Федеральное государственное автономное учреждение
высшего профессионального образования

Московский Физико-Технический Институт
КЛУБ ТЕХА ЛЕКЦИЙ

Алгоритмы и структуры данных

III СЕМЕСТР
Физтех Школа: *ФПМИ*
Направление: *ПМИ/КТ*
Лектор: *Мацкевич Степан Евгеньевич*

Автор: *Евсюков Никита*
Проект на overleaf
Проект на GitHub

2020 года

Содержание

1 Лекция 1: Поиск строк

Важно!

Из-за отсутствия записи лекции и презентации с неё она была затехана, исходя из программы прошлого года (т.к. новой программы пока нет). Был использован Кормен, emaxx и ещё некоторые источники. Возможно, этот материал будет дополнен или изменён со временем.

1.1 Основные понятия

Рассмотрим задачу поиска подстрок. Формализуем её следующим образом.

Утверждение. Пусть текст задан в виде массива $T[1 \dots n]$, а образец (шаблон) — в виде массива $P[1 \dots m]$, где $m \leq n$. Причём элементы массивов — символы из конечного алфавита.

Определение 1.1. Символьные массивы P и T называют строками символов.

Пусть Σ — конечный алфавит, а Σ^* — множество всех строк конечной длины, образованных с помощью этого алфавита, тогда введём некоторые формальные определения.

Определение 1.2. ω — префикс строки x (обозначается как $\omega \sqsubset x$), если $x = \omega y$ для некоторого $y \in \Sigma^*$

Определение 1.3. ω — суффикс строки x (обозначается как $\omega \sqsupset x$), если $x = y\omega$ для некоторого $y \in \Sigma^*$

Утверждение. Префикс (суффикс) может быть собственным, это означает, что он не совпадает с самой строкой.

1.2 Постановки задачи поиска. Тривиальный алгоритм поиска подстроки в строке.

Для начала дадим несколько вспомогательных определений.

Определение 1.4. P встречается в тексте со сдвигом s, если $0 \leq s \leq n-m$ и $T[s+j] = P[j]$ для $1 \leq j \leq m$

Определение 1.5. s — допустимый сдвиг, если P встречается в T со сдвигом s.

Утверждение. Задача поиска подстроки таким образом представляет собой задачу поиска всех допустимых сдвигов

Рассмотрим тривиалный алгоритм поиска подстроки в строке

```
n = T.size();
m = P.size();
for (size_t s = 0; s < n - m; ++s) {
  if (T.substr(s + 1, s + m) == P) {
    std::cout << s << std::endl;
  }
}
```

Утверждение. Очевидно, что время его работы равно $O((n-m+1)m)$, что в худшем случае ($m = \frac{n}{2}$) равно $O(n^2)$.

1.3 Префикс-функция. Тривиальный алгоритм нахождения.

Определение 1.6. Префикс-функция — массив чисел $\pi[0 \dots n-1]$, где $\pi[i]$ — наибольшая длина наибольшего собственного суффикса подстроки $s[0 \dots i]$, совпадающего с её префиксом. Или же

$$\pi[i] = \max\{k \colon (k < i) \wedge (s[0 \dots k-1] = s[i-k+1 \dots i]\}$$

Например, у строки abca префикс длины 1 совпадает с суффиксом, т.е. $\pi[4] = 1$

Исходя из определения можно написать простейший алгоритм

```
std::vector<int32_t> prefix(std::string& s) {
  std::vector<int32_t> pi(s.length());

  for (size_t i = 0; i < s.length(); ++i) {
    for (size_t k = 0; k <= i; ++k) {
      if (s.substr(0, k) == s.substr(i - k + 1, k)) {
        pi[i[ = k;
      }
    }
  }
}
```

Утверждение. Очевидно, что его асимптотика $O(n^3)$.

1.4 Линейный алгоритм нахождения. Доказательство времени работы.

Оптимизируем наш простейший алгоритм. Для начала заметим, что префикс функция увеличивается не более, чем на единицу, т.е. $\pi[i+1] - \pi[i] \leq 1$. Т.е. могло произойти не более n увеличений функции (каждый раз увеличение не более чем на 1) и, соответственно, не более n уменьшений.

Утверждение. Алгоритм может иметь асимптотику $O(n^2)$

Действительно, достаточно заметить, что нам нужно произвести $O(n)$ сравнений строк (сравнение необходимо только при увеличении/уменьшении префикс-функции).

Далее нужно как-то избавиться от тяжёлой операции сравнения подстрок. **Но как это сделать?** Будем использовать то, что мы уже посчитали.

Утверждение. Если $s[i+1] = s[\pi[i]]$, то $\pi[i+1] = \pi[i] + 1$

Доказательство легко видно на рисунке

$$\underbrace{\overbrace{s_1 s_2}^{\pi[i]} \overbrace{s_3}^{s_3=s_{i+1}}}_{\pi[i+1]} \dots \overbrace{s_{i-1} s_i}^{\pi[i]} s_{i+1}$$

А что делать, если $s[i+1] \neq s[\pi[i]]$? Тогда попробуем рассмотреть суффикс поменьше длиной k и проверить, существует ли равный ему префикс, в таком случае, если мы найдем максимальное такое k, то нам останется проверить равенство $s[i+1] = s[k]$.

