

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

ОТЧЕТ
по лабораторной работе №4
по дисциплине «Построение и анализ алгоритмов»
ТЕМА: «КНУТ-МОРРИС-ПРАТТ»

Студент гр. 3388

Гусакова К.А.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализуйте алгоритм КМП и с его помощью для заданных шаблона P ($|P| \leq 25000$) и текста T ($|T| \leq 5000000$) найдите все вхождения P в T .

2. Заданы две строки A ($|A| \leq 5000000$) и B ($|B| \leq 5000000$).

Определить, является ли A циклическим сдвигом B (это значит, что A и B имеют одинаковую длину и A состоит из суффикса B , склеенного с префиксом B). Например, `defabc` является циклическим сдвигом `abcdef`.

Описание алгоритма для решения задачи

Алгоритм Кнута-Морриса-Пратта (КМП) для поиска подстроки в строке позволяет эффективно находить все вхождения одного текста (шаблона) внутри другого (основного текста) за линейное время.

1. Построение префикс-функции:

- Массив `prefix[i]` хранит длину наибольшего префикса, совпадающего с суффиксом, в подстроке `pattern[0...i]`.
- Позволяет не возвращаться к началу шаблона при несовпадении, а откатиться к наиболее подходящей позиции.

2. Поиск всех вхождений:

- Двигаемся по `text`, сверяя символы с `pattern`.
- При несовпадении переходим к позиции, заданной `prefix[q - 1]`.
- При полном совпадении ($q == m$), запоминаем позицию начала вхождения, и продолжаем поиск, не начиная с начала шаблона.

Результат: печать всех индексов, где начинается совпадение.

Для второй задачи:

1. Строим префикс-функцию для строки A - как и раньше.

2. Поиск первого вхождения A в $BB = B + B$:

- Используем ту же КМП-логику.
- При первом полном совпадении шаблона A в строке BB , возвращаем индекс этого вхождения.

- Если индекс $< B.length()$, то A - циклический сдвиг B на index позиций.

Результат:

- Если A - циклический сдвиг B, то выводится минимальный сдвиг.
- Если нет - вывод -1.

Оценка сложности алгоритма по времени и памяти

1. `vector<int> kmpSearch(const string& pattern, const string& text)`

Построение префикс-функции $O(m)$, где m - длина шаблона. Поиск вхождений $O(n)$, где n - длина текста. Временная сложность $O(n+m)$

Префикс $O(m)$, результаты $O(n)$, строки $O(m + n)$

Пространственная сложность: $O(m + n)$

2. `int kmpSearchFirstOccurrence(const string& pattern, const string& text)` - Здесь ищем первое вхождение A в $BB = B + B$. Построение префикс-функции: $O(m)$, где $m = \text{len}(A)$

Длина текста BB - $2m$, поэтому поиск: $O(2m)$

Временная сложность $O(m) + O(2m) = O(m)$

Префикс-функция - $O(m)$, никаких дополнительных структур (результат - одно число). Строка $BB = B + B$ создаётся копированием: $O(2m)$

Пространственная сложность $O(m)$

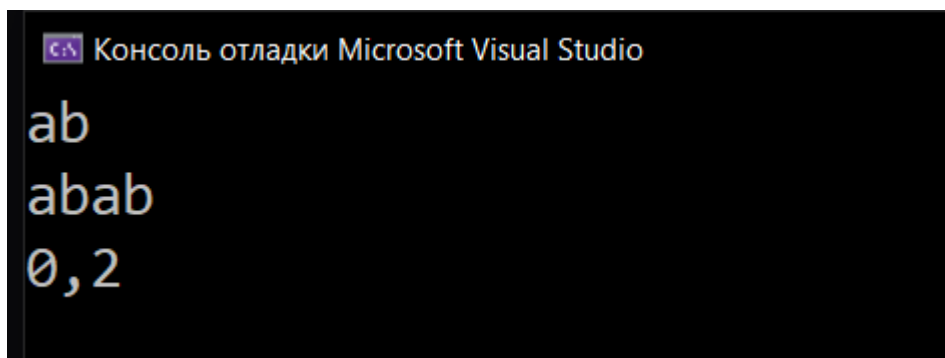
Использованные оптимизации.

Работа без дополнительных структур (нет никаких стеков, хешей и др вспомогательных структур)

Все строки и векторы передаются по константной ссылке (`const string&`), что исключает издержки на копирование данных в функции.

