String matching algorithms

Naive, Knuth-Morris-Pratt, Rabin-Karp

Типы алгоритмов

Алгоритмы поиска подстроки делятся на 2 типа алгоритмов/задач:

1. Точные алгоритмы поиска подстроки (Exact String Matching Algorithms)
2. Поиск похожих строк / Нечеткий поиск (Approximate String Matching Algorithms)

Поговорим о точных алгоритмах

Exact String Matching Algorithms

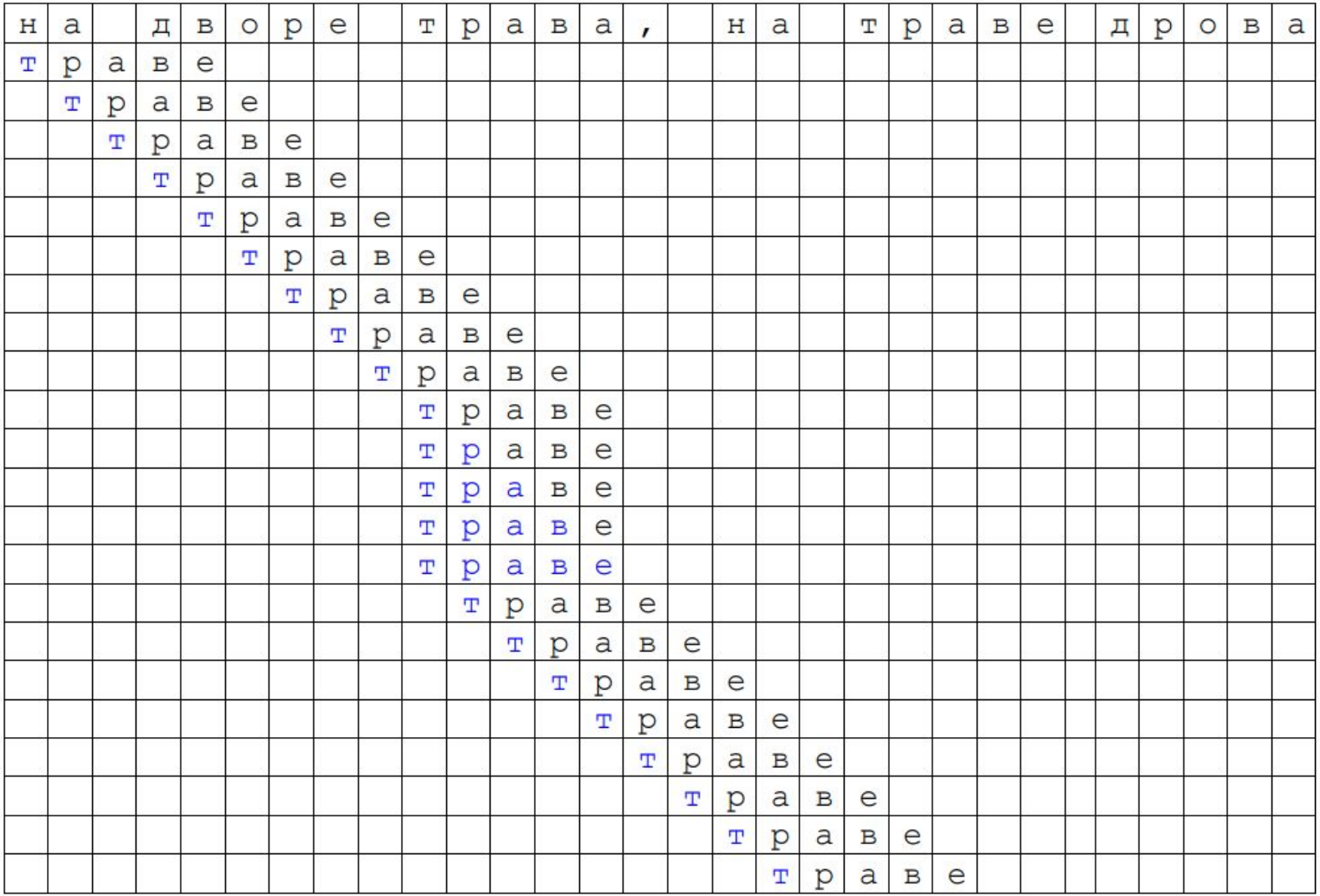
Задача: найти одно, несколько, или все совпадения определенной строки (шаблона) pat[m] в длинной строке (текст и др.) txt[n] (n>m) так, что совпадение будет точным.

Алфавит шаблона должен совпадать с алфавитом строк.

