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

ОТЧЕТ

по лабораторной работе № 5
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Ахо-Корасик

Студент гр. 1384 _____ Галенко А.С.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p1, \ldots, pn\} 1 \leq |pi| \leq 75$

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 P в текст T.

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≤|T|≤100000)

Шаблон (P,1≤|P|≤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

Выполнение работы.

Для решения первого задания был описан класс Aho_Corasick_tree, предоставляющий следующий интерфейс, позволяющий решить первую задачу:

- *void addStringtoBor(string line)*: Позволяет добавить строку в бор
- *Node getSuffixLink(Node *cur_node)*: Возвращает указатель элемента в векторе вершин бора, являющейся суффиксной ссылкой элемента *cur_node*.
- *Node getTerminalSuffixLink(Node *cur_node)*: Возвращает индекс «сжатой» (терминальной) суффиксной ссылки, т.е. такой, на которой заканчивается какое-либо слово.

- *Node* transit_across_sym(Node* cur_node, char sym)* Возвращает указатель элемента, соответствующий вершине, связанной с вершиной *node* ребром со значением *symb*. Если таковой нет, возвращается указатель элемента, связанного с вершиной суффиксной ссылкой *node* ребром со значением *symb*.
- *void find_samples(std::string &text, std::map<std::string, int> &patterns)*: Находит и сохраняет индексы вхождения сохранённых ранее строк в *text*. Map нужен для сохранения в очередь с приоритетом, которая в выводе даст отсортированный по индексу каждого паттерна список.
- void printRes(): Выводит список (ответ), для этого в цикле извлекается максимальный элемент структуры Text_Entry, и выводятся его поля на экран.

После ввода данных в main все строки-паттерны добавляются в бор, после чего вызывается *findSamples* для поданного текста. Список с ответом сортируется согласно условиям задачи и выводится его содержимое с помощью printRes().

Для решения второй задачи были добавлены следующие функции в класс:

- void convertPatternToBor(std::string &pattern, const char &wildcard) : Находит подстроки каждого паттерна, разделенные символом джокера (wildcard), и добавляет их в бор.
- void findAnswer(const std::string &text, const std::string &pattern) : функция итогового поиска шаблона в строке text

Был модифицирован метод *addStringtoBor* класса – теперь в классе есть массив bor_substrings, который сохраняет пары: указатель на вершину и индекс начала подстроки в шаблоне. При создании терминальной вершины в данный массив заносится соответствующая информация.

Выводы.

В процессе выполнения лабораторной работы был реализован алгоритм Ахо-Корасик. Были разработаны программы для точного поиска набора образцов в строке, а также для поиска всех вхождений строки, содержащей символ-джокер в тексте.

