DSA Final Project Report

Team ID: `team33`
Members: B09902004, B09902011, B09902100

> Refs:
> https://en.wikipedia.org/wiki/Shunting-yard_algorithm

Data Structure & Algorithms

`Find-Similar`

We use a hash table with tokens as keys to store mail ids that have that token. Mail ids are stored in both a linked list and an array, so that knowing both whether a given mail has this token can be done in $O(1)$ and getting all mail ids with this token can be done in $O(mails\ with\ this\ token)$. Building this hash table takes $O(mails \times token\ per\ mail)$.

As this hash table is being built, we also store the unique tokens of first 1000 mails. Then the intersection table is built by going through the unique tokens of each mail, and check for any other mails which also have this token by going through the linked lists in the hash table. This way we can calculate the number of intersecting tokens of each mail pair.

We then build a 1000×10000 table (offline) to store $|A \cap B|$ using the data computed above, and put it in the source code. In the code submitted to the judge, we simply loop over the table and calculate the similarity to all other mails. We then can respond the query in $O(mails)$ time.

`Expression-Match`

We first convert the infix expression to postfix in $O(length)$ time, then evaluate the expression in $O(length \times mails)$ time.

Infix to postfix is done using the Shunting-Yard algorithm. To evaluate the the postfix expression, we use a stack to store boolean values. If we get a token, push the mail id array. If we get a operator, pop the values, apply operation, then push the result back.

`Group-Analyse`

In this part, we apply a disjoint set model to solve this type of query.

First, we analyse the users in given mails, and find a method to hash these users' name that any user won't collide with others.

In `main.c`, for each GROUP-ANALYSE query, we will initialize an empty disjoint set at the begining. Then, we can get the information of *From* and *To* from given `mid`. For each user that hasn't appeared in this disjoint set n_g will add 1. If *From* and *To* are not in the same disjoint set, we will merge them together and n_g will minus 1. However, there is a special case that *From* equals to *To*. In this type of mail, n_g will not add 1 since this node cannot be a group. For each union operation, it will return the size of the disjoint set, and l_g will be substitute if the return value is bigger than l_g. At the end, n_g and l_g will be submitted through `api.h`.

Cost Estimation of Queries

`Find-Similar`	`Expression-Match`	`Group-Analyse`
$O(mails)$	$O(length \times mails)$	$O(mails \times \alpha(mails))$

Scheduling Strategy

Because `Find-Similar` on average has 80 times the reward for each answer ~~, which is perfectly balanced,~~ we decide to tackle this problem first. And since the source code file size is limited, we only want to answer queries with `mid` less than 1000. This gave us a glorious score of about 600000.

We tried only answering `Expression-Match`, but got merely 5000 points and was far from finishing all the `Expression-Match` queries. Answering only `Group-Analyse` gets us 50000 points but doesn't seems to finish all queries.

Therefore, we decide to finish all `Find-Similar` queies first, then answer `Group-Analyse` queries with above average rewards. This gives as 700000 points in total.

Additional Notes

For `Find-Similar`, after some offline tests, it turns out that calculating all the 10000×10000 similarity in the program is simply unviable. So we figure we can try and precompute some part of it, and surprisingly it works great.