Naive algorithm

Идея: “накладываем” шаблон на текст и проверяем символы один за другим. Если первый символ шаблона и текста совпадут, то проверяем второй символ и т.д. Если не совпадут, сдвигаем шаблон на 1.

```python
def NaiveSearch(txt, pat):
    n = len(txt)
    m = len(pat)
    for i in range(n-m+1):
        j = 0
        while j < m:
            if txt[i+j] != pat[j]:
                break
            j += 1
        if j == m:
            return i
    return -1
```


на дворе трава, на траве дрова
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава
трава

																	т	р	а	в	е							
																	т	р	а	в	е							
																	т	р	а	в	е							
																	т	р	а	в	е							
																	т	р	а	в	е							

Naive algorithm

Лучший случай: когда первый символ шаблона не присутствует в тексте

txt[] = "AABCCAADDEE";

pat[] = "FAA";

Время работы алгоритма в лучшем случае: O(n).

Naive algorithm

Худший случай: 2 варианта:

1. Когда все символы в тексте и в шаблоне одинаковые

txt[] = "AAAAAAAAAAAAAAAAAA";

pat[] = "AAAAA";

2. Когда отличается только последний символ

txt[] = "AAAAAAAAAAAAAAAAAB";

pat[] = "AAAAB";

Время работы алгоритма в худшем случае: O(m*(n-m+1))

Naive algorithm

Однако если m достаточно мало по сравнению с n, то тогда асимптотика получается близкой к O(n), поэтому этот алгоритм достаточно широко применяется на практике.

Naive algorithm

Преимущества

- Требует O(1) памяти.
- Приемлемое время работы на практике. Благодаря этому алгоритм применяется, например, в браузерах и текстовых редакторах (при использовании Ctrl + F), потому что обычно паттерн, который нужно найти, очень короткий по сравнению с самим текстом. Также примитивный алгоритм используется в стандартных библиотеках языков высокого уровня (C++, Java), потому что он не требует дополнительной памяти.
- Простая и понятная реализация.

Недостатки

- Требует $O(m \cdot (n-m+1))$ операций, вследствие чего алгоритм работает медленно в случае, когда длина паттерна достаточно велика

KMP (Knuth-Morris-Pratt algorithm)

Время работы алгоритма в худшем случае: O(n).

Основная идея алгоритма KMP заключается в следующем: всякий раз, когда мы обнаруживаем несоответствие (после нескольких совпадений), мы уже знаем некоторые символы в тексте следующего окна. Мы используем эту информацию, чтобы избежать сопоставления символов, которые, как мы знаем, в любом случае будут совпадать. Давайте рассмотрим пример ниже, чтобы понять это

KMP (Knuth-Morris-Pratt algorithm)

```
txt = "AAAAABAAABA"
```

```
pat = "AAAA"
```

Мы сравниваем первое окно **txt** с **pat**

```
txt = "AAAAABAAABA"
```

```
pat = "AAAA"
```

Нашли совпадение. Здесь всё аналогично Naive string matching

KMP (Knuth-Morris-Pratt algorithm)

На следующем шаге мы сравниваем следующее окно **txt** с **pat**.

```
txt = "AAAAABAAABA"

pat =  "AAAA" [pat сдвинулся на одну позицию]
```

Здесь KMP оптимизирует примитивный алгоритм

Во втором окне мы сравниваем только четвёртую A шаблона с четвёртым символом окна текста, чтобы решить, совпадает окно или нет.

Так как мы знаем, что первые три символа совпадут, мы пропустили их сравнивание

Префикс-функция

Прежде чем перейти к описанию алгоритма, необходимо рассмотреть понятие префикс-функции.

Дана строка s[0..n-1]. Требуется вычислить для неё префикс-функцию, т.е. массив чисел π [0..n-1], где π[i] вычисляется следующим образом: это такая наибольшая длина собственного суффикса (окончания) подстроки s[0..i], совпадающего с её префиксом (начало подстроки). В частности, значение π[0] полагается равным 0.

