МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЁТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритма Ахо-Корасик поиска набора образцов в строке.

Задание

- Задание 1

  Разработайте программу, решающую задачу точного поиска набора образцов.

  Вход:

  Первая строка содержит текст (T, $1 \leqslant |T| \leqslant 100000$).

  Вторая – число n ($1 \leqslant n \leqslant 3000$), каждая следующая из n строк содержит шаблон из набора $P = \{p_1, \ldots, p_n\}$, $1 \leqslant |p_i| \leqslant 75$.

  Все строки содержат символы из алфавита $\{A, C, G, T, N\}$

  Выход:

  Все вхождения образцов из P в T.

  Каждое вхождение образца в текст представить в виде двух чисел – i p.

  Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

  Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

  Sample Input:

```
NTAG
3
TAGT
TAG
```

```
T
```

Sample Output:

```
2 2
2 3
```

- Задание 2

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который „совпадает“ с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения в текст .

Например, образец $ab??c?$ с джокером $?$ встречается дважды в тексте $xabvccbababcax$.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида $???$ недопустимы.

Все строки содержат символы из алфавита $\{A, C, G, T, N\}$.

Вход:

Текст (T, $1 \leqslant |T| \leqslant 100000$)

Шаблон (P, $1 \leqslant |P| \leqslant 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

```
ACTANCA
A$$A$
$
```

Sample Output:

```
1
```

Вариант 1

На месте джокера может быть любой символ, за исключением заданного.

Описание алгоритма

Алгоритм Ахо-Корасик заключается в построении конечного автомата на основе набора шаблонов, и последующей обработке строки, в которой осуществляется поиск, с использованием этого автомата.

Автомат строится на основе дерева ключей (*бора*), полученного из множества образцов. Дерево ключей – это дерево, каждое ребро в котором помечено

Бор получается путём добавления каждого образца в него. Изначально бор состоит из одной вершины – корневой. Очередной образец добавляется в бор следующим образом:

1. Начинаем из корневой вершины и с первого символа строки.
2. Ищем ребро, из текущей вершины, помеченное первым символом строки. Это ребро не должно совпадать с тем, по которому мы пришли в эту вершину.
3. Если такое ребро есть, переходим по нему.
4. Если такого ребра нет, добавляем в дерево новую вершину и пускаем ребро из текущего элемента в новый. Помечаем это ребро текущим символом, и переходим по нему в новую вершину

5. Повторяем то же для следующего символа, пока строка не закончится.

Бор можно представить как конечный автомат. Тогда каждая вершина будет состоянием (корневая вершина – начальное состояние), а каждое ребро – переходом по символу, которым помечено ребро. Для каждого состояния v назовём состояние, из которого можно прийти в v его предком.

Для каждого состояния v определим строку, которой оно помечено, как строку, которой помечен его предок u, с приписанным в конце символом, которым помечен переход из u в v (корневое состояние помечено пустой строкой). Иначе говоря, это строка, состоящая из символов, которыми помечены переходы, которые надо будет пройти, чтобы добраться до v из корня.

Каждому состоянию также соответствует набор образцов, вхождения которых будут обнаружены при переходе в данное состояние. Обозначим этот набор o_v для состояния v. Изначально скажем, что образец входит в o_v если v помечена строкой, совпадающей с этим образцом. Для каждого образца найдётся только одно такое состояние; в реализации его можно отмечать при построении бора.

В каждое состояние нужно добавить переход, который будет выбираться когда ни один переход, помеченный каким-либо символом, не подходит (*связь неудачи*; обозначим как f_v для состояния v). Из начального состояния мы в таком случае перейдём в него же, равно как и из всех состояний, в который можно попасть из корневого напрямую. Далее, выполним обход дерева состояний в ширину начиная из начального состояния (обозначим его r). В каждом состоянии выполним следующее:

1. Обойдём все переходы из текущего состояния.

2. Допустим, мы смотрим на переход по символу x в состояние u. Найдём f_u. Для этого попробуем найти переход по символу x из состояния f_v (v – предок u). Если его нет, попробуем то же с состоянием f_{f_v}, и т.д.

3. Допустим, мы нашли переход в символ t. Тогда $f_u = t$, а иначе $f_u = r$.

4. Расширим набор o_u: добавим к нему набор o_{f_u}. Это нужно сделать потому, что строка, которой помечено состояние f_u, является собственным суффиксом строки, которой помечено состояние u.

Для вершины v, расположенной на удалении k от корня, f_v может быть расположена не далее, чем на удалении $k - 1$ от корня. Таким образом, при обработке вершины v при обходе дерева состояний в ширину, f_v, f_{f_v} и т.д., будут уже известны.

Поиск вхождений в алгоритме Ахо-Корасик выполняется следующим образом: по очереди берутся символы из строки, и для каждого символа выполняется один или несколько переходов по состояниям автомата. При каждом переходе, обозначающем совпадение символа с символом, которым помечен переход в автомате, расширяется множество совпадений образцов.

1. Из строки берётся очередной символ c.