ПРИЛОЖЕНИЕ 1
КОД ДЛЯ ПЕРВОЙ ЗАДАЧИ

task1.cpp:

```
#include <iostream>
#include <map>
#include <vector>
#include <queue>

/**
 * Структура Bor_Node определяет вершину в Боре,
 * is_terminal - флаг, является ли вершина терминалом
 * terminal_line - вся строка, которая сохраняется в конечной вершине,
 * parent_node - вершина-родитель,
 * suffix_pointer - суффиксная ссылка,
 * terminal_suffix_pointer - сжатая суффиксная ссылка,
 * parent_symbol - символ строки в данной вершине,
 * sons_nodes - вектор дочерних вершины,
 * transitions_array - вектор переходов (запоминаем переходы в ленивой
рекурсии),
 * используемый для вычисления суффиксных ссылок.
 */

struct Node
{
    bool is_terminal = false;
    std::string terminal_line;
    Node *parent_node = nullptr;
    Node *suffix_pointer = nullptr;
    Node *terminal_suffix_pointer = nullptr;
    char parent_symb;
    std::vector<Node*> sons_nodes;
    std::vector<Node*> transitions_array;
};

struct Text_Entry
{
    Text_Entry(int pos, int num)
            : pos_index(pos) , list_number(num) {}
    int pos_index;
    int list_number;
};

struct less_Text_Entry
{
    bool operator() (const Text_Entry *first, const Text_Entry *second) const
    {
        if(first->pos_index != second->pos_index)
            return first->pos_index > second->pos_index;
        else
            return first->list_number > second->list_number;
    }
};
```

```
      class Aho_Corasick_tree
      {

          std::vector<char> alphabet;
          std::priority_queue<Text_Entry*, std::vector<Text_Entry*>,
less_Text_Entry> answer_priority_queue;
          Node *root;

          int getAlphabetIndex(char sym)
          {
              for(int index = 0; index < alphabet.size(); index++)
              {
                  if(alphabet[index] == sym)
                      return index;
              }
              return -1;
          }
      public:
          Aho_Corasick_tree(std::vector<char> alphabet)
                  : alphabet(alphabet)
          {
              root = new Node;
              root->transitions_array.resize(alphabet.size());
              root->sons_nodes.resize(alphabet.size());
          }

          Node *getSuffixLink(Node *cur_node)
          {
              if(!cur_node->suffix_pointer) {
                  if (cur_node == root or cur_node->parent_node == root)
                      cur_node->suffix_pointer = root;
                  else
                      cur_node->suffix_pointer =
transit_across_sym(getSuffixLink(cur_node->parent_node),
                                                                      cur_node-
>parent_symb);
              }

              return cur_node->suffix_pointer;
          }

          Node *getTerminalSuffixLink(Node *cur_node)
          {
              if(!cur_node->terminal_suffix_pointer)
              {
                  Node *cur_suffix = getSuffixLink(cur_node);
                  if(cur_suffix == root)
                      cur_node->terminal_suffix_pointer = root;
                  else
                      cur_node->terminal_suffix_pointer = (cur_suffix->is_terminal)
                                                          ? cur_suffix :
getTerminalSuffixLink(cur_suffix);
              }
              return cur_node->terminal_suffix_pointer;
          }

          Node* transit_across_sym(Node* cur_node, char sym)
          {
```

```
            auto &cur_transitions = cur_node->transitions_array;
            int sym_index = getAlphabetIndex(sym);
            if(!cur_transitions[sym_index])
            {
                if(cur_node->sons_nodes[sym_index])
                    cur_transitions[sym_index] = cur_node->sons_nodes[sym_index];
                else if(cur_node == root)
                    cur_transitions[sym_index] = root;
                else
                    cur_transitions[sym_index] =
transit_across_sym(getSuffixLink(cur_node), sym);
            }
            return cur_transitions[sym_index];
        }

        void addStringtoTree(std::string line)
        {
            Node *cur_node = root;
            for(auto symb : line)
            {
                int symb_index = getAlphabetIndex(symb);
                auto &cur_sons = cur_node->sons_nodes;
                if(!cur_sons[symb_index])
                {
                    cur_sons[symb_index] = new Node;
                    cur_sons[symb_index]->parent_node = cur_node;
                    cur_sons[symb_index]->parent_symb = symb;
                    cur_sons[symb_index]-
>transitions_array.resize(alphabet.size());
                    cur_sons[symb_index]->sons_nodes.resize(alphabet.size());
                }
                cur_node = cur_sons[symb_index];
            }
            cur_node->is_terminal = true;
            cur_node->terminal_line = line;
        }


        void find_samples(std::string &text, std::map<std::string, int>
&patterns)
        {
            Node *cur_node = root;
            for(int index = 0; index < text.size(); index++)
            {
                cur_node = transit_across_sym(cur_node, text[index]);
                auto i_node = cur_node;
                while (i_node != root)
                {
                    if(i_node->is_terminal)
                        answer_priority_queue.push(new Text_Entry(index + 2 -
i_node->terminal_line.size(),

patterns[i_node->terminal_line]));
                    i_node = getTerminalSuffixLink(i_node);
                }
            }

        }
```

```
    void printRes()
    {
        while(!answer_priority_queue.empty())
        {
            Text_Entry *cur_entry = answer_priority_queue.top();
            std::cout << cur_entry->pos_index << ' '
                      << cur_entry->list_number << '\n';
            answer_priority_queue.pop();
        }
    }
};

int main()
{
    Aho_Corasick_tree solver_tree({'A', 'C', 'G', 'T', 'N'});
    std::string text, temp_line;
    std::map<std::string, int> patterns;
    int pattern_num;
    std::cin >> text;
    std::cin >> pattern_num;
    for(int i = 0; i < pattern_num; i++)
    {
        std::cin >> temp_line;
        solver_tree.addStringtoTree(temp_line);
        patterns[temp_line] = i + 1;
    }

    solver_tree.find_samples(text, patterns);
    solver_tree.printRes();

    return 0;
}
```

