Algorithm Programming Assignment #2 **Report**

B07901021 潘世軒

(1) Data structure

```
class ChordSet{
    friend class Node;
    public:
        ChordSet();
        ChordSet(int);
        bool connection_assign(int, int);
        int find_connection(int);
        bool chord_existence(int, int);
        void test();
    private:
        vector<int> data;

};
```

使用一個叫做 ChordSet 的資料，將讀入的.in 檔轉成可使用的集合，private member 中 data[i]存放和 i 爲連線的點，舉例來說，將作業提供的 12.in 讀入之後，data 會變成[4, 9, 6, 10, 0, 7, 2, 5, 11, 1, 3, 8]，將資料結構這樣設計的原因是，我們可以透過端點 i 直接去找到他連線的另外一端點(函式 find_connection)，不需要檢查全部的點。另外，也支援判斷某兩點是否存在連線(函式 chord_existance)。

(2) Algorithm

根據輸入建立好 ChordSet 後，先建立二維陣列 M，其中 M[i][j]用來表示 M(i, j)裡的最大 chord 數，在塡完這兩個表格後，由 M[0][N-1]開始跑遞迴，將範圍漸漸切小之後把範圍內的 chord 的起點(擇一)慢慢加入 vector Chordlist 中，最後將 Chordlist 依照輸出格式寫進.out 檔裡面，就完成本次計算了

```
    int **M;
    M = (int **)malloc(sizeof(int*) * N);
    for(int i = 0; i < N; i++){
        M[i] = (int *)malloc(sizeof(int) * N);
    }

    for(int l = 0; l < N; l++){
        for(int i = 0; i < N - l; i++){
            int j = i + l;
            if(i == j){
                M[i][j] = 0;
                continue;
            }
            int k = C.find_connection(j);

            if((k > j) || (k < i)){
                M[i][j] = M[i][j-1];
            }

            else if(k == i){
                M[i][j] = M[i+1][j-1] + 1;
            }

            else if(k < j && k > i){
```

```
                int A = M[i][j-1];
                int B = M[i][k-1] + 1 + M[k+1][j-1];
                if(A >= B){
                    M[i][j] = A;
                }
                else{
                    M[i][j] = B;
                }
            }
        }
    }

```

(Main.cpp 片段->填表格)

```
void traverse(int i, int j, ChordSet* C, vector<int>* list, int** M){
    int k = C->find_connection(j);
    if(!M[i][j]) return;
    else if((k > j) || (k < i)){
        traverse(i, j-1, C, list, M);
    }

    else if(k == i){
        list->push_back(i);
        traverse(i+1, j-1, C, list, M);
    }

    else if(k < j && k > i){
        int A = M[i][j-1];
        int B = M[i][k-1] + 1 + M[k+1][j-1];
        if(A >= B){
            traverse(i, j-1, C, list, M);
        }
        else{
            traverse(i, k-1, C, list, M);
            list->push_back(k);
            traverse(k+1, j-1, C, list, M);
        }
    }
}
```

(函式 traverse)

(3) Findings

這次作業比較困難的部分是控制記憶體用量的部分，因爲 bottom up 的 dynamic programming 很多都需要填完 $O(n^2)$的表格，因此如果不好好控管記憶體的話很容易就會炸開，有些能夠使用比較小的整數型態(如 short)的 vector，我就會盡量使用之，但是最後在做魔王測資 100000.in 的時候我在工作站還是要跑一陣子，甚至有時候還會被 killed 掉，只能祈禱沒有太多人同時在用同一台主機....。

另外 vector 雖然很好用，但是他在自己分配空間的時候是直接無腦兩倍的樣子，因此用比較古老的方法(如自己 malloc)可能會對記憶體用量有比較多的掌控度，另外根據有修計算機結構的同學說法，自己 malloc 可能比較快(？