2. Производится поиск перехода по символу c из текущего состояния v. Если такого перехода нет, выполняется переход в f_v и поиск производится там, и т.д. до тех пор, пока не будет найден подходящий переход, либо пока не завершится неудачей поиск ребра в начальном состоянии r.

3. Если подходящий переход, найден, выполняется проход по нему (допустим, в состояние u). Тогда множество совпадений расширяется на o_u (для каждого образца из o_u нужно будет рассчитать начало вхождения).

Описание алгоритма поиска шаблона с джокерами

Поиск одного шаблона с символом-джокером выполняется как поиск множества обычных образцов, но дополнительно для каждого символа строки по i хранится количество совпадений $c[i]$ частей шаблона, начало которого приложено к i-му символу.

Множество образцов получается разделением шаблона на куски, не содержащие джокер. Например, из шаблона $ab?de???ijk?$ ($?$ – джокер) получится 3 куска – ab (сдвиг 0), de (сдвиг 3), а также ijk (сдвиг 8).

Допустим, было обнаружено совпадение какого-либо образца, расположенного в шаблоне со сдвига l. Совпадение начинается в символе строки с индексом k. Тогда весь шаблон начинался бы с индекса $k - l$, поэтому значение $c[k - l]$ увеличивается на 1.

Если $c[i]$ становится равно количеству образцов, это означает, что обнаружено совпадение всего шаблона.

Если на месте некоторых вхождений джокера не может быть некоторых символов, можно использовать каждый такой джокер как отдельный образец длины 1. Если такой образец совпадает, соответствующий индекс строки i помечается специальным образом, и совпадения всего шаблона в нём не будет, вне зависимости от значения $c[i]$.

Нужно отметить, что при поиске шаблона длины m достаточно хранить только последние m значений $c[i]$.

Описание функций и структур данных

Структуры данных, представляющие автомат, общие для обоих задач:

- `Result` – связный список образцов, совпадения которых обнаруживаются в данном состоянии.

```
struct Result {
    size_t pat_idx;
    size_t length;
    Result *next;
};
```

`pat_idx` – номер образца, `length` – его длина.

Связный список используется для того, чтобы можно было легко добавить к списку одного состояния список другого.

- `Transition` – переход в другое состояние по определённому символу.

  ```
  struct Transition {
      char ch;
      State *dest;
  };
  ```

  `ch` – символ, по которому выполняется переход в состояние `dest`.

- `State` – состояние.

  ```
  struct State {
      std::vector<Transition> transitions {};
      State *fallback = nullptr;
      Result *results = nullptr;
      Result **results_back {&results};
  };
  ```

  Поле `transitions` – обычные переходы, а `fallback` – связь неудачи. Поля `result` и `results_back` обозначают список совпадений в данном состоянии.

  `results_back` используется, чтобы добавить список другого состояния к списку данного. Этот указатель содержит адрес поля `next` последнего элемента собственной части списка, либо адрес поля `results` данного состояния если его список пуст. Таким образом, чтобы добавить список другого состояния к текущему, надо записать адрес его первого элемента по адресу из `results_back`. Это используется в функции `forestIntoStateMachine`.

Многие функции также общие:

- `void addResult(State *stt, size_t pat_idx, size_t length);`

  Добавляет в состояние `stt` совпадение образца `pat_idx` длины `length`. Эта функция создаёт новую структуру `Result`, добавляет её в конец списка, и устанавливает указатель `results_back` в адрес её поля `next`.

- `Transition *findTransition(State *stt, char ch);`

  Ищет переход из состояния `stt` по символу `ch`. Если такого перехода нет, возвращает `nullptr`.

- ```
  void
  extendForest(State *root, const char *chars,
               size_t len, size_t idx);
  ```

  Добавляет новый образец в граф состояний с корневым состоянием `root`. Образец начинается по указателю `chars`, его длина равняется `len`, а в качестве `pat_idx` соответствующей структуры `Result` берётся `idx`.

  Эта функция добавляет в граф недостающие состояния. Для этого она проходит в графе путь из корня, который пришлось бы пройти, чтобы обнаружить совпадение с данным образцом, достраивая недостающие переходы и состояния.

  После того, как все состояния достроены, эта функция добавляет новое совпадение с данным образцом в конечное состояние. Для этого используется функция `addResult`.

- `State *findFallback(State *parent, char ch);`

  Ищет связь неудачи состояния, в которое можно перейти из `parent` по символу `ch`.

- `void forestIntoStateMachine(State *root);`

  Принимает корень `root` графа состояний и добавляет в этот граф переходы `fallback`, а также достраивает списки совпадений в вершинах. Алгоритм был подробно описан ранее.