ПРИЛОЖЕНИЕ 2
КОД ДЛЯ ВТОРОЙ ЗАДАЧИ

task2.cpp:

```
#include <iostream>
#include <vector>
#include <algorithm>

/**
 * Структура Bor_Node определяет вершину в Боре,
 * is_terninal - флаг, является ли вершина терминалом
 * terminal_line - вся строка, которая сохраняется в конечной вершине,
 * parent_node - вершина-родитель,
 * suffix_pointer - суффиксная ссылка,
 * terminal_suffix_pointer - сжатая суффиксная ссылка,
 * symbol - символ строки в данной вершине,
 * text_position - вектор для хранения найденных индексов
 * вхождения строки (терминальной вершины) в тексте,
 * sons_nodes - вектор дочерних вершины,
 * transitions_array - вектор переходов (запоминаем переходы в ленивой
рекурсии),
 * используемый для вычисления суффиксных ссылок.
 */
struct Bor_Node
{
    bool is_terminal = false;
    std::string terminal_line;
    Bor_Node *parent_node = nullptr;
    Bor_Node *suffix_pointer = nullptr;
    Bor_Node *terminal_suffix_pointer = nullptr;
    char symbol;
    std::vector<int> text_positions;
    std::vector<Bor_Node*> sons_nodes;
    std::vector<Bor_Node*> transitions_array;
};

/**
 *
 */
class Aho_Corasick_tree
{
    // алфавит текста и шаблона
    std::vector<char> alphabet;

    // вектор пар <подстрока шаблона в Боре> - <стартовая позиция в шаблоне>
    std::vector<std::pair<Bor_Node*, int>> bor_substrings;

    // вектор для определения стартовых позиций шаблона в тексте
```

```
    std::vector<int> search_array;

    // корневая вершина Бора
    Bor_Node *root;

    /**
     * Функция, рассчитывающая индекс в массиве sonds_nodes или
transitions_array
     * для символа из алфавита
     * @param sym - символ, который нужно перевести в индекс
     * @return -1, если символа нет в алфавите, в ином случае - найденную
позицию в алфавите
     */
    int getAlphabetIndex(char sym)
    {
        for(int index = 0; index < alphabet.size(); index++)
        {
            if(alphabet[index] == sym)
                return index;
        }
        return -1;
    }
public:
    /**
     * Конструктор класса, решающего задачу поиска шаблонов с масками
     * @param alphabet - принимает в аргументы алфавит текста и паттернов
     */
    explicit Aho_Corasick_tree(const std::vector<char>& alphabet)
            : alphabet(alphabet)
    {
        root = new Bor_Node;
        root->transitions_array.resize(alphabet.size());
        root->sons_nodes.resize(alphabet.size());
    }

    /**
     * Функция для поиска суффиксной ссылки
     * @param cur_node - верщина бора, для которой нужно найти
     * суффиксную ссылку
     * @return указатель на найденную вершину
     */
    Bor_Node *getSuffixLink(Bor_Node *cur_node)
    {
        if(!cur_node->suffix_pointer) {
            if (cur_node == root or cur_node->parent_node == root)
                cur_node->suffix_pointer = root;
            else
                cur_node->suffix_pointer =
transit_across_sym(getSuffixLink(cur_node->parent_node),
```

```
                                                                cur_node-
>symbol);
        }

        return cur_node->suffix_pointer;
    }

    /**
     * Рекурсивная функция для поиска сжатой суффиксной ссылки
     * @param cur_node вершина бора, для которой нужно найти
     * такую суффиксную ссылку
     * @return указатель на найденную вершину
     */
    Bor_Node *getTerminalSuffixLink(Bor_Node *cur_node)
    {
        if(!cur_node->terminal_suffix_pointer)
        {
            Bor_Node *cur_suffix = getSuffixLink(cur_node);
            if(cur_suffix == root)
                cur_node->terminal_suffix_pointer = root;
            else
                cur_node->terminal_suffix_pointer = (cur_suffix->is_terminal)
                                                    ? cur_suffix :
getTerminalSuffixLink(cur_suffix);
        }
        return cur_node->terminal_suffix_pointer;
    }

    /**
     * Функция для вычисления перехода между вершинами Бора
     * @param cur_node - вершина, для которой нужно найти переход по символу
     * @param sym - символ перехода
     * @return указатель на найденную вершину
     */
    Bor_Node* transit_across_sym(Bor_Node* cur_node, const char sym)
    {
        auto &cur_transitions = cur_node->transitions_array;
        int sym_index = getAlphabetIndex(sym);
        if(!cur_transitions[sym_index])
        {
            if(cur_node->sons_nodes[sym_index])
                cur_transitions[sym_index] = cur_node->sons_nodes[sym_index];
