^2)
                Store weights in adj
        obj.close()
        memo = new int*[N]
        for( i = 0 to n-1 )             //O(n*(2^n))
            Initialise memo row with -1 in each cell
        pathway = new int[N + 1]
        for( i = 0 to n-1 )             //O(n)
            pathway[i] = -1
        pathway[0] = 0
        pindex = 1

//function to print feasible paths
pathend()
        if( edge between pathway[N-1] and vertex 0 exists )
            s1 = "Feasible Paths Found:\n\n"
            total = 0
            pathway[N] = 0
            wtime = 0
            for( i = 0 to n-1 )         //O(n)
                a = pathway[i]
                total = total + adj [ pathway[i] ] [ pathway[i + 1] ]
                wtime = wtime + vcost[i]
            total = total + wtime
            if( total <= time_limit )
                if( feasible == 0 )
                    print(s1)
                for( i = 0 to n-1 )     //O(n)
                    print( vname[ pathway[i] ] )
                print('\t')
                print(total)
                feasible++

//held-karp function to solve actual problem
held_karp( position, mask )
        if( all bits of mask set to 1 ) //all cities have been visited
            pathend()
            Return adj[position][0]
        if( memo[position][mask] != -1 ) //cell has previously been set
            Return memo[position][mask]
        //if not a base case:
        ans = INF
```

```
        for( i = 0 to n - 1)                    //O(n))
                if( city not visited in mask )
                        pathway[pindex] = i
                        pindex++
                        new_mask = mask | 1 << i      //add city to mask as visited
                        new_ans = adj[position][i] + held_karp(i,
new_mask)    //O((n^2)*(2^n))
                        if( ans > new_ans )
                                ans = new_ans
                        pathway[pindex] = -1
                        pindex–
        Memo[position][mask] = ans //update memo table
        Return ans

//master function
problem _solver(fname)
        test_read(fname)                //O(n*(2^n))
        feasible = 0
        ans = held_karp(0, 1)           //O((n^2)*(2^n))
        if( !feasible )
                print(“No feasible circuit”)
                print(‘\n’)
                print(‘\n’)

