10271 Chopsticks

Set up

It was written in **C++ 11**. Run by command

```
g++ -lm -lcrypt -O2 -std=c++11 -pipe -DONLINE_JUDGE
```

It has been accepted by Uva.

Observation

- In a set {A,B,C}, A,B must be adajent (assuming all chopsticks are sorted by length)

Algorithm overview

- Build a preprocessing map M[chop][people], in where we store badness in M
    - If we first fix variable people, for each chopstick, there are two subproblem:
        - chopstick is in the optimal solution
            - Then we take chopstick i and i-1
            - M[chop][people] = min{M[chop-1][people],M[chop-2][people] + badness(chop -1, chop)}
        - Chopstick is not in the optimal solution
            - Total badness = the previous entry
            - M[chop][people] = M[chop-1][people]
- To satifsy the existence of C, chop >= people*3

Thought process about finding C

- To determine whether there is a C is tricky.
    - For some test case M[N][K] is solution but for some are M[N-2][K]
    - It's hard to deal with the last couple chopsticks to satify the C condition
    - In another word existing C is not guaranteed.
- Finally we figure out that we can sort chopsticks in decent order
    - In this way a C for each set/people is guaranteed.

10069 Distinct Subsequences

Set up

It was written in Java 1.9. Compile by command `javac Main.java`.

Then run by command `java Main`.

It has been accepted by Uva.

Observation

The question is to find the same subsequence in a given sequence. As we can see, it is not necessary for indices of sequence to ensure all letters are continously, but must be in the increasing order to consist to the subsequence.

In addition, the question is to count the number of occurrences and which indicates it is high possibility to use dynamic programming to decrease the complexity.

Algorithm overview

I treat X as a "sequence", and Z as a "subsequence".

Since the length of sequence is less than 10,000,the regular int type cannot hold such long indices of sequence. Therefore, a object of Java, BigInteger, can hold this requirement. There are two arguments in input strings. We need to have a 2D BigInteger array to hold these two arguments. And the size of 2D array depends on the size of given sequence and subsequence.

Then, we step into this question. We need to compare sequence and subsequence letter by letter, and use dynamic programming to save time. When the letter of sequence is equal to the letter of subsequence, we need to add its previous result of sequence and previous result of subsequence, and save the result to 2D array. Otherwise, update current result as the previous result of sequence.

Formally,

`if X[i] == Z[j], dp[i][j] = dp[i-1][j] + dp[i-1][j-1];`

`if X[i] != Z[j], dp[i][j] = dp[i-1][j];`

Data structure and Speical library

- Static 2D array of BigInteger
- BigInteger object and its methods and fields
    - BigInteger.One;

- BigInteger.Zero;
  BigInteger add(BigInteger val);

116 Undirectional TSP

Set up

The program is written using C++ with `c++11` standard. Compile the program by using the command `g++ -std=c++11 116.cpp`.

How it works

- The program starts by reading the dimensions, user input data and stores the data matrix in an $m \times n$ 2-dimentional vector.
- In order to find the smallest weight along the path, there are m posibilities to choose from the first column. The program starts by calling the recursive function `get_min(int row, int col)` with row from 1 to *m* and column equals to 0 indicating that the program starts looking from the first column (i.e. the solusion would be `min{ get_min(1, 0), get_min(2, 0), ... , get_min(m, 0) }`).
- Following the recursive rule `get_min(row, col) = min{ get_min(row + 1, col + 1), get_min(row, col + 1), get_min(row - 1, col + 1) }`, the function will then recursively call itself to find out the weight for the cases where we choose to go east, northeast and southeast. After we have 3 weights, we can determine which is the smallest weight and mark that tile as part of out path by adding the current row number to the path vector returned by the previous recursive function.
- The recursive process continues until we hit the very last row (i.e. when `col > input_column`). The program will return the pair of weight 0 and an empty path vector.
- At the end of therecursive process, we will know each of the weights and paths if we choose to go through the *m* posibilities. We then choose the smallest weight and the corresponding path as the final answer.

Note: During the recursive process, we need to have a map implementing the functionality of memoization with key that stores the (x, y) coordinates and value that stores the pair of total weight summing from the current coordinate until the right most tile and a vector denoting a path.

1222 Bribing FIPA

Set up

The program is written using C++ with `c++11` standard. Compile the program by using the command `g++ -std=c++11 1122.cpp`

Algorithm and dp set up

- A tree structure should be appiled from observations as follows:
  - n,m is quite small (200), making DFS, a common way to fill DP-tree possible
  - We can see a clear tree structure from the domination relation between countries.
- Transfer function:
  - **dp[u][j] = min(dp[u][j], dp[u][j-k]+dp[v][k]);**
  - Get dp's value from its children
- Use a 2d table to build a tree
  - Root node is a node to connect all countries
  - All countries are at the second level
  - Country under domination are child
    - for example
      - Aland is Boland's child so
        - dp[Boland][1] = Boland
        - dp[Boland][2] = Aland

Why we fail

- Input is very complicated
  - Countries' name are raw string so when we are trying to use it as dp table index we need to index it first
  - **Maintaining additional map to map the country name to ID add complexity to implementation**
  - For each line there is no number of children at hand
  - "#" set up is quite stupid
- Detail in dp implementation is complicated too
  - we believe we have the correct dp set and algorithm but we are missing one or two detail in implementation
  - Details we covered
    - Set infinity value to a value that can be added up without overflow cause INF+INF would happened
    - Init cost[0] to INF
    - Final result should be minimum value in dp[i][j] for all j > m