  При расширении списка совпадений состояния эта функция записывает адрес первого элемента нового списка по указателю `results_back`, но не меняет значения `results_back`, поэтому „чужие“ совпадения можно
  ```

отбросить, записав `nullptr` по адресу в `results_back`. Это используется при уничтожении автомата в `destroyStateMachine`.

- `void destroyStateMachine(State *root);`

  Освобождает память из-под автомата с корневым состоянием `root`.

- `bool stepMatching(State **cs, char c);`

  Выполняет шаг алгоритма Ахо-Корасик, но не обрабатывает совпадения. Используется при обработке очередного символа `c` строки, в которой осуществляется поиск. `cs` – адрес указателя на текущее состояние.

  Эта функция возвращает `true`когда на данном шаге могут быть совпадения, и `false`в противном случае.

- `std::ostream & writeStateMachine(std::ostream &os, State *root);`

  Выводит информацию об автомате с корневым состоянием `root` на поток `os`.

Реализации поиска нескольких образцов и поиска шаблона с джокерами различается в том, как строится бор и как выполняется обработка совпадений (ну и ввод-вывод, конечно).

Функции и структуры данных, использующиеся при поиске нескольких образцов (aho-corasick.cpp):

- `State *buildForest(std::vector<std::string> patterns)`

  Создаёт корневое состояние и просто вызывает `extendForest` для каждого образца из `patterns`.

- ```
  struct Match {
      size_t pat_idx;
      size_t start_idx;
  };
  ```
  ```

Структура, обозначающая совпадение шаблона `pat_idx`. Совпадение начинается в с индекса `start_idx`.

Для этого типа также определён оператор сравнения, чтобы эти структуры можно было сортировать.

- ```
  std::vector<Match>
  getMatches(State *root, const std::string &str)
  ```

  Выполняет поиск шаблонов в строке `str` с использованием автомата с начальным состоянием `root`. Использует функцию `stepMatching`. При обнаружении совпадения с каким-либо образцом, вычисляет индекс его начала и добавляет совпадение в вектор, который потом возвращает.

При поиске вхождений шаблона с джокером, выполняется дополнительная обработка шаблона. Он разделяется на части, а также в нём обнаруживаются джокеры, на месте которых не может быть какого-либо символа (то есть джокер совпадает с дополнением множества, содержащего только этот символ).

Функции и структуры данных для поиска шаблона, содержащего джокеры (wildcard.cpp):

- ```
  struct PartInfo {
      size_t offset;
      const char *chars;
      size_t length;
  };
  ```

  Обозначает часть шаблона длины `length`, которая начинается со сдвига `offset` в шаблоне. На начало этой части шаблона также указывает `chars`.

  Поля `chars` и `length` используются только при построении бора, а поле `offset` нужно чтобы находить начало шаблона по совпадению его части.

- ```
  struct Complement {
  ```
  ```

```
    size_t index;
    char ch;
};
```

Обозначает джокер, который совпадает с любым символом, кроме `ch` и расположен по индексу `index`.

Поле `index` используется так же, как и `offset` в `PartInfo` – для вычисления начала шаблона.

- ```
  struct Pattern {
      std::vector<PartInfo> parts;
      std::vector<Complement> complement;
      size_t length;
  };
  ```

  Структура, в которой содержится вся информация о шаблоне. `parts` – части без джокеров, `complement` – джокеры, не совпадающие с конкретными символами, `length` – длина шаблона.

- ```
  Pattern
  createPatternStructure(const std::string &pat,
                         char wildcard, char complement);
  ```

  Делает структуру `Pattern` для шаблона `pat`, причём в качестве символа-джокера берётся `wildcard`, а в качестве джокера, не совпадающего с определённым символом – `complement`. Символ указывается после символа джокера. Например, если `complement='^'`, то `"^x"` будет обозначать любой символ, кроме `'x'`.

  Поскольку „дополнительный джокер“ в строке занимает во входной строке 2 символа, длина совпадающей строки не будет равна длине строки `pat`. Все сдвиги, которые записывает эта функция, равно как и длина шаблона (`Pattern::length`) измеряются в „логических символах“ – символах строки, в которой выполняется поиск, а не символах строки `pat`.

- ```
  State *buildForestFromPattern(const Pattern &pat);
  ```
  ```

Делает бор из шаблона `pat`. Работает как `buildForest`, но добавляет вначале все части шаблона, а потом все элементы `complement`.

Совпадения элементов `complement` отличаются от совпадений элементов `parts` по индексу `pat_idx`: если `pat.parts.size()` $= n$, то для элемента `complment` с индексом i поле `pat_idx` будет равно $n + i$.

- ```
  std::vector<size_t>
  getTotalMatches(State *root,
                  const Pattern &pat,
                  const std::string &str);
  ```

  Выполняет поиск шаблона в строке `str`. Шаблон обозначается автоматом в корневым состоянием `root` и структурой с дополнительной информацией `pat`.

  Эта функция использует `stepMatching`, а сама только обрабатыват совпадения.

  Каждое совпадение может обозначать или вхождение части шаблона, или символ, с которым не совпадает один из джокеров.

  – Каждому индексу в строке сопоставляется число совпадений частей шаблона, приложенного к этому индексу (изначально 0). Если обнаружено совпадение части шаблона, по его индексу, длине совпадения, и сдвиге начала части вычисляется сдвиг начало шаблона, и значение, соответствующее этому сдвигу, увеличивается на 1.

