Алгоритмы и структуры данных

Лекция 25
Строки. Z-функция. Префикс-функция. Бор. Алгоритм Ахо-Корасик.

Сергей Леонидович Бабичев

Абстракция *строка символов*.

Строка символов

Работа со строками символов — необходимая часть интерфейса программы с пользователем.
Строки нужны для:

- вывода информации на экран
- именования внешних объектов — файлов, компьютеров, сетевых ресурсов

Главная проблема — динамически изменяемый размер.
Как считать строку неизвестной заранее длины?

- не разрешать пользователям работать с длинными строками?
- зарезервировать под неё место определённого размера?
- вводить посимвольно и расширять строку?

Строка символов

Классические представления:

1. Строка в языке Паскаль
   - появилась в ObjectPascal
   - $+$ удобные операции +, =, []
   - $+$ быстрая операция `length`
   - $-$ ограниченная длина
   - $-$ отсутствие контроля по умолчанию
2. Строка в языке Си
   - $+$ удобная операция []
   - $+$ неограниченная длина
   - $+$ простое представление
   - $-$ операция `length` $O(N)$
   - $-$ низкоуровневое программирование с указателями
   - $-$ очень низкая надёжность

Строка символов

Что нам хотелось бы от строк:

- отсутствие ограничений на размер и содержание
- удобные способы ввода и вывода
- интерфейс с операционной системой. Если требуется имя файла, то строка должна его предоставить в требуемом виде
- операции сложения строк с другими строками и с одиночными символами
- определение размера строки
- выделение подстроки

Абстракция *строка символов*

Интерфейс абстракции *строка символов*

- = — присвоение строки другой
- += — добавить символ или строку в конец
- [] — получение символа из строки/присвоение элементу строки
- `size` — получить размер
- `substr` — вырезать подстроку
- `c_str` — получить представление строки для системы

Символы строки представляются своими кодами в какой-либо кодировке.

Z-функция

Несколько определений

Пусть имеется строка `s[0..n)`

- *Подстрока* `sub(s,p,l)` — строка, состоящая из символов `s[p..p+l)`
- *Префикс* длины `l` — подстрока `s[0..l)`
- *Суффикс* длины `l` строки `s[0...k)` — подстрока `s[k-l..k)`
- *Собственный префикс* строки `s[0..k)` — префикс длины `l < n`
- *Собственный суффикс* строки `s[0..k)` — суффикс длины `l < n`

Z-функция

Definition

Z-функция от строки s и позиции p определяется как длина наибольшей подстроки строки s, начинающейся в позиции p, совпадающей с собственным префиксом строки s.

a	b	r	a	s	h	v	a	b	r	a	c	a	d	a	b	r	a
0	0	0	1	0	0	0	4	0	0	1	0	1	0	4	0	0	1

Как разработать алгоритм решения задачи нахождения z-функции от данной строки?

Z-функция: итерация 1

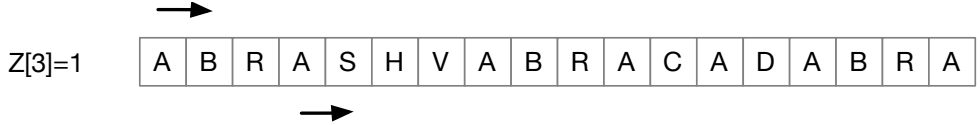
1. Обнулить выходной массив `ret`.
2. Для всех значений `j` от 1 до размера строки делать:
   1. Установить `ret[j]` равным длине максимальных совпадающих подстрок, начинающихся с 0 и с `j`.

Z-функция: итерация 1

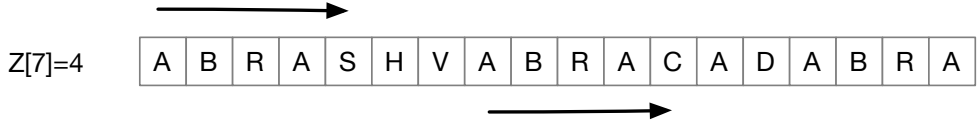
Верхняя стрелка — префикс, нижняя — подстрока с позиции p.
Конец стрелок — точка, где сопоставление закончено.

