Joey Zhang (Student ID: 1006750437) and Chris Tong (Student ID: 1005661874)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	...	...	Total
Mark										

Table 1: Marking Table

1 Exercise 1

A new railway route is being set up from city A to city B. There are totally n proposed sites along that route, on some of which a railway station is to be built. For each site $i(1 \leq i \leq n)$, you know its distance di from city A. Assume that $d1 \leq d2 \leq ...dn\text{-}1 \leq dn$. The government is willing to provide funds of p1, p2,..., pn for constructing a station at these n cities respectively. The p values are not necessarily in increasing order. Owing to some regulations by the government, no two railway stations may be located at a distance of less than or equal to 15 miles from one another. Your job is to come up with an efficient dynamic programming algorithm to compute the number of railway stations and their locations that maximize the total amount of money that the government will be willing to pay, subject to the aforementioned constraint. Write the relevant recurrence relation, a non-recursive pseudocode and analyze the time complexity of the algorithm.

Problem Solution:

1.1 Relevant Recurrence Relation

We begin the problem by considering an abstract scenario where we want to calculate the maximum amount of money to build a station 'i' at distance d_i. Here, we assume that up to station i-1 there has already been a station built at d_{i-1} There are two possible cases that we need to consider:

- In the case where $d_i - d_{i-1} > 15$
  - maximum amount of money is achieved at station i by the sum of P_i and the maximum amount of money obtainable by building stations up to station i-1.
- In the case where $d_i - d_{i-1} <= 15$
  - choice1: Choose to not build station $\rightarrow$ maximum amount of money is amount it took to build up to station i-1

- choice2: Choose to build station by forfeiting the construction of $i-1$ staion, $i-2$ station up until the distance is greater than 15 miles between station i and station i-k. $\rightarrow$ Maximum amount of money = P_i+ money for building up to station i-k.
- choose Max of the two choices to store.

Therefore, the relevant recurrence relation could be summarized as the following:

```
//Maximum amount of money up to station i
//is represented by max_money[i]
if d[i] - d[i-1] > 15{
    max_money[i] = max_money[i-1] +p[i]
}
else {
    max_money[i]= max(max_money[i-1], max_money[i-k] + p[i])
    //here, k is the number we determine
    //that allows station i to be seperated
    //by 15 miles with a previous station.
    //We choose the max of two choices.
}
```

1.2 Pseudocode

In the following Pseudocode the variables d,p, n, and max_money_array are of following nature:

- d: array for distance accessible using d[i]
- p: array for cost accessible using p[i]
- n: total number of stations.
- max_money_array: array storing maximum money obtainable at each i.

```
function maximum_money(d,p,n,max_money_array){
    //for base case
    max_money_array[1]=p[1]
    previous_station=1 //previous station initialized as 1

    //iterate from station 2 to last station n
    for i from 2 to n:
```

```
            if d[i] - d[previous_station] >15:
                max_money_array[i] = max_money_array[i-1] + p[i]
                previous_station=1
            else
                //find integer k
                k=find_k(d,i)
                if k<0 //no successful k is found
                    max_money_array[i]= max_money_array[i-1]
                else
                    max_money_array[i]=max(max_money_array[i-1]
                                           ,max_money_array[i]
                                           + max_money_array[i-k])
    }

    //supporting function
    function find_k(d, i){
        k=1
        distance = d[i]-d[i-k]
        while (distance <= 15):
            if(k>=i)
                return k=-1
            distance= d[i]-d[i-k]
            k++
        return k
    }
```

1.3 Time Complexity Analysis

To analyze time complexity, lets break our function down

- function find_k(d,i) takes worst case O(i) time iterating until k=i
- main function maximum_money(d,p,n,max_money_array) takes O(n) times iterating through all possible stations

Therefore, in the best case, the whole function runs in O(n) time where the else statement in the main function is never triggered.