  – Каждому индексу также сопоставляется флаг. Флаг изначально равен `false`. Если этот флаг равен `true`, этот индекс нужно игнорировать, т. к. один из „дополнительных джокеров“ не совпадает.

    Таким образом, если обнарено совпадение элемента `complement`, вычисляется сдвиг начала шаблона и ему сопоставляется `true`.

  Совпадения, для которых индекс шаблона получается меньше нуля или
  ```

больше максимального ($m - n + 1$, где m – длина строки поиска, а n – длина шаблона), совпадение отбрасывается.

Совпадение всего шаблона записывается если после обработки области, к которой он мог бы быть приложен, флаг несовпадения элемента `complement` равен `false`, а количество совпадающих частей равно общему количеству частей шаблона.

Хрянятся только последние n флагов и счётчиков частей. Для этого используется циклическая очередь; проверка совпадения происходит перед сбросом.

- `std::ostream & writePatternStructure(std::ostream &os, const Pattern &pat);`

  Выводит информацию о шаблоне из структуры `pat` на поток `os`.

Сложность алгоритма

Обозначим за n суммарную длину образцов, а за m длину строки, в которой осуществляется поиск, s – количество образцов.

Построение бора занимает $O(n)$ времени, т. к. для каждого символа каждого образца нужно или выполнить переход, или добавить новое состояние и выполнить переход.

Найдём сложность построения связей неудач для всех вершин, через которые придётся пройти при добавлении образца длины l в бор. Обозначим глубины вершин, на которые указывают связи неудач вершин цепочки, как df_i. Тогда $df_{i+1} \leqslant df_i + 1$ (каждый „спуск“ по связи неудачи уменьшает глубину, поэтому $df_{i+1} = df_i + 1$ будет происходить только когда не пришлось делать ни одного спуска, т. е. $f_{i+1} = g(f_i, c_i)$, где $g(v, c)$ – переход по символу c из вершины v, а $c_i + 1$ – i-й символ образца), поэтому глубина может увеличиваться не более l раз (каждый раз на 1).

Поскольку каждый „спуск“ уменьшает глубину, а увеличиваться она может не более l раз, всего при построении связей неудач образца длины l мож-

но осуществить не более l спусков. Таким образом, при построении связей неудач всех вершин придётся осуществить не более n спусков, поэтому сложность их построения $O(n)$.

Сложность построения списков совпадений также $O(n)$, т. к. для каждой вершины нужно всего лишь установить один указатель.

Таким образом, суммарная сложность по времени построения автомата $O(n)$.

Найдём сложность поиска в строке длины m. Поскольку при спуске глубина уменьшается, при совпадении символа увеличивается на 1, а символов всего m, можно совершить не более m „подъёмов“ и не более m „спусков“.

Таким образом, сложность по времени поиска образцов будет $O(m)$, а вместе с построением автомата – $O(n + m)$.

Найдём сложность по памяти. У нас будет не более $n+1$ состояний ($O(n)$), и не более n переходов (тоже $O(n)$), за исключением связей неудач, которые учитываются вместе с самими состояниями. Если всего образцов s, то обнаруживаемых совпадений (структур `Result`) будет $O(s)$ (они не копируются).

Таким образом, сложность по памяти поиска образцов – $O(n + s)$.

Сложность поиска шаблона с джокерами получается из сложности поиска набора шаблонов. В данной реализации используется дополнительный этап предобработки и дополнительная структура, но на сложность они не влияют. Сложность по времени: $O(n+m)$, а по памяти – $O(n)$, где n – длина шаблона, а m – длина строки поиска (считаем, что $s = O(n)$).

Тестирование

- Тесты поиска набора образцов

1. Ввод:

```
CCCA
1
CC
```

Вывод:

```
1 1
2 1
```

2. Ввод:

```
ACAGAC
2
AG
CAGA
```

Вывод:

```
2 2
3 1
```

3. Ввод:

```
ACAGACAGA
2
ACAGA
ACAGA
```

Вывод:

```
1 1
1 2
5 1
5 2
```

- Тесты поиска шаблона с джокерами

1. Ввод:

```
ACTANCA
A$$A$
$
```

Вывод:

```
1
```

2. Ввод:

```
AAAAGG
AAA?G
?
```

Вывод:

```
1
2
```

3. Ввод:

```
AAGAGGACG
A^AG
?^
```

Вывод:

```
4
7
```

Если после символа джокера идёт ещё один непробельный символ, он берётся за символ „дополнительного джокера“.

4. Ввод:

```
AAAC
^CA^A
?^
```

Вывод:

```
2
```

Запуск первого теста поиска набора образцов:

Входные данные:

```
CCCA
1
CC
```

Вывод:

```
Root state 0x55983a9ddf20
Building nodes for pattern "CC"
    No transition for `C'; new state 0x55983a9ddf60
    No transition for `C'; new state 0x55983a9ddfc0
Processing forest with root 0x55983a9ddf20
Processing state 0x55983a9ddf20 with fallback 0
    Fallback for state 0x55983a9ddf60 (transition on `C'): 0x55983a9ddf20
Processing state 0x55983a9ddf60 with fallback 0x55983a9ddf20
    Fallback for state 0x55983a9ddfc0 (transition on `C'): 0x55983a9ddf60
Processing state 0x55983a9ddfc0 with fallback 0x55983a9ddf60
```

