Algorithm PA2 report

B12901022 廖冠豪

1 Data Structure

In this assignment the solution is implement as a class, MPS.

As can be seen from the prototype of the MPS class on the right, it contains different data structures used to solve this problem. Including 32 bit integer arrays, 16 bit integer arrays, and `std::vector`.

```
#ifndef _MPS_H
#define _MPS_H
#include <vector>
#include <fstream>

class MPS {
public:
  int size;
  int *adj;
  unsigned short **dp_table_1;
  int **dp_table_2;
  std::vector<int> answer;

  void add(int i, int j);
  int run_dp();
  void print(std::fstream &fout);
  void find_prt(int i, int j);
  MPS(int n);
  ~MPS();
private:
  int calc(int i, int j);
};

#endif
```

1.1 Arrays

The MPS class contains multiple arrays.
All arrays are declared as pointers and the space is allocated dynamiclly at runtime when the size of the input is known.

1.1.1 `MPS::adj[]`

The 32 bit integer array `adj[]` is used to store the information about the chords. The size of adj is $2n$, the number of total points. If there exist a chord ij then `adj[i]` $= j$ and `adj[j]` $= i$. For example, given the following input

12
0 4
1 9
2 6
3 10
5 7
8 11
0

The content of adj is

```
adj : [4, 9, 6, 10, 0, 7, 2, 5, 11, 1, 3, 8]
```

1.1.2 `dp_table`

I use 2D arrays to store the value of $M(i, j)$. Notice that instead of creating one 2D array, I use two separate 2D arrays, `dp_table1` and `dp_table2`. The reason for doing so is that, in most cases, since $M(i, j) \leq (j - i + 1)/2$,the value of most entries of the dp table is small enough to fit in unsigned 16 bit integers (0–65535).
More specifically, for $n < 65535$ we can use a 2D array of 16 bit integer to store the dp table. Even for n surpassing this range, we need only use another 2D array of 32 bit integer array to store the values of $M(i, j)$ where $(j - i + 1)/2 \geq 65535$.
Another optimization for memory usage is that since $M(i, j) = 0$ for $i \geq j$ we only need to store the $M(i, j)$ where $i < j$.
The method for storing the dp table is concluded as following.

$$M(i,j) = \begin{cases} 0 & i \geq j \\ \texttt{dp\_table1[i][j - i]} & i < j, \quad j - i < 2 \times 65535 - 1 \\ \texttt{dp\_table2[i][j - i - 2 ×65535 + 1]} & i < j, \quad j - i \geq 2 \times 65535 - 1 \end{cases}$$

Note that although a 16 bit integer can store the value $M(i, j) = 65535$, we use `dp_table2` to save $M(i, j)$ if $j - i = 2 \times 65535 - 1$, this is because we reserve the number 65535 in `dp_table2` to indicate that this entry of the dp table hasn't been calculated in the dp process.

1.2 `std::vector`

1.2.1 `MPS::answer`

When constructing the chords in the optimal solution, a `std::vector` is used. For every chord in the maximum planar subset, its lower endpoint is stored in this vector. Using `std::vector` is not necessary here, the process of recording the chords in the maximum planar subset can also be done with other data structures such as primitive arrays, but the `push_back()` function provided by `std::vector` makes it convenient to add chords to the planar subset.

2 Algorithm

A dynamic programming approach is adopted in this assignment.
Suppose that kj is a chord, the optimal substructure(recurrence relation) for $M(i,j)$ is

$$M(i,j) = \begin{cases} 0 & i \geq j \\ M(i+1, j-1) + 1 & k = i \\ M(i, j-1) & k > j \vee k < i \\ \max\{M(i,j-1), M(i,k-1) + M(k+1, j-1) + 1\} & i < k < j \end{cases}$$

(*reference: HW2*)

2.1 Recursive solution

Starting from $M(0, 2n-1)$, we can conduct the dp process recursively.
The function `calc(int i, int j)` is used to calculate $M(i,j)$.
When `calc(i, j)` is called, we first check if this is a base case ($i \geq j$). If it is a base case, the function returns 0.
Then we check whether $M(i,j)$ is already calculated. We use 65535 in `dp_table1` and -1 in `dp_table2` to indicate that this entry of the dp table is not yet calculated.
If $M(i,j)$ has already been calculated, we simply return the corresponding value in the dp table. If not, we decide based on `adj[j]` which case this is.
(Notice that `adj[j]` is equal to k as defined in the recurrence relation above.)
By this recursive process, we can fill the dp table in a way such that $M(0, 2n-1)$ and all other information required to construct the maximum planar subset are known.

2.2 Constructing the maximum planar subset

The function `find_prt(int i, int j)` is used to find the chords that maximize $M(i, j)$.
Starting from `find_prt(0, 2n - 1)`, the chords in the maximum planar subset can also be found in a recursive manner.
When `find_prt(i, j)` is called, we first check if this is a base case, that is, whether $i \geq j$; if so, we simply return.
Next, based on `adj[j]`, the recurrence relation above, and the value of $M(i, j)$ we can determine how $M(i, j)$ is obtained, and therefore which subcase to process.
For the following process, let $k =$ `adj[j]`.