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max_{k=0\ldots i} \{\ k\ :\ s[0\ldots k-1] = s[i-k+1\ldots i]\ \}.$$

Например, для строки "abcabcd" префикс-функция равна: [0, 0, 0, 1, 2, 3, 0]

- у строки "a" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "ab" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abc" нет нетривиального префикса, совпадающего с суффиксом;
- у строки "abca" префикс длины 1 совпадает с суффиксом;
- у строки "abcab" префикс длины 2 совпадает с суффиксом;
- у строки "abcabc" префикс длины 3 совпадает с суффиксом;
- у строки "abcabcd" нет нетривиального префикса, совпадающего с суффиксом

Другой пример — для строки "aabaaab" она равна [0, 1, 0, 1, 2, 2, 3]

```python
def prefix(s):
    v = [0]*len(s)
    for i in range(1,len(s)):
        k = v[i-1]
        while k > 0 and s[k] != s[i]:
            k = v[k-1]
        if s[k] == s[i]:
            k = k + 1
        v[i] = k
    return v

def kmp(txt, pat):
    index = -1
    f = prefix(pat)
    print('prefix:', f)
    k = 0
    for i in range(len(txt)):
        while k > 0 and pat[k] != txt[i]:
            k = f[k-1]
        if pat[k] == txt[i]:
            k = k + 1
        if k == len(pat):
            index = i - len(pat) + 1
            break
    return index
```

КМР (Knuth-Morris-Pratt algorithm)

Особенности КМП-поиска:

1. требуется порядка O(m+n) сравнений символов для получения результата;

2. схема КМП-поиска дает подлинный выигрыш только тогда, когда неудаче предшествовало некоторое число совпадений. Лишь в этом случае образ сдвигается более чем на единицу. К несчастью совпадения встречаются значительно реже чем несовпадения. Поэтому выигрыш от КМП-поиска в большинстве случаев текстов весьма незначителен.

3. Затраты памяти O(m)

Rabin-Karp algorithm

Алгоритм Рабина — Карпа — это алгоритм поиска подстроки с использованием хеширования.

Хеш-функция

Хеш-функция (англ. *hash function* от *hash* — «превращать в фарш», «мешанина»), или **функция свёртки** — функция, осуществляющая преобразование массива входных данных произвольной длины в выходную битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется **хешированием**.

Rabin-Karp algorithm

Как и примитивный алгоритм, алгоритм Рабина-Карпа также сдвигает шаблон на одно значение. Но в отличие от примитивного алгоритма, алгоритм Рабина Карпа сопоставляет хеш-значение шаблона с хеш-значением текущей подстроки текста, и если хеш-значения совпадают, то только тогда он начинает сопоставлять отдельные символы.

Rabin-Karp algorithm

Таким образом, алгоритм Рабина Карпа должен вычислять хеш-значения для следующих строк:

1) Сам паттерн

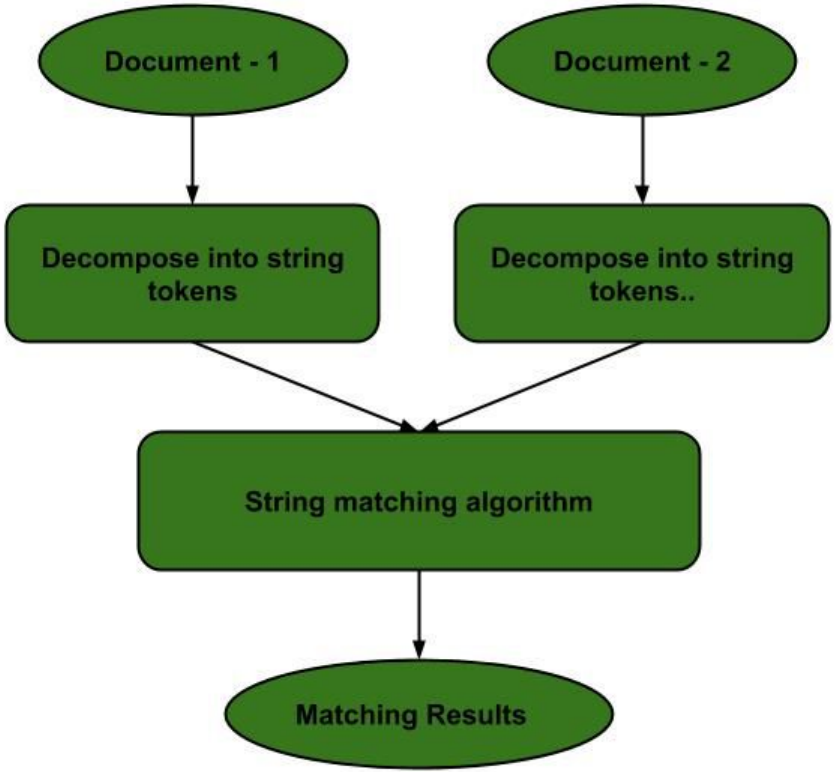
2) Все подстроки текста длины m.

