Data Structure and Algorithm, Spring 2021
Final Project - Email Searcher 2.0 Spec

Competition & Report Due: 23:59:59, Tuesday, June 22, 2021

TA E-mail: dsa_ta@csie.ntu.edu.tw

Instructions

- This document describes the spec of final project. Please refer to the initial announcement for other rules.

Header File

In this project, you can only access the data and reply the answer through functions provided in a given header file, `"api.h"`. **You must include this header file as the first header file in your submission.** Before introducing the provided functions, we first introduce the data types predefined in `"api.h"` for mails and queries.

Predefined Types

Mail Type

The type `mail` is defined as follows.

```
typedef struct mail {
    int id;
    char from[32], to[32], subject[256], content[100000];
} mail;
```

Each mail will have an integer within the range of `[0, n_mails - 1]`, where `n_mails` is the number of mails to be discussed later, as its unique mail ID. Each of the other entry will store a printable string that ends with `'\0'`. The set of mails is guaranteed to be a subset of the released mail data of last year's final project located at

https://www.csie.ntu.edu.tw/~htlin/course/dsa20spring/project/.

Query Type

The type `query` is defined as follows.

```
typedef struct query {
    int id;
    double reward;

    enum query_type { expression_match, find_similar, group_analyse } type;
    union query_data {
        struct { char expression[2048]; } expression_match_data;
        struct { int mid; double threshold; } find_similar_data;
        struct { int len, mids[512]; } group_analyse_data;
    } data;
} query;
```

The `type` entry stores the type of query. Then, the corresponding data is stored in the entry at `data.*_data`, where `*` is the string that matches the type. For instance, if the type is `expression_match`, the corresponding data can be access through the `data.expression_match_data` entry. Each query will also have its reward in the `reward` entry, and its unique query `id`. **For simplicity, we will take the penalty of returning a wrong answer to the query to be -0.5 times the reward.**

Helper Functions

Initialization Function

We provide a function called `api.init`, which helps receive the mails and queries. **Noted that this function can only be called within the first 30 milliseconds of your program execution.** The timer of judging your program begins when `api.init` returns. The function prototype is defined as follows.

```
void api.init(int *n_mails, int *n_queries, mail **mails, query **queries);
```

You are supposed to pass two integers by their addresses (i.e. call by reference in C) as the first two arguments. The function will then save the number of mails (and queries) in the corresponding integer before returning. You are also supposed to pass two pointers by their addresses (i.e. call by reference in C) as the last two arguments. The function will then allocate two arrays, one for storing mails and the other for storing queries, and save the address of the array in the corresponding pointer.

Answering Function

To reply to queries, we provide a function called `api.answer`. The function prototype is defined as follows.

```
void api.answer(int query_id, int answer[], int answer_length);
```

For each query, you need to store the answers in an array and reply with this function so that the answer will be judged. **Note that for each query, only the first attempt of answering will be judged, and subsequent attempts will be ignored.** If your program reaches the time limit within the answer function, the program can terminate without completing the answer function.

Example Usage

```
#include "api.h" /*"api.h" must be include as the first header file.*/

int n_mails, n_queries;
mail *mails;
query *queries;

int main(){
    api.init(&n_mails, &n_queries, &mails, &queries); /* initialization */
    for(int i = 0; i < n_queries; i++){
        /* guessing no-match for all expression-match queries */
        if(queries[i].type == expression_match)
            api.answer(i, NULL, 0);
    }
    return 0;
}
```

Query

There will be three types of queries, Expression-Match, Find-Similar and Group-Analyse, each with its own requirements and rewards. We start with some definitions first.

Definitions

Tokens

Tokens, the basic components in expressions, are defined as successive alpha-numeric characters, or `/([A-Za-z0-9]+)/g` in regular expression. That is, all non-alpha-numeric characters can be viewed as delimiters of the tokens. For example, `"dsa2021"` and `"mails"` are tokens while `"A+"` is not (only `"A"` within `"A+"` is a token). Also, the `"ppl"` within `"apple"` is not a token. Here is another example of tokens in a mail's content. Furthermore, tokens are **case-insensitive**, which means `"Apple"` and `"apple"` represent the same token. When speaking of the token set of a mail, it refers to the set formed by the tokens within the mail's subject and content.