- Сопоставление для p=1

- Сопоставление для p=3



- Сопоставление для p=7



Z-функция: итерация 1

```
vector<int> z1(string const &s) {
    vector<int> ret(s.size());
    for (size_t j = 1; j < s.size(); j++) {
        size_t p = j;
        while (p < s.size() && s[p] == s[p-j])
           p++;
        ret[j] = p-j;
    }
    return ret;
}
```

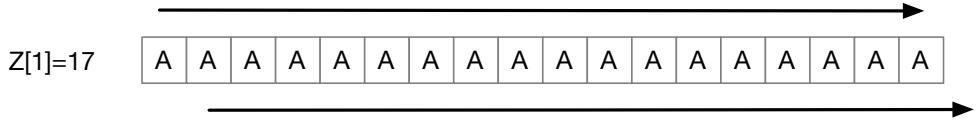
Z-функция: итерация 1

Три ключевых вопроса:

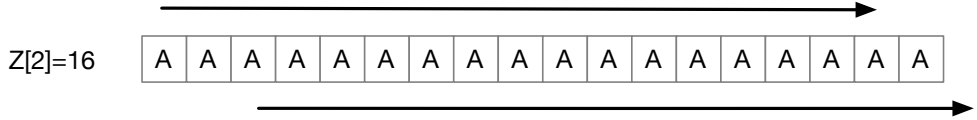
1. Корректен ли алгоритм?
2. Какова его сложность?
3. Можно ли его улучшить?

Z-функция: итерация 1: сложность

- Сопоставление для p=1



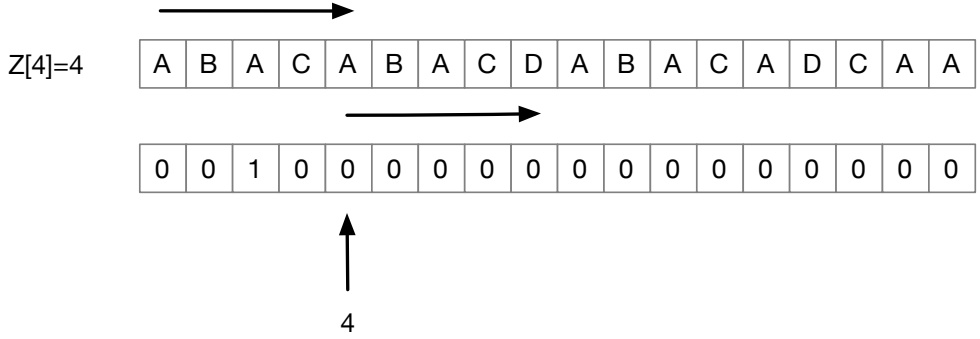
- Сопоставление для p=2



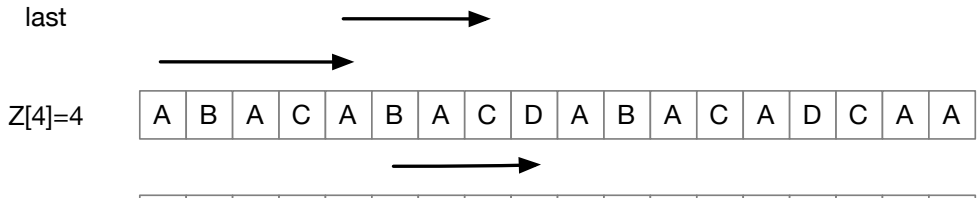
$$T(N) = (N-1) + (N-2) + \cdots + 1 = O(N^2)$$