```
State machine:
State 0x55983a9ddf20:
    Transition on 'C' to 0x55983a9ddf60
    Fallback to 0
State 0x55983a9ddf60:
    Transition on 'C' to 0x55983a9ddfc0
    Fallback to 0x55983a9ddf20
State 0x55983a9ddfc0:
    Fallback to 0x55983a9ddf60
    Result #0 of length 2

Looking for matches in string "CCCA"
Looking for transition for `C'
    Found transition from 0x55983a9ddf20 to 0x55983a9ddf60
Looking for transition for `C'
    Found transition from 0x55983a9ddf60 to 0x55983a9ddfc0
Found pattern #0 starting at 1
Looking for transition for `C'
    No transition from 0x55983a9ddfc0; fallback to 0x55983a9ddf60
    Found transition from 0x55983a9ddf60 to 0x55983a9ddfc0
Found pattern #0 starting at 2
Looking for transition for `A'
    No transition from 0x55983a9ddfc0; fallback to 0x55983a9ddf60
    No transition from 0x55983a9ddf60; fallback to 0x55983a9ddf20
    No transition from root; exiting
1 1
2 1
```

Выводы

В процессе выполнения лабораторной работы был изучен алгоритм Ахо-Корасик и его применения для поиска множества образцов и для поиска одного шаблона с символами-джокерами.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: aho-corasick.cpp

```
#include <vector>
#include <string>
#include <queue>
#include <iostream>
#include <stack>
#include <algorithm>


struct Result {
    size_t pat_idx;
    size_t length;
    Result *next;
};

struct State;

struct Transition {
    char ch;
    State *dest;
};

struct State {
    std::vector<Transition> transitions {};
    State *fallback = nullptr;
    Result *results = nullptr;
    Result **results_back {&results};
};

void
addResult(State *stt, size_t pat_idx, size_t length)
{
    auto *nr = new Result {pat_idx, length, nullptr};
    *stt->results_back = nr;
    stt->results_back = &nr->next;
}

Transition *
findTransition(State *stt, char ch)
{
    for (Transition &t: stt->transitions) {
        if (t.ch == ch) {
            return &t;
        }
    }
    return nullptr;
```

```
}

void
extendForest(State *root, const char *chars, size_t len, size_t idx)
{
    State *s = root;
    for (size_t i = 0; i < len; ++i) {
        char c = chars[i];
        if (auto *t = findTransition(s, c)) {
            s = t->dest;

#ifdef ENABLE_DEBUG
            std::cerr << "\tFound transition for `" << c
                      << "' to state " << s << "\n";
#endif
        } else {
            auto *ns = new State {};
            s->transitions.push_back(Transition{c, ns});
            s = ns;

#ifdef ENABLE_DEBUG
            std::cerr << "\tNo transition for `" << c
                      << "'; new state " << s << "\n";
#endif
        }
    }

    addResult(s, idx, len);
}

State *
buildForest(std::vector<std::string> patterns)
{
    auto *root = new State {};

#ifdef ENABLE_DEBUG
    std::cerr << "Root state " << root << "\n";
#endif

    for (size_t i = 0; i < patterns.size(); ++i) {
        const std::string &pat = patterns[i];

#ifdef ENABLE_DEBUG
        std::cerr << "Building nodes for pattern \"" << pat << "\"\n";
#endif

        extendForest(root, pat.c_str(), pat.length(), i);
    }

    return root;
}
```

```
State *
findFallback(State *parent, char ch)
{
    for (State *p = parent; p != nullptr; p = p->fallback) {
        if (auto *t = findTransition(p, ch)) {
            return t->dest;
        }
    }
    return nullptr;
}

void
forestIntoStateMachine(State *root)
{
    std::queue<State *> q {{root}};

#ifdef ENABLE_DEBUG
    std::cerr << "Processing forest with root " << root << "\n";
#endif

    while (!q.empty()) {
        State *s = q.front();
        q.pop();

        State *sf = s->fallback;

#ifdef ENABLE_DEBUG
        std::cerr << "Processing state " << s
                  << " with fallback " << sf << "\n";
#endif

        for (auto &t: s->transitions) {
            State *d = t.dest;
            q.push(d);

            State *df = findFallback(sf, t.ch);

            // df == nullptr => no match on root, so no suffix matches
            // any pattern's prefix, so we'll try a null prefix -- that
            // is, fall back to root.
            d->fallback = df ? df : root;

            // 'attach' fallback's result list to ours so that we don't
            // have to inspect fallbacks all the way down on match.
            *d->results_back = d->fallback->results;

#ifdef ENABLE_DEBUG
            std::cerr << "\tFallback for state " << d
                      << " (transition on `" << t.ch << "')"
                      << ": " << d->fallback << "\n";
#endif
        }
```