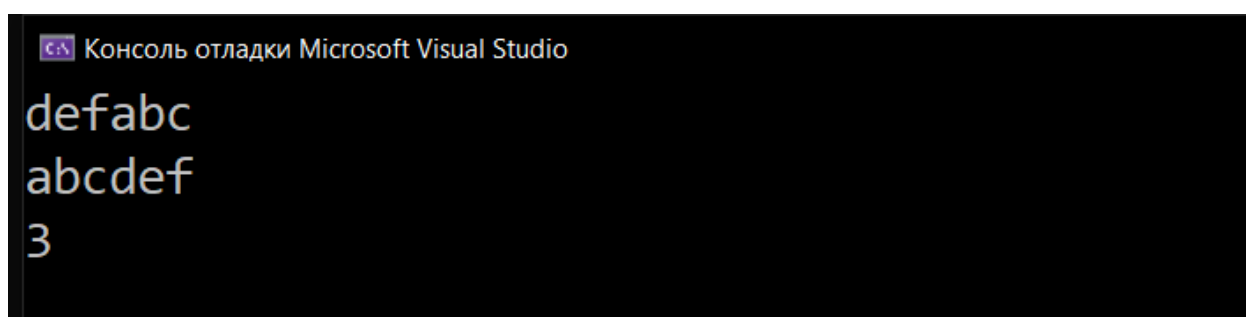
В задаче на сдвиг строк (поиск вхождения A в $B+B$) поиск завершается при первом совпадении, что экономит время.

Тестирование. Показ граничных случаев алгоритма.



```
Консоль отладки Microsoft Visual Studio  
ab  
abab  
0,2
```

Рисунок 1 – результат 1 программы.



```
Консоль отладки Microsoft Visual Studio  
defabc  
abcdef  
3
```

Рисунок 2 - результат 2 программы.

ВЫВОД

Обе программы демонстрируют эффективность алгоритма Кнута-Морриса-Пратта в задачах поиска подстрок. Благодаря предварительному построению префикс-функции поиск выполняется за линейное время, что делает его подходящим для обработки длинных строк. Первая программа показывает применение алгоритма для нахождения всех вхождений шаблона, а вторая — для определения циклического сдвига строк. Обе реализации используют одинаковую идею, но адаптированы под разные цели. Это подчеркивает универсальность алгоритма КМП и его практическую значимость для задач сопоставления строк.

ПРИЛОЖЕНИЕ. ИСХОДНЫЙ КОД ПРОГРАММЫ

1.

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <cstring>
using namespace std;

const int ALPHABET = 5;
const int MAXNODES = 300000;

int to_index(char c) {
    if (c == 'A') return 0;
    if (c == 'C') return 1;
    if (c == 'G') return 2;
    if (c == 'T') return 3;
    return 4; // N
}

char from_index(int i) {
    return "ACGTN"[i];
}

struct Node {
    int next[ALPHABET];
    int link;
    vector<int> output;
    int parent;
    char parent_char;
    Node() {
        memset(next, -1, sizeof(next));
        link = -1;
        parent = -1;
        parent_char = 0;
    }
};

Node trie[MAXNODES];
int trie_size = 1;

void add_pattern(const string& pattern, int id) {
    cout << "\nДобавляется паттерн \"" << pattern << "\" c id = " << id << endl;
    int node = 0;
    for (int i = 0; i < pattern.size(); i++) {
        int c = to_index(pattern[i]);
        if (trie[node].next[c] == -1) {
            trie[trie_size] = Node();
            trie[trie_size].parent = node;
            trie[trie_size].parent_char = pattern[i];
            trie[node].next[c] = trie_size;
        }
    }
}
```

```

        cout << "    Создана вершина " << trie_size << " из "
<< node
        << " по символу '" << pattern[i] << "'\n";
        trie_size++;
    }
    node = trie[node].next[c];
}
trie[node].output.push_back(id);
}

void build_links() {
    cout << "\n=== Построение суффиксных ссылок ===\n";
    queue<int> q;
    trie[0].link = 0;

    for (int c = 0; c < ALPHABET; c++) {
        if (trie[0].next[c] != -1) {
            trie[trie[0].next[c]].link = 0;
            q.push(trie[0].next[c]);
        }
        else {
            trie[0].next[c] = 0;
        }
    }