Z-функция: наблюдение над поведением

- Сопоставление для p=4

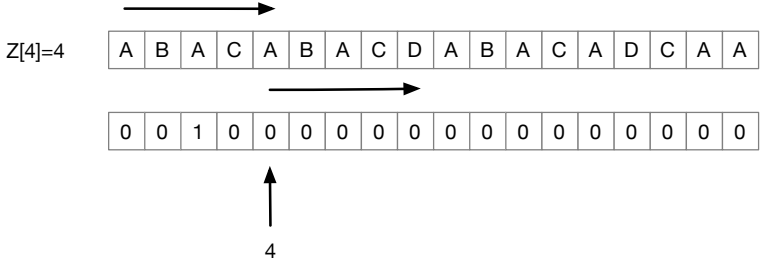


- Сопоставление для p=5

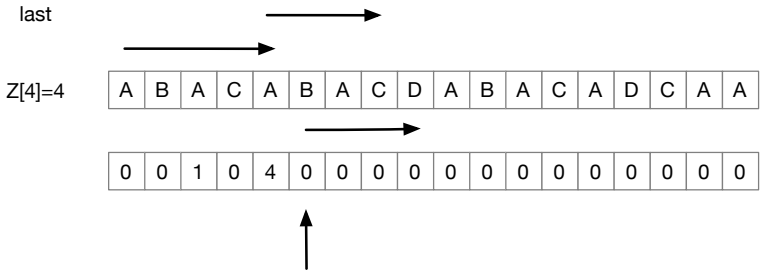


Z-функция: наблюдение над поведением

- Введём понятие *последняя сопоставленная подстрока* или *Z-блок*



- Сопоставление для p=5



- Если индекс p попадает внутрь последней сопоставленной подстроки, то сдвигаем его сразу за правую границу. Нам эта подстрока больше не нужна.
- Если индекс p *не* попадает внутрь последней сопоставленной подстроки, то ищем как обычно.

Два изменения алгоритма:

1. начало поиска теперь может сдвинуться сразу на несколько позиций;
2. при выходе индекса p за границу подстроки сопоставления меняем эту подстроку на новую.

Z-функция: итерация 2

```
vector<int> z2(string const &s) {
  vector<int> ret(s.size());
  for (int j = 1, l = 0, r = 0; j < s.size(); j++) {
    int p = j > r ? j : j+min(r-j+1, ret[j-l]);
    while (p < s.size() && s[p] == s[p-j])
      p++;
    ret[j] = p-j;
    if (p > r) {
      l = j;
      r = p-1;
    }
  }
  return ret;
}
```

Z-функция: итерация 2

Те же самые три вопроса:

1. Корректен ли алгоритм?
2. Какова его сложность?
3. Можно ли его улучшить?

Сложность алгоритма?

- Инвариант: положение конца подстроки сопоставления изменяется на её длину.
- Поиск внутри строки сопоставления — $O(L)$.
- Каждый символ строки — $O(1)$.
- Сложность всего алгоритма:

$$T(N) = O(N).$$

Префикс-функция

Definition

Префикс-функция от строки s и позиции p определяется как длина наибольшего собственного суффикса строки $s_1, s_2, \ldots, s_p$, совпадающего с префиксом той же длины.

a	b	r	a	s	h	v	a	b	r	a	c	a	d	a	b	r	a
0	0	0	1	0	0	0	1	2	3	4	0	1	0	1	2	3	4

Обозначается как $\pi(s, p)$ или π_p для заданной строки s.

Префикс-функция: вычисление

Definition

Супрефикс строки s — суффикс, совпадающий с префиксом.

Пусть для строки s известны $\pi_0, \pi_1, \ldots, \pi_{i-1}$.
Нужно вычислить π_i.

Lemma

$\pi_{i-1} \geqslant \pi_i - 1$

Доказательство.

Рассмотрим сопоставление для i.
Если мы отбросим последний символ в обоих сопоставленных подстроках, то совпадение произойдёт минимум по $\pi_i - 1$ символу. Это значит, что значение π_{i-1} не может быть меньше $\pi_i - 1$. □

Префикс-функция: вычисление

- Следствие из леммы: $\pi_i \leqslant \pi_{i-1} + 1$.
- Увеличение супрефикса происходит тогда и только тогда, когда символы за префиксом и суффиксом одинаковы.
- Если символы — разные, будем перебирать все супрефиксы, заканчивающиеся в i в порядке убывания длины.