Rabin-Karp algorithm

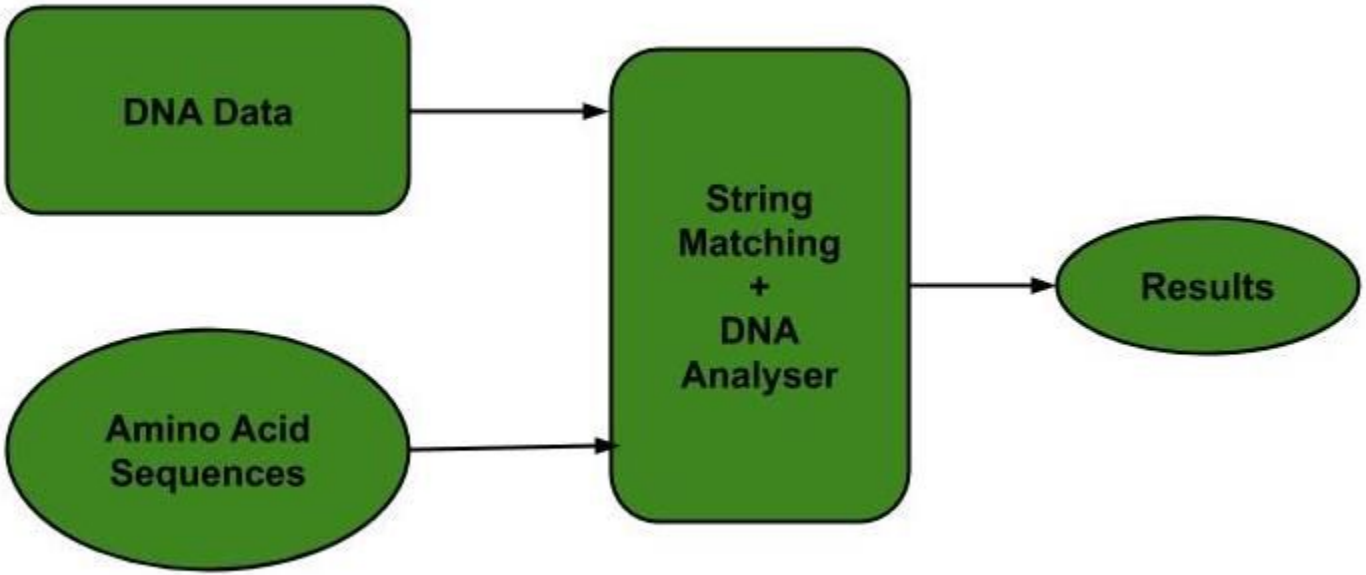
Алгоритм редко используется для поиска одиночного шаблона, но имеет значительную теоретическую важность и очень эффективен в поиске совпадений множественных шаблонов одинаковой длины. Для текста длины n и шаблона длины m его среднее и лучшее время исполнения равно O(n) при правильном выборе хеш-функции, но в худшем случае он имеет эффективность O(nm), что является одной из причин того, почему он не слишком широко используется.

```python
def RabinKarp(string, pattern):
    n, m = len(string), len(pattern)
    hpattern = hash(pattern);
    for i in range(n-m+1):
        hs = hash(string[i:i+m])
        if hs == hpattern:
            if string[i:i+m] == pattern:
                return i
    return -1
```