    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (int c = 0; c < ALPHABET; c++) {
            int u = trie[v].next[c];
            if (u != -1) {
                int f = trie[v].link;
                while (trie[f].next[c] == -1) {
                    f = trie[f].link;
                }
                trie[u].link = trie[f].next[c];
                cout << "Вершина " << u << " (символ '" <<
trie[u].parent_char << "') получает ссылку на "
                    << trie[u].link << endl;

                for (int j = 0; j <
trie[trie[u].link].output.size(); j++) {
                    int out = trie[trie[u].link].output[j];
                    trie[u].output.push_back(out);
                    cout << "    Наследует выход: " << out <<
endl;
                }

                q.push(u);
            }
        }
    }

    cout << "\n=== Характеристики вершин ===\n";

```

```

        for (int i = 0; i < trie_size; i++) {
            cout << "Вершина " << i << ": ";
            if (i != 0) {
                cout << "родитель = " << trie[i].parent << " ('" <<
trie[i].parent_char << "'), ";
            }
            cout << "link = " << trie[i].link << ", переходы: ";
            for (int j = 0; j < ALPHABET; j++) {
                if (trie[i].next[j] != -1) {
                    cout << from_index(j) << "->" << trie[i].next[j]
<< " ";
                }
            }
            if (!trie[i].output.empty()) {
                cout << ", выходы: ";
                for (int x : trie[i].output) cout << x << " ";
            }
            cout << endl;
        }
    }

int main() {
    setlocale(LC_ALL, "russian");
    string T;
    int n;
    cin >> T >> n;

    vector<string> patterns(n);
    vector<int> pattern_lengths(n);

    for (int i = 0; i < n; i++) {
        cin >> patterns[i];
        pattern_lengths[i] = patterns[i].size();
        add_pattern(patterns[i], i + 1);
    }

    build_links();

    cout << "\n=== Обработка текста ===\n";
    vector<pair<int, int>> result;
    int node = 0;
    for (int i = 0; i < T.size(); i++) {
        int c = to_index(T[i]);
        while (trie[node].next[c] == -1) {
            node = trie[node].link;
        }
        node = trie[node].next[c];

        cout << "Символ '" << T[i] << "' на позиции " << i + 1
<< ", перешли в вершину " << node << endl;

        for (int j = 0; j < trie[node].output.size(); j++) {
            int pattern_id = trie[node].output[j];

```

```

        int pos = i - pattern_lengths[pattern_id - 1] + 2;
        cout << "    Найден паттерн " << pattern_id << " на
позиции " << pos << endl;
        result.push_back({ pos, pattern_id });
    }
}

sort(result.begin(), result.end());

cout << "\n=== Результаты совпадений (позиция, ID шаблона)
===\n";
for (const auto& p : result) {
    cout << p.first << " " << p.second << "\n";
}

return 0;
}

```

2.

```

#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
#include <algorithm>
using namespace std;

const int ALPHABET = 5;
const int MAXNODES = 300000;

int to_index(char c) {
    if (c == 'A') return 0;
    if (c == 'C') return 1;
    if (c == 'G') return 2;
    if (c == 'T') return 3;
    return 4; // 'N'
}

char from_index(int i) {
    return "ACGTN"[i];
}

struct Node {
    int next[ALPHABET];
    int link;
    int terminal_link;
    vector<int> output;
    int parent;
    char parent_char;
    Node() {
        memset(next, -1, sizeof(next));
        link = -1;
        terminal_link = -1;
        parent = -1;
        parent_char = 0;
    }
}

```



```

    }
};

Node trie[MAXNODES];
int trie_size = 1;

struct SubPattern {
    string s;
    int offset;
};

void add_pattern(const string& pattern, int id) {
    int node = 0;
    for (int i = 0; i < pattern.size(); i++) {
        int c = to_index(pattern[i]);
        if (trie[node].next[c] == -1) {
            trie[trie_size] = Node();
            trie[trie_size].parent = node;
            trie[trie_size].parent_char = pattern[i];
            trie[node].next[c] = trie_size;

            cout << "Добавлена вершина " << trie_size << ": " <<
pattern[i] << " из " << node << endl;

            trie_size++;
        }
        node = trie[node].next[c];
    }
    trie[node].output.push_back(id);
    cout << "Паттерн \"" << pattern << "\" добавлен в вершину "
<< node << " (ID: " << id << ")\n";
}

void build_links() {
    cout << "\n=== Построение суффиксных и терминальных ссылок
===\n";

    queue<int> q;
    trie[0].link = 0;

    for (int c = 0; c < ALPHABET; c++) {
        if (trie[0].next[c] != -1) {
            trie[trie[0].next[c]].link = 0;
            q.push(trie[0].next[c]);
        }
        else {
            trie[0].next[c] = 0;
        }
    }