            else if(cur_node == root)
                cur_transitions[sym_index] = root;
            else
                cur_transitions[sym_index] =
transit_across_sym(getSuffixLink(cur_node), sym);
        }
        return cur_transitions[sym_index];
    }
```

```
    /**
     * Функция для поиска подстрок, которые разделены
     * символом wildcard в pattern. Найденные строки добавляются в Бор
     * @param pattern - шаблон поиска
     * @param wildcard - символ джокера
     */
    void convertPatternToBor(std::string &pattern, const char &wildcard)
    {
        std::string substring;
        int index;
        auto position = [&] () -> int { return int(index - substring.length()
+ 1); };
        for(index = 0; index < pattern.size(); index++)
        {
            if(pattern[index] != wildcard)
                substring.push_back(pattern[index]);
            else if(!substring.empty())
            {
                addStringToTree(substring, position());
                substring.clear();
            }
        }
        if(!substring.empty())
            addStringToTree(substring, position());

    }

    /**
     * Функция для добавления строки line в Бор
     * @param line - строка, которую нужно добавить
     * @param pattern_position - индекс начала line в строке шаблона,
     * который необходимо сохранить в конечной вершине бора
     */
    void addStringToTree(std::string &line, int pattern_position)
    {
        Bor_Node *cur_node = root;
        for(int index = 0; index < line.size(); index++)
        {
            int symb_index = getAlphabetIndex(line[index]);
            auto &cur_sons = cur_node->sons_nodes;
            if(!cur_sons[symb_index])
            {
                cur_sons[symb_index] = new Bor_Node;
                cur_sons[symb_index]->parent_node = cur_node;
                cur_sons[symb_index]->symbol = line[index];
                cur_sons[symb_index]-
>transitions_array.resize(alphabet.size());
                cur_sons[symb_index]->sons_nodes.resize(alphabet.size());
            }
```

```
            cur_node = cur_sons[symb_index];
        }
        cur_node->is_terminal = true;
        cur_node->terminal_line = line;
        bor_substrings.push_back(std::make_pair(cur_node, pattern_position));
    }


    /**
     * Функция для поиска подстрок из Бора в
     * строке text
     * @param text - строка, по которой производится поиск
     */
    void find_samples(const std::string &text)
    {
        Bor_Node *cur_node = root;
        int finded_index;
        for(int index = 0; index < text.size(); index++)
        {
            cur_node = transit_across_sym(cur_node, text[index]);
            auto i_node = cur_node;
            while (i_node != root)
            {
                if(i_node->is_terminal)
                {
                    finded_index = index - i_node->terminal_line.length() +
2;
                    i_node->text_positions.push_back(finded_index);
                }
                i_node = getTerminalSuffixLink(i_node);
            }
        }
    }

    /**
     * Функция итогового поиска шаблона в text
     * @param text - строка, по которой производится поиск
     * @param pattern - шаблон
     */
    void findAnswer(const std::string &text, const std::string &pattern)
    {
        find_samples(text);
        search_array.resize(text.length());
        int substrings_count = (int) bor_substrings.size();
        std::fill(search_array.begin(), search_array.end(), 0);
        for(auto cur_subsring_node : bor_substrings)
        {
            for(auto cur_text_position : cur_subsring_node.first-
>text_positions)
            {
```

```
                int index = cur_text_position - cur_subsring_node.second + 1;
                if(index >= 0 and text.length() >= pattern.length() + index -
1)
                    search_array[index]++;
            }
        }

        for(int index = 0; index < search_array.size(); index++)
        {
            if(search_array[index] == substrings_count)
                std::cout << index << "\n";
        }

    }

};

int main()
{
    Aho_Corasick_tree solver_tree({'A', 'C', 'G', 'T', 'N'});
    std::string text, pattern;
    char joker;
    std::cin >> text;
    std::cin >> pattern;
    std::cin >> joker;

    solver_tree.convertPatternToBor(pattern, joker);
    solver_tree.findAnswer(text, pattern);

    return 0;
}
```