However in the worst case where else is always triggered and find_k(d,i) functions runs for O(i) each time. We have the following summation series 1+2+3+4+5+....+n.

Until in the end we have $O(n^2)$ complexity

2 Exercise 2

Given a string s, we want to find the longest sub-sequence s' of s that is a palindrome (reads the same in both directions). The letters don't have to be consecutive. For example, "adcda" is the longest palindromic sub-sequence of "taidfcdag", others such as "aa" or "dcd" are palindromic sub-sequences but not the longest. Develop a dynamic programming algorithm that returns the longest palindrome of a given string s. Give pseudocode, explain your algorithm and analyze its time and space complexity.

Problem Solution:

2.1 Algorithm Description

The algorithm for solving this problem makes use of Dynamic programming in that it uses a 2D matrix to store palindrome results for a smaller given substring and build larger palindrome results on these smaller palindrome results already obtained.

The indices i and j used to access the matrix indicate the starting and ending index of a substring. As an example. i=2 and j=4 means substring of length 3 constructed with the given string characters s[2],s[3],s[4]. Similarly, when i=j it means that we are accessing a substring of length 1 that is made of of one character.

The algorithm starts by initializing the diagonal elements of the matrix to values of 1. This is to say that each letter of a given string input is already itself a palindrome that is of length 1.

Next, we iterate through the entire string over all possible starting and ending positions for a substring. For each of these iterations, we check if the current substring has its first character and last character as equivalent. If they are the same, we simply add 2 to its length (2D_array[index to first chracter][index to second chracter]). If they are not the same, we choose the maximum result of a the substring the excludes either excludes the first of the chracter or the last and store the result in the 2D array at index i and j.

To retrieve the longest palindrome itself, we need to make usage of a backtracking function. This function builds the longest Subsequence by iterating through the entire length of the string itself and pivot-ably append characters into an initially empty

string. It first checks if the current character and its corresponding end character match. If they match, the chracter is appended to the string. If they don't match, either the left pointer or right pointer is moved and adjusted according to which two substrings contain the maximum palindrome substring within the substring.

Finally, to return the string in the palindrome format required, we have to append a reversed version of itself to it. Then return.

2.2 Pseudocode

```
function longest_palindrome_substring(s){
    //Initialization
    2d_array = [size][size] //n by n 2d array
    size=len(s)

    //Populate diagonal elements as 1
    for i=0:1:size-1{
        2d_array[i][i]=1
    }

    //iterate over all possible substring
    for length=2:1:size{ //all possible lengths
        for start_index=0:1:size-length{ //all substring at diff starting positions
            end_index= start_index+ length-1 //ending index of substring
            if (s[start_index] == s[end_index]){
                2d_array= 2d_array[start_index+1][end_index-1] + 2
            }
            else{ //choose maximum of longest sub-substring
                if (2d_array[start_index+1][end_index]>2d_array[start_index][end_index-1]){
                    2d_array[start_index][end_index] = 2d_array[start_index+1][end_index]
                }
                else 2d_array[start_index][end_index] = 2d_array[start_index][end_index-1]
            }
        }
    }

    backtrack_result= backtrack(s,2d_array)
    longest_palindrome_substring= backtrack_result+reverse_string(back_track_result)
    return longest_plaindrome_substring
}

function backtrack(s,2d_array){
    //initialize empty string
    empty_string=[]
    start=0
    end= len(s)-1

    //iterate through string
    while (1){
        if(start>end) {
            return empty_string
        }
        else{
            if (s[end]==s[start]){
```

```
                empty_string.append(s[start])

                //pivot start and end pointer
                end++
                start++
            }
            else{
                if(2d_array[start+1][end]>table[start][end-1]){
                    start++
                }
                else{
                    end--
                }
            }
        }
    }
}
```

2.3 Time and Space Complexity Analysis

- Space: Used an array of size n^2 where n is the length of the input string. This is the case even though we only use half of the 2d_array. Notice the other bottom half of the diagonal matrix remains empty.
- Time: $O(n^2)$ because the double for loop was the most dominant. Although the backtracking function has O(n) time complexity, the overall process is dominated by the two for loops.