    while (!q.empty()) {
        int v = q.front();
        q.pop();
    }
}

```

```

        for (int c = 0; c < ALPHABET; c++) {
            int u = trie[v].next[c];
            if (u != -1) {
                int f = trie[v].link;
                while (trie[f].next[c] == -1) {
                    f = trie[f].link;
                }
                trie[u].link = trie[f].next[c];
                trie[u].terminal_link =
(!trie[trie[u].link].output.empty())
                ? trie[u].link
                : trie[trie[u].link].terminal_link;

                cout << "Вершина " << u << ": суффиксная ссылка
-> " << trie[u].link
                << ", терминальная ссылка -> " <<
trie[u].terminal_link << endl;

                q.push(u);
            }
        }
    }

    cout << "\n=== Характеристики автомата (всех вершин) ===\n";
    for (int i = 0; i < trie_size; i++) {
        cout << "Вершина " << i << ": ";
        if (i != 0) {
            cout << "родитель = " << trie[i].parent << " (" <<
trie[i].parent_char << "'), ";
        }
        cout << "link = " << trie[i].link << ", terminal_link =
" << trie[i].terminal_link << ", ";
        cout << "переходы: ";
        for (int j = 0; j < ALPHABET; j++) {
            if (trie[i].next[j] != -1) {
                cout << from_index(j) << "->" << trie[i].next[j]
<< " ";
            }
        }
        if (!trie[i].output.empty()) {
            cout << ", выходы: ";
            for (int id : trie[i].output) cout << id << " ";
        }
        cout << endl;
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    string T, P;
    char wildcard;
    cin >> T >> P >> wildcard;

```

```

vector<SubPattern> subpatterns;
string current = "";
int offset = 0;

for (int i = 0; i < P.size(); i++) {
    if (P[i] == wildcard) {
        if (!current.empty()) {
            subpatterns.push_back({ current, offset });
            current = "";
        }
        offset = i + 1;
    }
    else {
        if (current.empty()) offset = i;
        current += P[i];
    }
}
if (!current.empty()) {
    subpatterns.push_back({ current, offset });
}

cout << "\n=== Разбиение шаблона на подстроки ===\n";
for (int i = 0; i < subpatterns.size(); i++) {
    cout << i << ": \"" << subpatterns[i].s << "\" (offset = " << subpatterns[i].offset << ")\n";
    add_pattern(subpatterns[i].s, i);
}

build_links();

int m = P.size();
vector<int> match(T.size(), 0);

cout << "\n=== Обработка текста ===\n";
int node = 0;
for (int i = 0; i < T.size(); i++) {
    int c = to_index(T[i]);
    while (trie[node].next[c] == -1)
        node = trie[node].link;
    node = trie[node].next[c];

    cout << "Символ '" << T[i] << "' (позиция " << i + 1 <<
    ")", перешли в вершину " << node << endl;

    int temp = node;
    while (temp != -1 && temp != 0) {
        for (int j = 0; j < trie[temp].output.size(); j++) {
            int pat_id = trie[temp].output[j];
            int pos = i - subpatterns[pat_id].s.size() + 1 -
subpatterns[pat_id].offset;
            if (0 <= pos && pos + m <= T.size()) {
                match[pos]++;
            }
        }
        temp = trie[temp].link;
    }
}

```

```

        cout << "    Найден шаблон " << pat_id << " на
позиции " << pos << endl;
    }
}
temp = trie[temp].terminal_link;
}
}

cout << "\n=== Все совпадения шаблона (позиции начала
шаблона) ===\n";
vector<int> found_positions;
for (int i = 0; i < T.size(); i++) {
    if (match[i] == subpatterns.size()) {
        cout << (i + 1) << "\n";
        found_positions.push_back(i);
    }
}

int max_out = 0;
for (int i = 0; i < trie_size; i++) {
    int cnt = 0;
    for (int j = 0; j < ALPHABET; j++)
        if (trie[i].next[j] != -1) cnt++;
    max_out = max(max_out, cnt);
}
cout << "\nМакс. количество исходящих рёбер из вершины: " <<
max_out << endl;

cout << "\n=== Строка без вхождений шаблона ===\n";
vector<bool> masked(T.size(), false);
for (int start : found_positions) {
    for (int i = start; i < start + m; i++) {
        if (i < T.size()) masked[i] = true;
    }
}

for (int i = 0; i < T.size(); i++) {
    if (!masked[i]) cout << T[i];
}
cout << endl;

return 0;
}

```