//main function
main
        tcases = 0
        print(“Enter number of test cases: “)
        input(tcases)
        fname = file path of first test case file
        system(“PAUSE”)
        system(“CLS”)
        cnum = ‘1’
        for( x = 1 to tcases )          //O(n)
                print(“Test case # “)
                print(x)
                print(“: “)
                fname[index for test case number] = cnum
                problem_solver(fname)                   //O((n^3)*(2^n))
                for(d = 0 to n - 1 )
                        delete[ ] adj[d]
                        delete[ ] memo[d]
                delete [ ] adj
                adj = 0
                memo = 0
                delete [ ] vcost
                vcost = 0
                time_limit = 0
                N = 0
                delete [ ] vname
                vname = 0
                cnum++
                print(‘////////////////////////////////////////////’)
```

Time complexity:

(i) Without including file-reading loop:
O((n^2) * (2^n))

(ii) With file-reading loop:
O((n^3) * (2^n))
An additional n is multiplied to accommodate for the outermost loop to iterate over all test cases.

(iii) Complexities of functions:

a. test_read **O(n*(2^n))** taken to fill memo table with -1
b. pathend **O(n)** taken to print path of n vertices and check feasibility
c. held_karp **O((n^2)*(2^n))** to run for all hamiltonian circuits starting and ending at 'h'

Conclusion:

While a better time complexity can be achieved using algorithms that provide an approximate answer, for the purpose of this assignment, an exact answer was sought out since this project has us deal with relatively small graphs. Thus, the held-karp algorithm was used to resolve the traveling salesman problem represented here by a robot moving between locations in a warehouse. To calculate all possible hamiltonian cycles which visit the starting nodes, we need a memoization table of size N x 2^N. Running this algorithm recursively with a for loop for N iterations to fill this table and use it to find feasible paths is what leads to a solution having (n^2)*(2^n) time complexity on its on and slightly more if we include the loop for running all test cases.

Question 3:

Solved by: Daniyal Kaleem (21I-2992) (B)

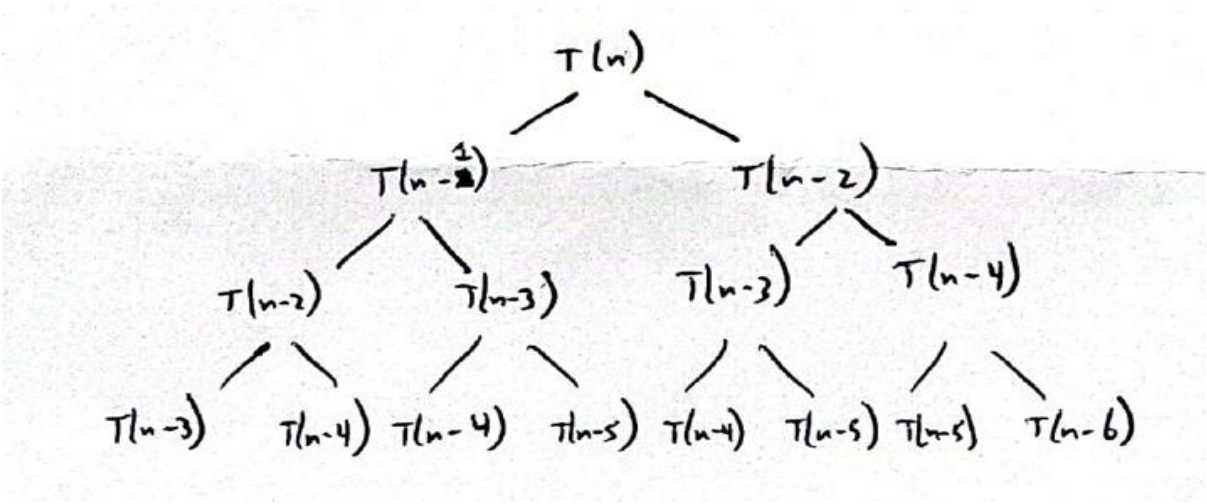
Part A:

Deriving Optimal Solution:

The problem presented to us is a fibonacci problem. To reach 'n' emails Aamir can either receive 1 or 2 emails which means he could either have n-1 or n-2 emails before it. Therefore we can conclude that the number of ways that Aamir can receive 'n' emails equals the sum of the number of ways that he can receive n-1 or n-2 emails.

Recursive Definition:

T(n) = T(n-1) + T(n-2) for all positive n
Where T(n) = 1 for 0 n 2



The recursion tree drawn above represents the recursive calls taking place.
As can be seen, values to be calculated are repeated throughout the recursion tree hence we can store the result of past calculations dynamically to be used instead of repeating calculations.
Since at any point in time we only need previous 2 values to compute current values, we do not need to store all values and just store previous 2 values at any given time to minimize space complexity and still maintain the program as dynamic.

Pseudocode:

Compute_Email_Ways(n)	Time Complexity
num1, num2 ← 1	O(1)
While n is greater than 1	O(n)
num1 = num1 + num2	O(n)
num2 = num1 - num2	O(n)
n--	O(n)
end while	
return num1	O(1)

Time Complexity: O(n)
Space Complexity: O(1)

Below is table asked to be computed in project:

“n” emails	Number of Ways
3	3
8	34
75	3.41645e+15
1225	7.40222e+255

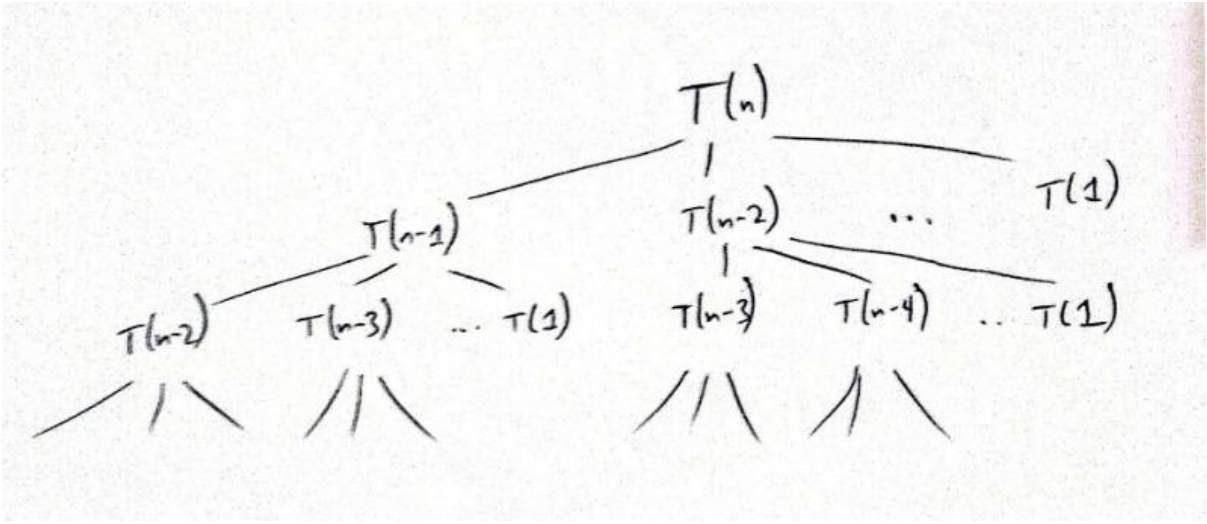
Part B:

Deriving Optimal Solution:
This problem formulates a least cost problem otherwise known as shortest cost problem.

We have n stops with different costs for going from one stop to another. The car starts at stop 1 and needs to reach stop n.
The car cannot go to a previous stop and it can also skip stops.
Objective: Find minimum cost to go from stop 1 to n

To find out the minimum cost to reach any stop n, we would need to calculate the minimum cost to go to all previous stops, add the distance to go to n from that stop directly to n. From these resultant values we will choose the minimum value as the minimal cost to reach stop n.

Recursive Definition:
Let c[i][j] represent the cost to go directly from stop i to stop j.
T(n) = Min(T(n-1)+c[n-1][n], T(n-2)+c[n-2][n], ... , T(1) + c[1][n])
Where T(1) = 0



The recursion tree drawn above represents all the recursive calls taking place.
As can be seen, values to be calculated are repeated throughout the recursion tree hence we can store the result of past calculations dynamically to be used instead of repeating calculations.

Pseudocode: Time complexity has been noted in areas of significance.

Least_Cost_Path(n, c)	Time Complexity
memo[n] ← ∞	
parent[n] ← 0	
memo[0] ← 0	
for i ← 1 to n	O(n)
for j ← 0 to i	O(n2)
If memo[j] + c[j][i] greater than memo[i]	O(n2)
memo[i] = memo[j] + c[j][i]	
parent[i] = j	
end if	
end for	
end for	
Print memo[n-1] as minimal cost	

//Path calculation algorithm

```
k  0
path[n]
p ← n-1

While p != 0                                                    O(n)
        path[k] = p + 1
        p = parent[p]
        k++
end while

Print 1
While k > 0                                                     O(n)
        k--
        Print path[k]
end while

return
```

	Time Complexity	Space Complexity
Optimal Cost	O(n2)	O(n) w/o cost matrix else O(n2)
Optimal Path	O(n)	O(n)