3 Exercise 3

Consider a directed graph $G = (V, E)$ with edge weights w_i , where the weights can be either positive or negative. We want to determine if there exists a simple cycle in G (meaning the only repeated vertices are the first and last vertex) such that the sum of edge weights on this cycle is exactly 0. Prove that this problem is NP-complete by reducing it from the Subset Sum Problem. The Subset Sum Problem involves finding whether there exists a subset of a given set of integers whose sum is equal to a given target integer.

Problem Solution:

According to lecture materials, we need to prove NP-complete problems via 5 steps. The following subsections will address each of the 5 steps in detail and attempt to prove that the determination of a simple cycle in G could be classified as NP-complete. We will now refer to the above problem as problem Q and assume the

Subset Sum problem is P.

3.1 Prove that problem Q $\in NP$

A problem is said to belong in the NP set of problems if it is "verifiable" in polynomial time. Let's assume that we already have found a solution that contains a path for a simple cycle.

To confirm that the it is a simple cycle, we just have to sum up all the weights in for the along the edges in the path and verify that it sums up to 0.

Then, assuming that the path is given to us and has n edges with assigned n weights. Then the time it takes us to iterate through these weights and sum them up takes a total of O(n) time.

Therefore, the problem is "verifiable" in polynomial time and thus $\in$ NP

3.2 Select a known NP-complete problem

As mentioned in the problem, we will use an existing NP-complete problem known as the Subset Sum Problem to for reduction.

3.3 The Mapping/Transformation from P to Q

We will prove the transformation from P to Q by constructing the graph G=(V,E) using an arbitrary Subset Sum problem.

Suppose in a set of all real numbers $S = s1, s2, s3, s4....sn$. We want to find a subset S' such that the subset of numbers S'=s1,s2... adds up to precisely 't'.

We can construct a directed graph G with n vertices labeled from 0,1,2,3,4 up to n, and edges (i,j) where i and j are vertices 'from' and indices 'to'. And all edges have weights s_k as drawn from the set S. Additionally, let there be edges (j,0) of weight '-t.' Edge(j,0) means an edge stemming from any vertex 'j' to vertex 0.

3.4 Prove that transformation works

With the above graph, lets suppose that there exists a subset $U \subset G$, where U contains the set of vertices, that sum up the edge weights to 't.' Then, we can claim that starting at the vertex 0 and going to the vertices in U before finally return to vertex 0 through any edge (j,0), we would have to have a zero weight cycle. Since

the edges (j,0) have weight '-t,' and the sum of edge weights from travelling through vertices in U is 't'.

Refer to figure 1. for a visual I drafted up to visualize the problem.

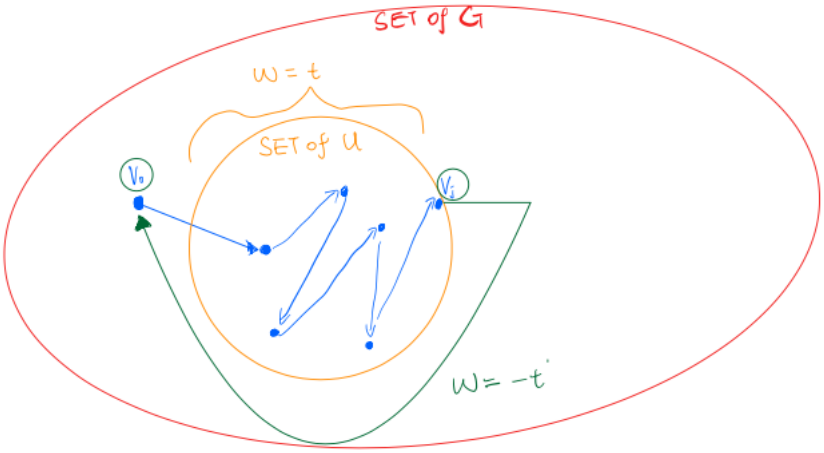

Figure 1. The visualization of set U in set G. Note the vertex V0 and vertex Vj and the positive weight of t and negative weight of t of the edge vj to v0

Therefore, a subset S' of natural integers would only add up to desired integer 't' if there is a simple cycle in graph G.