```
    }
}

void
destroyStateMachine(State *root)
{
    std::queue<State *> q {{root}};

    while (!q.empty()) {
        State *s = q.front();
        q.pop();

        for (auto &t: s->transitions) {
            q.push(t.dest);
        }

        // only keep our own results
        *s->results_back = nullptr;

        // there may be duplicate patterns, so multiple own results in
        // one node
        for (Result *n, *r = s->results; r; r = n) {
            n = r->next;
            delete r;
        }

        delete s;
    }
}




bool
stepMatching(State **cs, char c)
{
#ifdef ENABLE_DEBUG
    std::cerr << "Looking for transition for `" << c << "'\n";
#endif

    for (;;) {
        if (auto *t = findTransition(*cs, c)) {
#ifdef ENABLE_DEBUG
            std::cerr << "\tFound transition from " << *cs
                      << " to " << t->dest << "\n";
#endif

            *cs = t->dest;
            return true;
        } else if ((*cs)->fallback) {
#ifdef ENABLE_DEBUG
            std::cerr << "\tNo transition from " << *cs
```

```
                    << "; fallback to " << (*cs)->fallback << "\n";
#endif

            *cs = (*cs)->fallback;
        } else {
#ifdef ENABLE_DEBUG
            std::cerr << "\tNo transition from root; exiting\n";
#endif

            return false;
        }
    }
}

struct Match {
    size_t pat_idx;
    size_t start_idx;

    bool operator<(const Match &o) const
    {
        if (start_idx == o.start_idx)
            return pat_idx < o.pat_idx;
        return start_idx < o.start_idx;
    }
};

std::vector<Match>
getMatches(State *root, const std::string &str)
{
    std::vector<Match> matches {};
    State *cs = root;

#ifdef ENABLE_DEBUG
    std::cerr << "Looking for matches in string \"" << str << "\"\n";
#endif

    for (size_t i = 0; i < str.size(); ++i) {
        if (stepMatching(&cs, str[i])) {
            for (Result *r = cs->results; r; r = r->next) {
                size_t start = i - r->length + 1;
                matches.push_back(Match{r->pat_idx, start});

#ifdef ENABLE_DEBUG
                std::cerr << "Found pattern #" << r->pat_idx
                          << " starting at " << (start + 1) << "\n";
#endif
            }
        }
    }

    return matches;
}
```

```
std::ostream &
writeStateMachine(std::ostream &os, State *root)
{
    std::stack<State *> stk {{root}};

    while (!stk.empty()) {
        State *s = stk.top();
        stk.pop();

        os << "State " << s << ":\n";
        for (auto &t: s->transitions) {
            stk.push(t.dest);
            os << "\tTransition on '" << t.ch
               << "' to " << t.dest << "\n";
        }

        os << "\tFallback to " << s->fallback << "\n";

        for (Result *r = s->results; r; r = r->next) {
            os << "\tResult #" << r->pat_idx
               << " of length " << r->length << "\n";
        }
    }

    return os;
}

int
main()
{
    std::string search_s;
    std::getline(std::cin, search_s);

    int n_patterns;
    std::cin >> n_patterns >> std::ws;

    std::vector<std::string> patterns;
    patterns.resize(n_patterns);
    for (int i = 0; i < n_patterns; i++) {
        std::getline(std::cin, patterns[i]);
    }

    State *root = buildForest(patterns);
    forestIntoStateMachine(root);

#ifdef ENABLE_DEBUG
    writeStateMachine(std::cerr << "\nState machine:\n", root) << "\n";
#endif
```

```
    auto matches = getMatches(root, search_s);

    std::sort(matches.begin(), matches.end());
    for (auto &match: matches) {
        std::cout << (match.start_idx + 1) << " "
                  << (match.pat_idx + 1) << "\n";
    }

    destroyStateMachine(root);

    return 0;
}
```

Название файла: wildcard.cpp

```
#include <vector>
#include <string>
#include <queue>
#include <iostream>
#include <stack>
#include <algorithm>


struct Result {
    size_t pat_idx;
    size_t length;
    Result *next;
};

struct State;

struct Transition {
    char ch;
    State *dest;
};

struct State {
    std::vector<Transition> transitions {};
    State *fallback = nullptr;
    Result *results = nullptr;
    Result **results_back {&results};
};

void
addResult(State *stt, size_t pat_idx, size_t length)
{
    auto *nr = new Result {pat_idx, length, nullptr};
    *stt->results_back = nr;
    stt->results_back = &nr->next;
}
```

```
Transition *
findTransition(State *stt, char ch)
{
    for (Transition &t: stt->transitions) {
        if (t.ch == ch) {
            return &t;
        }
    }
    return nullptr;
}

void
extendForest(State *root, const char *chars, size_t len, size_t idx)
{
    State *s = root;
    for (size_t i = 0; i < len; ++i) {
        char c = chars[i];
        if (auto *t = findTransition(s, c)) {
            s = t->dest;

#ifdef ENABLE_DEBUG
            std::cerr << "\tFound transition for `" << c
                      << "' to state " << s << "\n";
#endif
        } else {
            auto *ns = new State {};
            s->transitions.push_back(Transition{c, ns});
            s = ns;