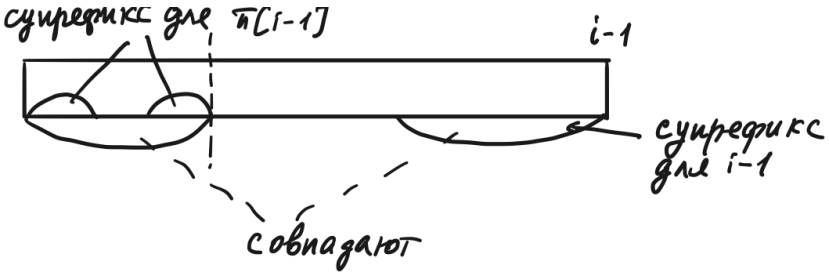
Префикс-функция: алгоритм вычисления

```
auto pi(string const &s) {
    vector<int> ret(s.size());
    for (size_t i = 1; i < s.size(); i++) {
        int m = ret[i-1]; // max possible suprefix
        while (m > 0 && s[m] != s[i]) m = ret[m-1];
        if (s[i] == s[m]) m++;
        ret[i] = m;
    }
    return ret;
}
```

Префикс-функция: алгоритм вычисления

Почему это работает?

- Нам нужен самый длинный супрефикс в предыдущей позиции.
- Что такое `ret[m-1]`?
- Это длина супрефикса в предыдущей позиции.
- Если с ним *не* совпало, ищем второй по длине супрефикс.



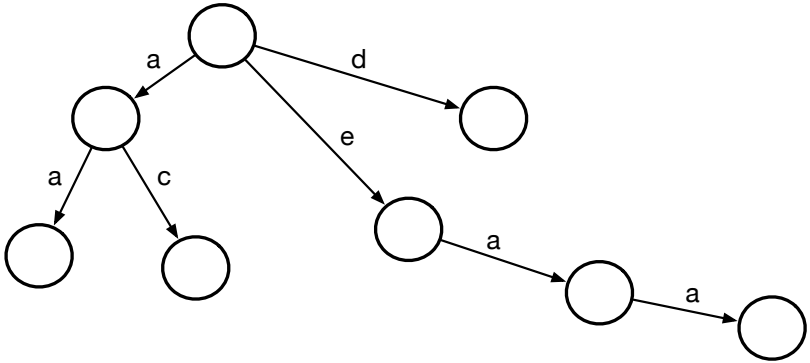
Префикс-функция: асимптотика

- Посмотрим на поведение m в функции.
- m может либо увеличиваться на единицу не более m раз, либо уменьшаться при переходе на новый супрефикс.
- m никогда не становится меньше нуля.
- Количество изменение m есть $O(|s|)$.
- Это и есть сложность алгоритма.

Бор.

Definition

Бор (trie) — корневое дерево, на рёбрах которого написаны символы, обладающее следующим свойством: все символы, исходящие из одной вершины различны.



Построение бора

Будем хранить бор в векторе узлов.
Каждый узел хранит номера узлов, ведущих по дереву.

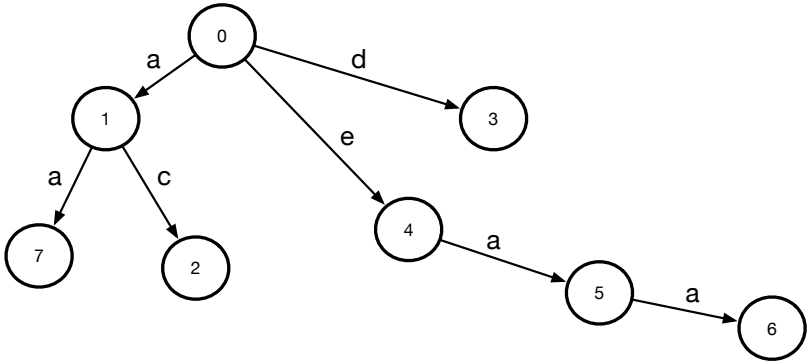
```
struct node {
    map<char,int> to; // children
    bool term;        // is it terminal?
};

vector<node> trie;
```

Построение бора

```
void addWord(string const &s) {
    int v = 0;   // root
    for (auto c: s) {
        if (trie[v].to.count(c) == 0) {
            trie.push_back(node());
            trie[v].to[c] = trie.size()-1;
        }
        v = trie[v].to[c];
    }
    trie[v].term = true;
};
```

Свойства бора

- Установим биекцию между вершинами $v \in V$ и строками, которые определяются при пути от корня до них.
- Строка eaa определяет вершину 6, вершина 6 определяет строку eaa.