3.5 Prove that transformation runs in polynomial time

Constructing the above graph of G takes polynomial time as we need to create constant numbers of edges for each integer number in S. In conclusion, the simple cycle problem as inquired could be reduced in polynomial time from the NP-complete subset-sum problem. Therefore, the simple cycle problem in graph G(V,E) is also a NP-complete problem.

4 Exercise 4

The final year is coming and you are trying to complete your degree requirement! There are k areas in each of which you still need to take one course. Luckily for each of the areas, there is a list (set) of courses being offered C1,C2,...,Ck that you have not yet taken. Unfortunately, there are unavoidable time conflicts. Each area has coordinated the schedules so that there are no conflicts internally, but you need to

decide if there is a way to choose a course from each area so that there are no conflicts so that you can complete your degree next year. Prove this problem is NP-complete. Hint: Model it as a graph problem.

Problem Solution: Similarly, we assume that the problem we are trying to solve is Q.

4.1 Prove that problem Q $\in NP$

In order to prove that problem Q is in NP. We have to show that we can verify a given solution in polynomial time. This is possible if we are given a list of k courses to take so that we can graduate next year. Even assuming a linear comparison of each course with every other k-1 course, at most, this verification would take no more than $O(n^2)$ time. This is assuming that k gets as large as n.

4.2 Select a known NP-complete problem

Since the hint has been given for us to model this as a graph problem, we will choose to reduce problem Q from problem P: Vertex Cover problem which is also a NP-complete problem. Recall that the vertex cover problem is the problem of finding the smallest set of vertices in a graph that is incident to at least one other vertex in the graph.

4.3 The Mapping/Transformation from P to Q

Since we know that there are always going to be unavoidable conflicts between two courses from two different areas, it is assured that connections could always be made between these courses from different areas.

In other words, this is saying that across our k areas and out of all the courses of k areas, there are at least k conflicts. We can make use of this fact when building our graph.

The vertices of our graph will represent courses in the k different areas. An edge connecting two vertices means that an existing conflict between two courses belonging to different areas. Note that there will only be edges between courses from different areas due to that within each area, no conflict exists between courses.

With this, we can call a non-deterministic vertex cover algorithm to find $\geq k$ numbers of vertices in G on our graph. These k vertices will then represent the courses that

have no conflict with another course.

After this, a final step of iterating through the found minimum vertices is needed to get rid of all the duplicated vertices within the same area.

Finally, a set containing k courses from k different areas, without inter-conflict, will be preserved.

4.4 Prove that transformation works

Lets assume that we have a vertex cover set of size k for the graph G. This means that we have found k courses that had maximum conflict(a lot of edges extending from said vertex to other vertices from different areas). This means that these k courses will not have conflict between one another because the k courses are from different areas.

4.5 Prove the transformation runs in polynomial time

Since the construction of graph G can be done in polynomial time (assuming worst case with k areas each have k courses $O(k^2)$), and the removal of duplicates can be done in at most linear time O(n), we know that transformation could be run in polynomial time entirely. Thus, one day if we have indeed found a polynomial and deterministic algorithm for vertex cover, we would be able to solve this course selection problem in polynomial time. Until then, this problem is proven to be a NP-complete problem because it could be reduced from the known NP-complete problem of vertex cover.

5 Exercise 5: Programming Exercise

The programming exercise is attached in conjunction with the submission of this assignment report as an .ipynb file.