Todo Swamp

Serokell Test Task

Artem Ohanjanyan
<artem.ohandzhanian@serokell.io>*

Kirill Andreev
<kirill.andreev@serokell.io>*

Jonn Mostovoy
<jm@serokell.io>

Nikolay Yakimov
<nikolay.yakimov@serokell.io>

May 2019 - January 2023

Problem statement

Your task is to complete our implementation of a simple CLI to-do list. Find a project with typed holes to fill under `solution/` directory. It should support the following queries:

- `add "<description>" <tags>` — add a to-do item and print its index (indexes start at 0 and increase by 1);
- `done <i>` — mark an item with index `<i>` as done and print «done»;
- `search <query>` — print the indexes of items which match `<query>` and, optionally, the text of the item.

Where

- `<description>` is a string of `<word>`s, separated by space characters;
- `<tags>` is zero or more `<tag>`s separated by space characters;
- `<word>` is one or more lowercase Latin alphabet characters or a hyphen ($\{a, \ldots, z\} \cup \{\texttt{-}\}$);
- `<tag>` is a `#` followed by one `<word>`;
- `<i>` is a non-negative integer;
- `<query>` is zero or more `<word>`s or `<tag>`s separated by space characters.

We call a string «$s_1 s_2 \ldots s_n$» a *subsequence* of string «$t_1 t_2 \ldots t_m$» if and only if

$$\exists k_1 < \ldots < k_n \forall i \in \{1 \ldots n\} : s_i = t_{k_i},$$

Or, informally, a subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Character equality $s_i = t_{k_i}$ is case insensitive. A `<word>` s in the `<query>` matches items with descriptions that have a word t, such that s is a subsequence of t.

A `<tag>` $\#s$ in the `<query>` matches items which have a tag $\#t$ such that s is a subsequence of t. A `<query>` matches items which are matched by all `<word>`s and `<tag>`s in the `<query>`. If `<query>` is empty, it matches all items.

*These E-Mail addresses are no longer available.

Input format

The first line in STDIN contains a single natural n (possibly followed by spaces) — the number of queries. Each of the next n lines in STDIN contains one query. Results should be printed to STDOUT.

A query is either:

- `add`, followed by a double-quoted line of text and 0 or more words or tags, separated by space characters;
- `done`, followed by an index of the item to be excluded from the search from now on;
- `search`, followed by 0 or more words or tags to be used as subsequences in search.

The tester doesn't wait for output before sending more input, hence it's feasible to batch-process. However, be aware that the total CPU time is limited (see Running submissions), so if you run out of time while processing a batch, your program will be stopped before getting the chance to output, and you won't get any score for that batch.

Test data

The items to be added, the words, and tags are made from a list of 7775 English words, which are *slightly* warped (example: `add "lyxunpmmy ceoe" #hanice #aofmhimbe #esoule #ughgigf`). Two warping strategies are employed:

5%-replace-20%-drop
: A Bernoulli trial with a 20% success probability is performed for each letter. If the trial succeeds, the letter is dropped. Otherwise, another Bernoulli trial is performed with a 5% success probability, and if that succeeds, the letter is deterministically replaced by another letter. If both trials fail, the letter is left as is.

25%-insertion
: Two independent Bernoulli trials, with a success probability of 25% each, are performed for each letter. If the former is successful, a random letter is inserted before the current one, if the latter is successful, a random letter is inserted after the current one.

The choice of warping strategy depends on the test itself and the choice of char positions to be warped depends on the random seed chosen for the test.

Each item is a string of *0* or more warped (as described above) words, separated by space characters. If warping reduces a word to 0 symbols, it is omitted.

Test data generation

Four different generators are present:

- `Standard` — generates all `add` instructions first, then `search` ones, then closes all items with `done`.
- `RatioDone` — generates all `add`s, then closes a large percentage of them and then runs `search`es.
- `Interleaved` — generates all `add`s, then all `done`s. The `search`es are interleaved with the generated sequence.
- `Lattice` — a lattice-based approach, discussed further.

The first three use the 5%-replace-20%-drop strategy of word warping. The `Lattice` generator uses 25%-insertion as a warping strategy. Its purpose is to generate a huge input/output mapping. For it to be possible, during generation we need to be able to perfectly predict the output of each fuzzy search. To achieve this, instead of generating items, we are generating a lattice of possible queries for which we can determine the perfect output. Since we have a deterministic lattice of queries, we can now simulate the usage of the system over time (adding and marking items as done) by enabling and disabling points in this lattice. We run such a simulation and, again, due to determinism, it produces a huge input and an even huger output file with $O(n \cdot log(n))$ asymptotic.