Context Similarity

The context similarity of two mails is defined as $\frac{|A \cap B|}{|A \cup B|}$, where A, B denotes the token sets of the two mails.

Expressions

Expressions are recursively defined in the order of precedence as follows, let `[token]` and `[expr]` denotes arbitrary token and expression.

- `[token]` - a single-token expression returns true on a mail iff the mail contains the token in its subject or content
- `([expr])` - an expression wrapped by parentheses evaluates to true iff the internal `[expr]` evaluates to true
- `![expr]` - an expression negated by `!` evaluates to true iff the `[expr]` evaluates to false
- `[expr]&[expr]` - two expressions with `&` evaluates to true iff both `[expr]`'s evaluate to true
- `[expr]|[expr]` - two expressions with `!` evaluates to true iff at least one of the `[expr]`'s evaluates to true

Given that `[token]` and all expression operators do not contain white spaces, `[expr]` will not contain any white spaces as well.

Query Tasks

Expression Match

In the EXPRESSION-MATCH query, an `expression` will be given. To answer such query, you need to find all the mails on which the expression returns true. Then, return the matching mail IDs in **increasing order** through `api.answer`. As the type `query` suggests, the length of expression is restricted to 2048 characters, including the ending null byte. We also guarantee that all the expressions are legal, and all the tokens exist in certain mails.

Find Similar

In the FIND-SIMILAR query, a mail ID `mid` and a threshold `threshold` is given. You need to find all the other mails, excluding `mid`, that are of context similarity higher than the `threshold` with respect to `mid`. Then, return those mail IDs in **increasing order** through `api.answer`. We guarantee that the mail ID will be valid and the threshold is within $[0, 1]$. We also guarantee that the threshold would be chosen such that you can directly compare your similarity to the threshold without worrying about the floating point precision.

Group Analyse

In the GROUP-ANALYSE query, you will be given a subset of mails by their IDs and need to analyse on the equivalence relation among the senders and recipients in the mail set. The size of the set is stored in `len` and the mail IDs are stored from `mids[0]` to `mids[len-1]`. You will only need to consider all senders and recipients in the mail set for each query. The relation between the senders and recipients is defined as follows.

- If A sends a mail to B in the mail set, then A is related to B and B is related to A.
- If A is related to B and C is related to B, then A is related to C and C is related to A.

With the relation definition, a group is defined as a set such that every member is related to every other member within the set. In this task, you need to calculate the number of groups n_g, and the size of the largest group l_g. Then, return an array of length 2 through `api.answer`, where the first value of the array stores n_g and second stores l_g. We guarantee that each mail ID will be valid and unique. In addition, the mail set will not be empty.

Task Rewards

As you might have discovered, the three types of queries come with different difficulties. Therefore, different types of queries are associated with different rewards. Basically, the reward R of

a certain task is roughly normal distributed, $R \sim \mathcal{N}(\mu, \sigma^2)$, but each query type is associated with a different (μ, σ^2).

Local Testing Environment

Besides the DSA Judge, a local test environment will be released for the ease of testing your code without wasting quotas. However, the testing environment only supports Unix-based systems. For those who uses the Windows Operating System, we will also release a Repl.it Template for you to fork and test your solution.

The score shown in test environment is only for your reference. We will only take the score on the DSA Judge scoreboard as the grading criteria. So please remember to submit your best solutions to the DSA Judge. Also, it is common that the score fluctuates in this kind of performance competitions. So we decide to increase the submission quota from 5 to 10 each day.

It is worth knowing that the input format and hash function used will be different between the local testing environment and the DSA Judge environment. Therefore, please follow the guideline and use the api provided to access the data, instead of manipulating the data by yourself in any manner. Failure to follow the access api could mean violating the course policy and may result in serious penalties.

Baselines

The baselines will be release later (at last 2 weeks a head of due date).