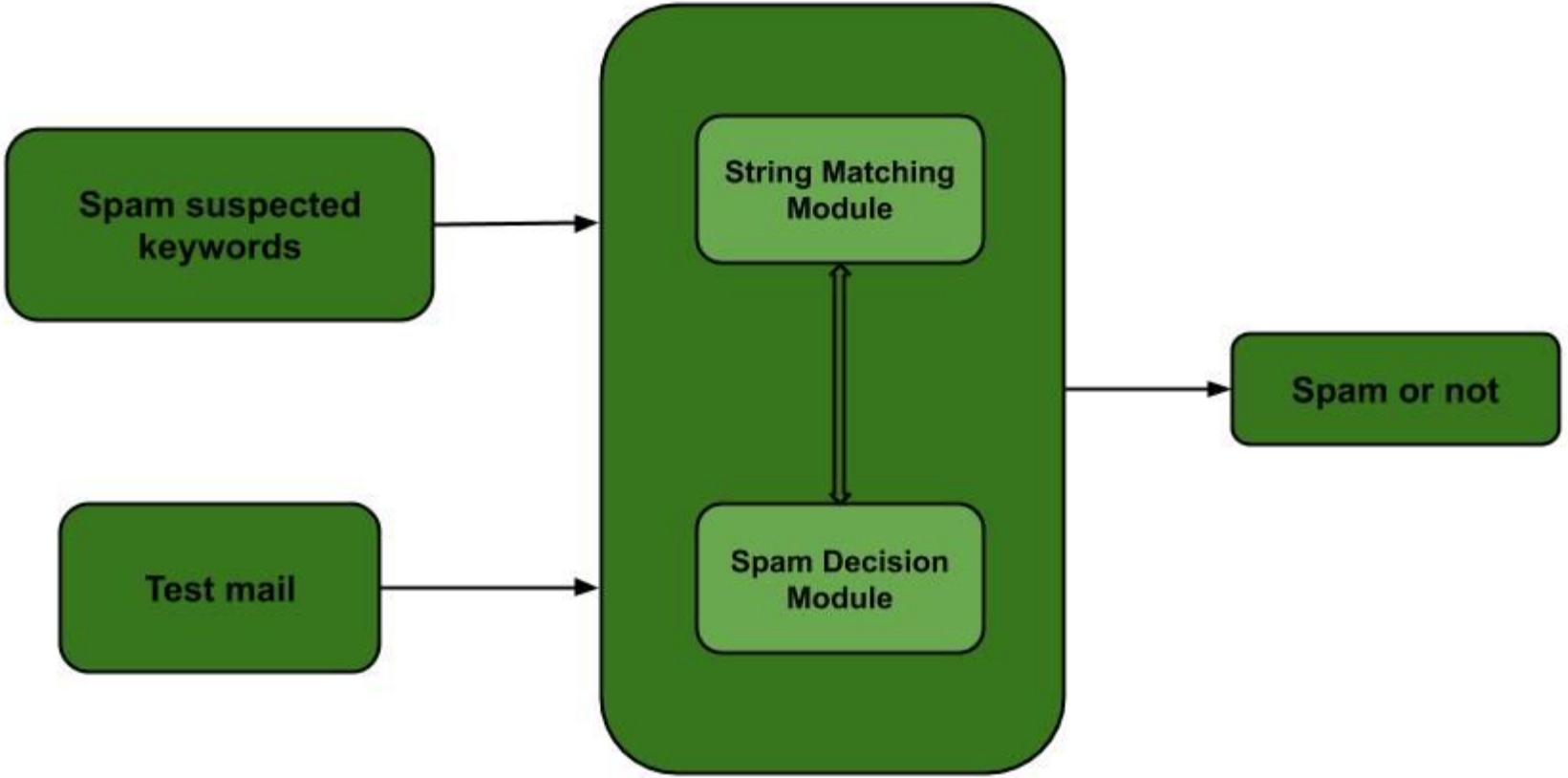
Прим ки



Антипла
сравнива эти
алгоритм того,
является


DNA Data
Amino Acid Sequences
String Matching + DNA Analyser
Results

- **Цифровая криминалистика:** Алгоритмы поиска подстроки используются для поиска конкретных текстовых строк, представляющих интерес в цифровом криминалистическом тексте, которые нужны в расследовании.
- **Проверка орфографии**


Spam suspected keywords
Test mail
String Matching Module
Spam Decision Module
Spam or not

Поисковые системы или поиск контента в больших базах данных: для эффективной категоризации и организации данных используются алгоритмы поиска подстроки. Классификация осуществляется на основе ключевых слов поиска. Таким образом, алгоритмы сопоставления строк облегчают поиск информации

Система обнаружения вторжений: пакеты данных, содержащие ключевые слова, связанные со вторжением, обнаруживаются с помощью алгоритмов поиска подстроки. Весь вредоносный код хранится в базе данных, и все поступающие данные сравниваются с сохраненными данными. Если совпадение найдено, генерируется сигнал тревоги.

Выбор алгоритма

На сегодняшний день существует огромное разнообразие алгоритмов поиска подстроки. Программисту приходится выбирать подходящий в зависимости от таких факторов.

1. Нужна ли вообще оптимизация, или хватает примитивного алгоритма? Как правило, именно его реализуют стандартные библиотеки языков программирования.
2. «Враждебность» пользователя. Другими словами: будет ли пользователь намеренно задавать данные, на которых алгоритм будет медленно работать?
3. Грамматика языка может быть недружественной к тем или иным эвристикам, которые ускоряют поиск «в среднем».
4. Размер алфавита. Многие алгоритмы (особенно основанные на сравнении с конца) имеют эвристики, связанные с несовпавшим символом. На больших алфавитах таблица символов будет занимать много памяти, на малых — соответствующая эвристика будет неэффективной.
5. Возможность проиндексировать *строку*. Если таковая есть, поиск серьёзно ускорится.
6. Требуется ли одновременный поиск нескольких строк? Приблизительный поиск? Побочные свойства некоторых алгоритмов (Ахо-Корасик, двоичного алгоритма) позволяют такое.