

Task 11 - RegExp parsing

Running tests and code coverage:

```
1 cmake -DCMAKE_BUILD_TYPE:STRING=Debug .. && make coverage_report
```

You can check coverage in `build/parser-coverage.html`.

Даны α , буква x и натуральное число k . Вывести длину кратчайшего слова из языка L , содержащего ровно k букв x .

Algorithm description and correctness (см. pdf)

```
ab + c.aba. * .bac. + . + * b 2
```

Алгоритм представляет из себя динамику на стеке. Будем двигаться по записи регулярного выражения и для каждого подслова выражения, которое само является регулярным выражением искать ответ (проверять, соответствует ли подслово условию + поддерживать минимальность ответа), потом будем искать ответ для какой-либо композиции этих выражений и т.д., пока не распарсим полностью ввод.

Обозначения:

- regular - выражение в польской записи
- stack - стек обработки
- len - то же самое, что и k
- letter - то же самое, что и x

Корректно обрабатывать выражение в польской записи позволяет стек.

```
1 std::vector<DpHandler> vec = stack.top()[i]
2 // - вектор размера len + 1, отвечает какому-то регулярному выражению.
3
4 vec[i] = {bool has_right_cnt, int min_len_of_correct} // пара аргументов
```

$$\forall i \in [0, k] \quad vec[i].has_right_cnt = \begin{cases} true, & \text{в данном выражении можно найти слово с ровно } i \text{ буквами } x \\ false, & \text{иначе} \end{cases}$$
$$\forall i \in [0, k] \quad vec[i].min_len_of_correct - \text{минимальная длина такого слова}$$

Корректная работа обеспечивается переходами между состояниями ДП разбором случаев для каждого типа операций (+ - union, . - concat, * - star). Рассмотрим эти случаи.

1. $RegExp = c, c \in \Sigma$

В случае одной буквы нас интересуют два случая - $i = 0$ и $i = 1$. Они разбираются так:

$$vec[0].has_right_cnt = \begin{cases} false, & c = letter, \text{ т.е. выражение не имеет слов без букв letter} \\ true, & c \neq letter, \text{ т.к. можем вывести слово - саму букву} \end{cases}$$
$$vec[0].min_len_of_correct = \begin{cases} -1, & c = letter \\ 1, & c \neq letter \end{cases}$$

(-1 в случае `vec[0]` - деталь реализации, сигнализирует о том, что выражение - буква `x`, позволяет отличить данный случай в функциях от случая `vec[0] = {true, 0}`, который отвечает замыканию Клини (действительно, минимальное по длине подходящее слово - пустое))

$$vec[1].has_right_cnt = \begin{cases} false, & c \neq letter, \text{ т.е. выражение не имеет слов с буквами } letter \\ true, & c = letter, \text{ само выражение есть } letter \end{cases}$$

$$vec[1].min_len_of_correct = \begin{cases} 1, & c = letter \\ 0, & c \neq letter \end{cases}$$

Остальная часть массива заполняется парами по умолчанию (`false, 0`). Отправляем `vec` в стек.

Асимптотика: $O(len)$

2. $RegExp = RegExp1 \cup (+) RegExp2$

Сложение двух выражений означает то, что мы вольны выбрать `i` символов либо из первого, либо из второго. Т.е, если хотя бы в одном есть слова с `i` символами, то в сумме такие слова тоже есть. Поддержать минимальность также несложно, достаточно взять (в общем случае), минимум по `vec[i].min_len_of_correct` первого и второго.

Тем не менее, в коде обработка разделена на две части базовая - для `vec[0]`, и для остальных `vec[i]`.

База:

Пусть `RegExp1` отвечает вектор `one`, `RegExp2` отвечает вектор `two`.

$$a := one[0].min_len_of_correct, \quad b := two[0].min_len_of_correct$$

Case0: оба выражения - буквы `letter`.

В таком случае, мы не можем найти слово с 0 буквами `letter`, `vec[0] = {false, 0}`

Case1: Одно из выражений - буква `letter`.

Тогда `vec[0] = {true, correct_len}`, где `correct_len` - минимальная длина неоднобуквенного выражения

Case2: Оба выражения невырожденные

$$vec[0] = \{true, \min(len1, len2)\}, \text{ где } len_i - \text{ мин. длина соответственного выражения}$$

Основные случаи:

Действуем так, как написано выше в описании секции. Отправляем `vec` в стек.