#ifdef ENABLE_DEBUG
            std::cerr << "\tNo transition for `" << c
                      << "'; new state " << s << "\n";
#endif
        }
    }

    addResult(s, idx, len);
}

State *
findFallback(State *parent, char ch)
{
    for (State *p = parent; p != nullptr; p = p->fallback) {
        if (auto *t = findTransition(p, ch)) {
            return t->dest;
        }
    }
    return nullptr;
}

void
forestIntoStateMachine(State *root)
```

```
{
    std::queue<State *> q {{root}};

#ifdef ENABLE_DEBUG
    std::cerr << "Processing forest with root " << root << "\n";
#endif

    while (!q.empty()) {
        State *s = q.front();
        q.pop();

        State *sf = s->fallback;

#ifdef ENABLE_DEBUG
        std::cerr << "Processing state " << s
                  << " with fallback " << sf << "\n";
#endif

        for (auto &t: s->transitions) {
            State *d = t.dest;
            q.push(d);

            State *df = findFallback(sf, t.ch);

            // df == nullptr => no match on root, so no suffix matches
            // any pattern's prefix, so we'll try a null prefix -- that
            // is, fall back to root.
            d->fallback = df ? df : root;

            // 'attach' fallback's result list to ours so that we don't
            // have to inspect fallbacks all the way down on match.
            *d->results_back = d->fallback->results;

#ifdef ENABLE_DEBUG
            std::cerr << "\tFallback for state " << d
                      << " (transition on `" << t.ch << "')"
                      << ": " << d->fallback << "\n";
#endif
        }
    }
}

void
destroyStateMachine(State *root)
{
    std::queue<State *> q {{root}};

    while (!q.empty()) {
        State *s = q.front();
        q.pop();

        for (auto &t: s->transitions) {
```

```
            q.push(t.dest);
        }

        // only keep our own results
        *s->results_back = nullptr;

        // there may be duplicate patterns, so multiple own results in
        // one node
        for (Result *n, *r = s->results; r; r = n) {
            n = r->next;
            delete r;
        }

        delete s;
    }
}
```

```
struct PartInfo {
    size_t offset;
    const char *chars;
    size_t length;
};

struct Complement {
    size_t index;
    char ch;
};

struct Pattern {
    std::vector<PartInfo> parts;
    std::vector<Complement> complement;
    size_t length;
};

Pattern
createPatternStructure(const std::string &pat,
                       char wildcard, char complement)
{
    std::vector<PartInfo> parts;
    std::vector<Complement> compls;

    size_t extra_offset = 0;

    const char
        *hd = pat.c_str(),
        *pc = hd;
    while (pc < hd + pat.size()) {
        const char *end = pc;
        for (char c = *end;; c = *++end) {
```

```
            if (c == 0
                || c == wildcard
                || c == complement) {
                break;
            }
        }

        if (end != pc) {
            size_t
                offset = pc - hd - extra_offset,
                length = end - pc;
            parts.push_back(PartInfo{offset, pc, length});

#ifdef ENABLE_DEBUG
            std::cerr << "Part at offset " << offset
                      << " of length " << length
                      << ": \"";
            std::cerr.write(pc, length) << "\"\n";
#endif
        }

        pc = end;
        if (complement
            && *pc == complement) {
            size_t offset = pc - hd - extra_offset;
            char ch = pc[1];
            compls.push_back(Complement{offset, ch});

#ifdef ENABLE_DEBUG
            std::cerr << "Complement to char `" << ch
                      << "' at offset " << offset << "\n";
#endif

            pc += 2;
            ++extra_offset;
        }
        for (; *pc == wildcard; ++pc);
    }

    size_t real_size = pat.size() - extra_offset;
    return Pattern{std::move(parts), std::move(compls), real_size};
}

State *
buildForestFromPattern(const Pattern &pat)
{
    auto *root = new State {};

#ifdef ENABLE_DEBUG
    std::cerr << "Root state " << root << "\n";
#endif
```

```
    size_t n = pat.parts.size();
    for (size_t i = 0; i < n; ++i) {
        const auto &part = pat.parts[i];

#ifdef ENABLE_DEBUG
        std::cerr << "Building nodes for part \"";
        std::cerr.write(part.chars, part.length) << "\" at offset "
                                                 << part.offset << "\n";
#endif

        extendForest(root, part.chars, part.length, i);
    }

    for (size_t i = 0; i < pat.complement.size(); ++i) {
        const Complement &c = pat.complement[i];

#ifdef ENABLE_DEBUG
        std::cerr << "Building nodes for complement `" << c.ch
                  << "' at offset " << c.index << "\n";
#endif

        extendForest(root, &c.ch, 1, n + i);
    }

    return root;
}




bool
stepMatching(State **cs, char c)
{
#ifdef ENABLE_DEBUG
    std::cerr << "Looking for transition for `" << c << "'\n";
#endif

    for (;;) {
        if (auto *t = findTransition(*cs, c)) {
#ifdef ENABLE_DEBUG
            std::cerr << "\tFound transition from " << *cs
                      << " to " << t->dest << "\n";
#endif