Покажем на примере

$$\overbrace{\underbrace{s_0 s_1}_{k} s_2 s_3}^{\pi[i]} \dots \overbrace{s_{i-3} s_{i-2} \underbrace{s_{i-1} s_i}_{k}}^{\pi[i]}$$

Утверждение. $k = \pi[\pi[i] - 1]$ (вычитание единицы из-за нумерации строк с 0)

Это легко вытекает из предыдущего рисунка, действительно, если мы рассмотрим суффикс длины $\pi[i]$ и найдём в нём ещё один суффикс, отвечающий нашему условию, то мы получим требуемое.

Таким образом, получим итоговый алгоритм:

1. Считаем $\pi[i]$ от $i = 1$ до $i = n - 1$
2. Тестируем образец длины j по описанной выше схеме
3. Останавливаем перебор при $j = 0$

```cpp
std::vector<int32_t> prefix(std::string &s) {
  std::vector<int32_t> pi(s.length());

  for (size_t i = 1; i < s.length(); ++i) {
    size_t j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) {
      j = pi[j - 1];
    }
    if (s[i] == s[j]) ++j;
    pi[i] = j;
  }
  return pi;
}
```

Утверждение. Представленный алгоритм работает за $O(n)$

1.5 Подсчёт префикс-функции для строки q$t. Алгоритм Кнута-Морриса-Пратта

Пусть $|q| = n, |t| = m$. Рассмотрим значение префикс-функции в таком случае.

Утверждение. Значения $\pi[i]$ при $i > n$ равны 0, а равенство $\pi[i] = n$ означает окончание вхождения искомого паттерна.

Таким образом получаем алгоритм Кнута-Морриса-Пратта. Т.к. значение префикс-функции не может превысить n мы можем хранить только искомую строку (следует из самого алгоритма описанного выше). Таким образом получаем требуемую асимптотику.

Утверждение. Алгоритм Кнута-Морриса-Пратта работает за $O(n + m)$ и $O(n)$ памяти.

1.6 Z-функция. Тривиальный алгоритм нахождения.

Определение 1.7. Z-функция строки s — массив длины n ($|s| = n$), где $z[i]$ — наибольший общий префикс строки s и её i-ого суффикса.

Рассмотрим тривиальный алгоритм, который перебирает ответ для каждого i

```
std::vector<int32_t> zFunction(std::string& s) {
  std::vector<int32_t> z(s.length());

  for (size_t i = 1; i < s.length(); ++i) {
    while ((i + z[i] < n) && (s[z[i]] == s[i + z[i]])) {
      ++z[i];
    }
  }
  return z;
}
```

Утверждение. Асимптотика такого алгоритма, очевидно, $O(n^2)$

1.7 Линейный поиск Z-функции. Доказательство времени работы.

Для оптимизации алгоритма воспользуемся тем же, т.е. будет использовать вычисленные значения.

Определение 1.8. Отрезок совпадения — подстрока, совпадающая с префиксом строки s

Например, для строки abab, ab — отрезок совпадения.

Будем вычислять значения z-функции по очереди от $i = 1$ до $n-1$ и хранить значения $[l; r]$ самого правого отрезка совпадения, т.е. r указывает нам на правую границу, до которой просканировал алглоритм.

Далее на некотором i-ом шаге возможно два случая:

- $i > r$, тогда нам ничего не известно про следующие символы, т.к. они не были просканированы алгоритмом, так что запустим тривиальный алгоритм от этого i, после чего (если $z[i] > 0$) обновим значения $[l; r]$
- $i \leq r$, тогда мы можем инициализировать значение $z[i]$ чем-то большим 0. Рассмотрим на примере.

Утверждение. В случае $i \le r$ можно проинициализировать $z[i]$ таким образом, а далее аналогичным образом запустить тривиальный алгоритм поиска.

$$z[i] = \min\left(r - i + 1, z[i - l]\right)$$

Доказательство этого утверждения увидим на рисунке

$$a_1 a_2 a_3 \underbrace{\overbrace{a_4}^{i-l} a_5}_{z[i-l]} \dots a_l a_{l+1} a_{l+2} \underbrace{a_i a_r}_{z[i-l]}$$

Т.к. мы знаем, что $[l; r]$ — отрезок совпадения, а $z[i - l]$ мы уже посчитали, то для начального инициализации можно будет использовать посчитанное значение. Ограничение в $r - i + 1$ нужно для того, чтобы повторяющийся кусок не вышел за границы r, т.к. мы ничего не знаем о символах после r.

Приведём реализацию:

```
std::vector<int32_t> zFunction (std::string& s) {
  int32_t l = 0, r = 0;
  std::vector<int32_t> z(s.length());

  for (int32_t i = 0; i < s.length(); ++i) {
    if (i <= r) {
      z[i] = std::min(r - i + 1, z[i - l]);
    }

    while ((i + z[i] < n) && (s[z[i]] == s[i + z[i]])) {
      ++z[i];
    }

    if (i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  return z;
}
```

Утверждение. Асимптотика данного алгоритма $O(n)$

Доказательство. Достаточно рассмотреть цикл *while*, т.к. остальные операции выполняются за константу. Заметим что при $i > r$ каждая итерация цикла продвигает r вправо (кроме случаев, когда $s[0] \neq s[i]$, тогда вовсе не будет итераций цикла *while*. Если же $i \le r$, итерация цикла либо продвинет r, либо не случится вовсе. В таком случае, итераций цикла *while* будет не более n штук. ∎

1.8 Применение для поиска подстроки в строке.

Применение и оптимизация памяти аналогично префикс-функции.