Асимптотика: $O(len)$

3. $RegExp = RegExp1 \cdot RegExp2$

Как мы можем получить `i` символов `letter` в конкатенации? Мы можем взять `j` символов из `RegExp1` и

`(i - j)` символов из `RegExp2`. Тогда длина такого слова будет суммой длин слов регулярных выражений. Традиционно, рассмотрим базовый случай. `one`, `two` - обозначения для выражений.

Case0: фиксируем `one[0] = {has_right_cnt, min_len_of_correct}`

```

1 for i in range(1, len+1):
2     cnt_in_concat = j * two[j].has_right_cnt
3     # слово с cnt_in_concat буквами letter
4     # если has_right_cnt = false, то в конкат. нет letter (в one тоже)
5     if (cnt_in_concat != 0):
6         vec[cnt_in_concat].has_right_cnt = true
7         vec[cnt_in_concat].min_len_of_correct = ....

```

`vec[cnt_in_concat].min_len_of_correct` при этом равен сумме длины `one[0]` и длины `two[j]`, либо же сумме 1 и длины `two[j]` в вырожденном случае, когда `one[0].min_len_of_correct = -1` (выражение - сама буква letter).

Аналогично разбирается случай зафиксированного `two[0]`.

Case1: Как можем посчитать значение для `vec[0]`? Рассмотрим случай `one[0]` и `two[0]` для `vec[0]`. Введем соответств. обозначения a, b .

Если хотя бы одно выражение - буква, то нельзя в конкатенации найти таких слов, в которых не было бы буквы letter. Поэтому `vec[0] = {false, 0}`.

Иначе, `vec[0] = {true, a + b}`

Основные случаи:

Как было сказано выше, будем набирать i букв letter составлением комбинаций слов `RegExp1` и `RegExp2` с разными количествами букв letter. Случаи, когда в комбинации присутствует слово с 0 буквами letter, разобраны выше.

Пройдемся двумя циклами по состояниям `RegExp1` и `RegExp2`:

```

1 for i in range(1, len+1):
2     for j in range(1, len+1):
3         cnt_in_concat = i * one[i].has_right_cnt + j *
           two[j].has_right_cnt;

```

`cnt_in_concat` - линейная комбинация, показывает, как мы можем набрать такое число букв:

Если в `one[i]` и в `two[j]` есть ровно i, j букв, то количество букв в конкатенации будет в точности этой комбинацией.

Иначе, мы все еще можем набрать `cnt_in_concat` букв, но только в комбинации с `one[0]` или `two[0]` и когда в одном из выражений есть ровно `cnt_in_concat` букв. Эти случаи были рассмотрены в базе.

В других случаях, когда `one[i].has_right_cnt = false` (`two[j].has_right_cnt = false`) и $i \neq 0$ ($j \neq 0$) - составить правильную комбинацию мы **не можем**, потому что одно из выражений имеет ненулевое количество букв letter. Поэтому далее в коде проверяется условие:

```

if (cnt_in_concat != 0 && cnt_in_concat <= len && one[i].has_right_cnt &&
two[j].has_right_cnt)

```

Если условие соблюдено, то `vec[cnt_in_concat] = {true, len1 + len2}`, где a, b - минимальные длины `one[i], two[j]`.

Минимальность поддерживается за счет корректно подсчитанных минимальных длин в `one[i], two[j]`.

Отправляем `vec` в стек.

Асимптотика: $O(len^2)$

4. $\text{RegExp} = (\text{RegExp1})^*$

Замыкание Клини есть сумма степеней RegExp1 :

$$(\text{RegExp1})^* = \sum_{n=1}^{\infty} (\text{RegExp1})^n$$

Каждая итерация выглядит как: $\text{RegExp} += \text{RegExp} \cdot \text{RegExp}$.

Сколько нужно итерироваться, чтобы гарантированно рассмотреть случаи $i = 1, \dots, \text{len}$?

На каждой итерации мы либо не изменяем состояния vec (когда при дальнейших итерациях получаются слова, с количеством букв $\text{letter} > \text{len}$), либо изменяем хотя бы одно состояние $\text{vec}[i]$. Поэтому точно хватит len итераций.

В конце вне цикла надо не забыть поставить $\text{vec}[0] = \{\text{true}, 0\}$ - минимально возможное слово без букв letter - пустое.

Каждая итерация - concat и sum , поэтому:

Асимптотика: $O(\text{len}^3)$

Ответ: В конце работы алгоритма в стеке останется один элемент, который будет соответствовать всему разобранному выражению. Как и в любом дп, ответом будет являться $\text{stack.top()}[\text{len}]$

Итоговая асимптотика: $O(|\alpha| \cdot \text{len}^3)$ - самые тяжелые операции здесь за куб, их количество не более длины регулярки.