Output format

Notice that the project must be present inside the `./solution` directory inside the `.tar` or `.tar.gz` archive that you provide as the submission. Our automated testing system will reject any other archive formats. Also, please remove build artifacts, testing, and all other unrelated data, so that the archive size does not exceed 1 Mb. If the archive contains no `./solution` directory or that directory contains no buildable project, the solution will be rejected with a score of 0.

Each test will contain:

- count of commands to execute
- `search` queries:
  - word queries — with words that match whole words or subsequences of words;
  - tag queries — tags that match whole tags or subsequences of tags;
  - mixed queries — queries that are both word and tag queries;
- two other commands from the specification: `add` and `done`

There may be trailing whitespace characters after each command. There must be no trailing whitespace characters after responses. The order of `add`, `done` and `search` commands is arbitrary, `done N` command cannot precede adding the item with the number N. If a search query returns multiple items, their order does not matter. A valid response is:

- for `add` - An index of added item, starting from 0;
- for `done` - a string `done`;
- for `search` - an `N item(s) found` line, where `N` is the number of search results; it should be followed by `N` lines, each containing an index of the item found and (optionally) its text representation. The checker program ignores the text representation. Even if a search returns no results, the line `0 item(s) found` should still be present.

Scoring details

- Each search query result printed correctly adds 1 to your score.
- Each incorrect search query result subtracts 5 from your score.
- Each misplaced or incorrectly printed response to other queries also subtracts 5 from your score.

Notice that the responses are compared on a one-to-one basis. Missing a response or adding an unnecessary one will have a big negative effect on the overall score. That means that if your heuristic decides to ignore a *search* query, it still *MUST* print `0 item(s) found` to STDOUT to avoid drowning in negative scores.

Problem size

The maximum input size is $5 \cdot 10^6$ commands except for a single Lattice input where you can gain points on a virtually unlimited amount of data. While algorithms and data structures matter in this problem, a lot can be achieved with non-asymptotic optimizations, so we invite you to come up with heuristics to achieve higher scores! On small tasks (up to 10,000 maximum score), we expect your solution to provide complete answers - so all the search queries should produce precise results in terms of identities of items found. Finally, in the code shipped together with this specification, there may be inconsistencies with the said specification. In case of such conflict, the specification trumps the code, and we invite you to fix such inconsistencies in the code.

Running submissions

Submissions will run on a multi-core system with access to several gigabytes of RAM and *total* CPU time limited to 10*s*. Haskell submissions will be built with GHC resolved by `lts-22.32` resolver. Rust submissions will be built with current `cargo stable`. If you have any questions or problems with the tooling, including the compiler version, please contact us via `<jobs@serokell.io>` and specify a subject line as `"Test Task D: Haskell Issue"` or `"Test Task D: Rust Issue"`, depending on the language you're using. We invite you to send your submissions as a tarball to `<jobs@serokell.io>` and specify a subject line as `"Test Task D: Haskell Submission"` or `"Test Task D: Rust Submission"`, depending on the language you're using.

Appendix A: Example

```
 stdin  10
 stdin  add "buy bread" #groceries
stdout  0
 stdin  add "buy milk" #groceries
stdout  1
 stdin  add "call parents" #relatives
stdout  2
 stdin  search #groceries
stdout  2 item(s) found
stdout  1 "buy milk" #groceries
stdout  0 "buy bread" #groceries
 stdin  search buy
stdout  2 item(s) found
stdout  1 "buy milk" #groceries
stdout  0 "buy bread" #groceries
 stdin  search a
stdout  2 item(s) found
stdout  2 "call parents" #relatives
stdout  0 "buy bread" #groceries
 stdin  done 0
stdout  done
 stdin  search a
stdout  1 item(s) found
stdout  2 "call parents" #relatives
 stdin  done 2
stdout  done
 stdin  search a
stdout  0 item(s) found
```

Appendix B: Lattice generation

The `Lattice` generator builds a lattice graph of search items first, all of them are initially deactivated. To achieve that, it selects a bunch of words from the global word array and randomly assigns each word to either the tag or the description of an item. Each initial item produced this way either has one tag or one word in the description. Then the generator repeatedly cross-breeds items on the top level (i.e. the ones with no descendants) by merging their word sets and tag sets, respectively, until the required amount of layers is filled.

The `sup` relation of the lattice generated is a subsequence one, and its bottom layer is made of singular tag-or-word items. Care is taken, so each item that is a parent to another is linked back in the generated graph.

Then, the generator starts making `add`, `done` and `search` commands:

- `add` — an item at a random point is activated; its contents become the body of `add` command. The contents of the new item are bloated with random chars, to make the task more interesting and to prevent graph reconstruction.
- `done` — a random item previously activated by `add` is deactivated.
- `search` — a random item is selected. It becomes a search query, and both itself and its *alive* children become the search results.