            *cs = t->dest;
            return true;
        } else if ((*cs)->fallback) {
#ifdef ENABLE_DEBUG
            std::cerr << "\tNo transition from " << *cs
                      << "; fallback to " << (*cs)->fallback << "\n";
#endif
```

```
            *cs = (*cs)->fallback;
        } else {
#ifdef ENABLE_DEBUG
            std::cerr << "\tNo transition from root; exiting\n";
#endif

            return false;
        }
    }
}

std::vector<size_t>
getTotalMatches(State *root,
                const Pattern &pat,
                const std::string &str)
{
    size_t n_parts = pat.parts.size();
    size_t str_len = str.size();

    std::vector<size_t> partial_matches (pat.length);
    std::vector<bool> disabled (pat.length);
    size_t match_shift = 0;

    std::vector<size_t> matches {};
    State *cs = root;

    for (size_t i = 0; i < str_len; ++i) {
        partial_matches[match_shift] = 0;
        disabled[match_shift] = false;

        if (stepMatching(&cs, str[i])) {
            for (Result *r = cs->results; r; r = r->next) {
                size_t total_off;
                bool complement = r->pat_idx >= n_parts;
                if (complement) {
                    size_t c_idx = r->pat_idx - n_parts;
                    total_off = pat.complement[c_idx].index;
                } else {
                    const PartInfo &part = pat.parts[r->pat_idx];
                    total_off = part.offset + r->length - 1;
                }

                if (i < total_off) {
                    continue;
                }

                size_t start = i - total_off;
                if (start + pat.length > str_len) {
                    continue;
                }

#ifdef ENABLE_DEBUG
```

```
                std::cerr << "Part #" << r->pat_idx
                          << " matches at " << (i - r->length + 1)
                          << " (pattern starts at " << start  << ")\n";
#endif

                // -pat.length < match_shift - total_off < pat.length
                size_t idx =
                    (pat.length + match_shift - total_off) % pat.length;

                if (complement) {
                    disabled[idx] = true;

#ifdef ENABLE_DEBUG
                    std::cerr << "Complement found; disabling match at "
                              << start << "\n";
#endif
                } else if (!disabled[idx]) {
                    ++partial_matches[idx];

#ifdef ENABLE_DEBUG
                    std::cerr << partial_matches[idx]
                              << "/" << n_parts
                              << " parts matched at offset "
                              << start << "\n";
#endif
                }
            }
        }

        if (++match_shift == pat.length) {
            match_shift = 0;
        }

        if ((i + 1) >= pat.length
            && !disabled[match_shift]
            && partial_matches[match_shift] == n_parts) {
            size_t start = i + 1 - pat.length;

#ifdef ENABLE_DEBUG
            std::cerr << "Pattern matched at " << start << "\n";
#endif

            matches.push_back(start);
        }
    }

    return matches;
}
```

```
std::ostream &
writeStateMachine(std::ostream &os, State *root)
{
    std::stack<State *> stk {{root}};

    while (!stk.empty()) {
        State *s = stk.top();
        stk.pop();

        os << "State " << s << ":\n";
        for (auto &t: s->transitions) {
            stk.push(t.dest);
            os << "\tTransition on '" << t.ch
               << "' to " << t.dest << "\n";
        }

        os << "\tFallback to " << s->fallback << "\n";

        for (Result *r = s->results; r; r = r->next) {
            os << "\tResult #" << r->pat_idx
               << " of length " << r->length << "\n";
        }
    }

    return os;
}

std::ostream &
writePatternStructure(std::ostream &os, const Pattern &pat)
{
    for (const auto &part: pat.parts) {
        os << "Part at offset " << part.offset
           << " of length " << part.length
           << ": \"";
        os.write(part.chars, part.length) << "\"\n";
    }

    for (const auto &comp: pat.complement) {
        os << "Complement to char `" << comp.ch
           << "' at index " << comp.index << "\n";
    }

    os << "Total length of pattern: " << pat.length << "\n";

    return os;
}

int
main()
{
    std::string search_s;
    std::getline(std::cin, search_s);
```

```
    std::string pattern;
    std::getline(std::cin, pattern);

    char wildcard;
    std::cin >> wildcard;

    char complement;
    if (!(std::cin >> complement)) {
        complement = 0;
    }

    Pattern pat = createPatternStructure(pattern, wildcard, complement);

#ifdef ENABLE_DEBUG
    writePatternStructure(std::cerr << "\nPattern:\n", pat) << "\n";
#endif

    State *root = buildForestFromPattern(pat);
    forestIntoStateMachine(root);

#ifdef ENABLE_DEBUG
    writeStateMachine(std::cerr << "\nState machine:\n", root) << "\n";
#endif

    auto matches = getTotalMatches(root, pat, search_s);

    for (int m: matches) {
        std::cout << (m + 1) << "\n";
    }

    destroyStateMachine(root);

    return 0;
}
```