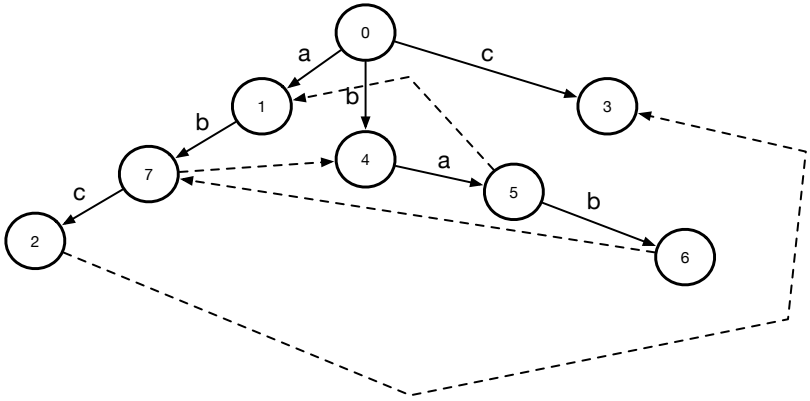


Бор как поисковая структура

- Можно ли использовать построенный бор, чтобы искать с его помощью строки в каком-то тексте?
- Пока нет.
- При неудачном поиске мы отправимся в корень, а нужно в какое-то другое место.
- В какое?
- Туда, где возможен самый длинный супрефикс.
- Для поиска одной строки можно построить автомат префикс-функцией.
- Для нескольких строк поиска модифицируем алгоритм, стоя конечный автомат алгоритмом Ахо-Корасик.

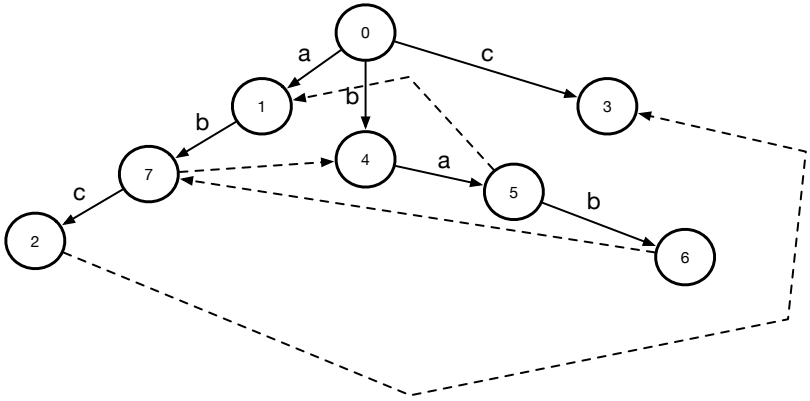
Изменение узла для автомата

- Добавим к каждой вершине v ещё одну ссылку $link(v)$, показывающую на самый длинный *собственный* суффикс строки v, имеющийся в боре.
- Вершина $link(v)$ соответствует какой-то строке.

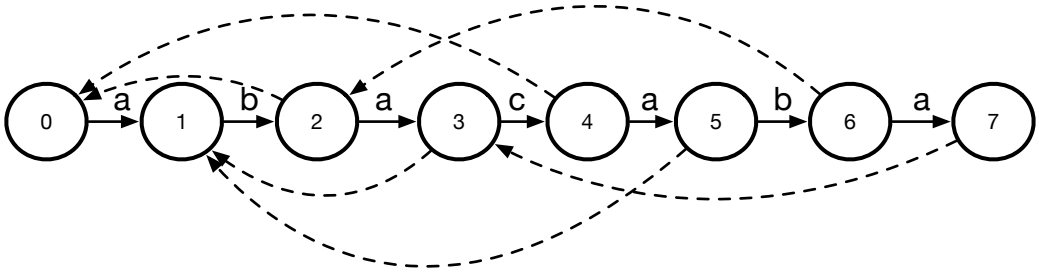


Изменение узла для автомата

- Добавим к каждой вершине v ещё одну ссылку $link(v)$, показывающую на самый длинный *собственный* суффикс строки v, имеющийся в боре.
- Вершина $link(v)$ соответствует какой-то строке.