`find_prt(i, j)`

$$\begin{cases} \texttt{return} & i \geq j \\ \texttt{answer.push\_back(i)}, \texttt{find\_prt(i + 1, j - 1)} & k = j \\ \texttt{find\_prt(i, j - 1)} & M(i, j) = M(i, j-1) \\ \texttt{answer.push\_back(k)}, \texttt{find\_prt(i, k - 1)}, \texttt{find\_prt(k + 1, j - 1)} & \text{else} \end{cases}$$

2.3 Printing the solution

First print out `calc(0, 2n - 1)`, which is the number of chords in the maximum planar subset.
Then after running `find_prt(0, 2n - 1)`, we have the lower endpoint of every chord in the maximum planar subset in the `answer` vector. We then use `std::sort` to sort this vector to fit the output standard.
After `answer` is sorted, we can print out the chords in the maximum planar subset in increasing order of the first endpoint.

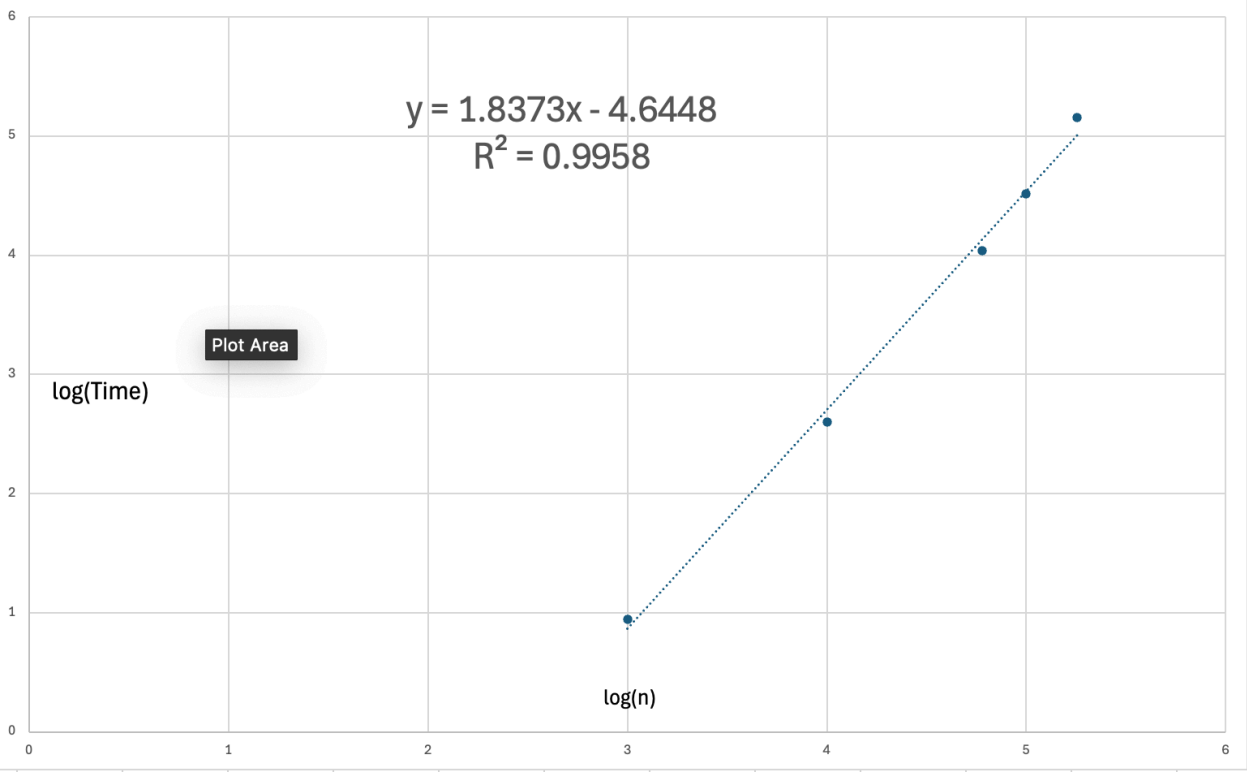
3 Time Complexity Analysis

The time complexity for initializing the data (`adj[]`) is $O(n)$, since there are n chords and each can be processed in constant time.
The time complexity for the dp process (`calc(0, 2n - 1)`) is $O(n^2)$, since the dp table is a n-by-n triangle and we have to fill its entries.
Similarly, the time complexity for constructing the maximum planar subset is also $O(n^2)$, as it also have to process the $O(n^2)$ subcases in the dp table, and each subcase takes $O(1)$ time.
Lastly the time complexity for printing the output is $O(n)$, since there are at most n chords in the maximum planar subset, and printing each chord takes constant time.

The performance of this algorithm for the test cases is as follows

n	CPU Time(ms)	Memory(kB)
12	0.226	6072
1000	8.861	7128
10000	399.9765	103968
60000	10888.6	3527948
100000	32711.7	9844968
180000	143295	34221028

By potting the logarithm of CPU time and the logarithm of n, we can clearly see that the time complexity of this algorithm is indeed approximately $O(n^2)$.



The data point for $n = 12$ is neglected, since it is too small compared with other n values, the lower-order terms has a significant effect in this data point, and would affect the asymptotic analysis.

README

This is README file for Algorithm PA2
Author: B12901022 廖冠豪 (GUAN-HAU, LIAO)
Date: 2024/11/3
=====
SYNOPSIS:

mps <input_file_name> <output_file_name>

This program solves the maximum planar subset problem.
=====
DIRECTORY:

bin/ executable binary
doc/ reports
inputs/ input data
outputs/ output result
lib/ library
src/ source C++ codes
utility/ checker
======
HOW TO COMPILE:

Under the root directory of this PA, simply type
make
======
HOW TO RUN:

./bin/mps <input_file_name> <output_file_name>
For example,
under b12901022_pa2
./bin/mps ./inputs/12.in ./outputs/12.out
======