В автомате одна строка



Мы видим работу префикс-функции.

Поиск после построения суффиксных ссылок для всех вершин

1. Дополняем каждый узел виртуально или физически вершинами с символами алфавита, для которых нет прямых ссылок вправо.
2. Для каждой такой вершины устанавливаем обратную ссылку в корень.
3. Переходим в корень.
4. Каждая очередная буква отправляет нас в новую вершину.
5. Если текущая вершина — терминальная, то соответствие найдено.

Построение суффиксных ссылок — алгоритм Ахо-Корасик

Соображения:

- Мы строим дерево с несколькими уровнями. Все суффиксные ссылки ведут на вышестоящие уровни.
- Дерево можно строить слой за слоем.
- Поле $next(v)$ можно использовать как для реальных, так и для виртуальных узлов.
- Для обхода по слоям дерева удобно применять алгоритм BFS.
- Поддерживаем очередь вершин для которых уже известны $link$ и $next$.
- Общий цикл: достаём вершину из очереди, рассматриваем всех её детей, считаем все нужные поля каждого ребёнка и отправляем в очередь.

Алгоритм Ахо-Корасик: детали реализации

Для удобства изложения поменяем структуру узла, будем хранить ссылки вниз в массиве.

```
struct node {
    static const int ASIZE = 5;
    int next[ASIZE];
    int link;
    bool term;
    node() {
        for (int i = 0; i < ASIZE; i++) next[i] = -1;
        link = -1;
        term = false;
    }
};
```

Алгоритм Ахо-Корасик: детали реализации

Инициализация пустого пока дерева.

```
struct AhoCorasick {
    vector<node> trie;
    AhoCorasick() {
        trie.push_back(node()); // Empty node = root with index 0
        trie[0].link = -1;
        for (int i = 0; i < node::ASIZE; i++) {
            trie[0].next[0] = -1;
        }
        trie[0].term = false;
    }
```

Алгоритм Ахо-Корасик: детали реализации

```
void addWord(string const &s) {
    int curr = 0;
    for (auto c: s) {
        c -= 'a';
        if (trie[curr].next[c] == -1) {
            trie.push_back(node());
            trie[curr].next[c] = trie.size() - 1;
        }
        curr = trie[curr].next[c];
    }
    trie[curr].term = true;
}
```

Алгоритм Ахо-Корасик: детали реализации

После создания объекта мы можем добавить в него нужные нам слова для формирования бора.

```
AhoCorasick ac;
ac.addWord("ab");
ac.addWord("cab");
ac.addWord("adc");
ac.addWord("abec");
```

Построение суффиксных ссылок проведём функцией $build$.

```
ac.build();
```

Алгоритм Ахо-Корасик: функция build

Функция $build$ разбивается на две части: инициализация и собственно построение автомата. Перед запуском основного цикла готовим корневой узел.

```
void build() {
    trie[0].link = 0;
    for (int i = 0; i < node::ASIZE; i++) {
        if (trie[0].next[i] >= 0) continue;
        trie[0].next[i] = 0;
    }
    queue<int> q;
    q.push(0);
```

Алгоритм Ахо-Корасик: функция build

```
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int c = 0; c < node::ASIZE; c++) {
                int u = trie[v].next[c];
                if (trie[u].link >= 0) continue;
                trie[u].link = v == 0 ? 0 : trie[trie[v].link].next[c];
                for (int d = 0; d < node::ASIZE; d++) {
                    if (trie[u].next[d] >= 0) continue;
                    trie[u].next[d]int tod = trie[trie[u].link].next[d];
                }
                q.push(u);
            }
        }
    }
};
```