Цель работы.

Изучить алгоритм Ахо-Корасик поиска вхождений нескольких подстрок в текст. Реализовать его для поиска набора образцов и поиска шаблонной строки в тексте на одном из языков программирования.

Задание.

Задание №1:

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст $(T, 1 \le |T| \le 10000)$.

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

Все строки содержат символы из алфавита $\{A, C, G, T, N\}$

Выход:

Все вхождения образцов из Pв 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.

Выполнение работы.

Был реализован алгоритм Ахо-Корасик, который находит набор шаблонов в тексте.

Был написан класс Vertex, который является представлением узла бора.

Поля класса Vertex:

- vertex – символ в узле;
- isTerminal – является ли вершина терминальной;
- parent – указатель на вершину-родителя;

- child – указатели на вершины детей;
- suffixLink – суффиксальная ссылка;
- terminalLink – терминальная суффиксальная ссылка;
- number – номер подстроки;
- stringLen – длина подстроки.

Также были реализованы следующие функции:

1. createTrie() – создает автомат из переданных подстрок;
2. freeMemory() – очищение памяти;
3. ahoCorasick() – функция, реализующая алгоритм Ахо-Корасик.

Реализация программы представлена в приложении А.

Сложность алгоритма.

Так как таблица сыновей каждой вершины хранится в std::map, основой которого является красно-черное дерево, то расход памяти - $O(n)$, *где n – сумма длин всех подстрок из набора*.

А сложность алгоритма по времени его выполнения: $O((H+n)\log a+k)$, гдеH – длина текста, a- количество символов в алфавите,k – количество символов со всех совпадениях.

Выводы.

Реализована программа, которая решает задачу точного поиска набора образцов в тексте (алгоритм Ахо-Корасик).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

class Vertex
{
public:
   Vertex(char vertex_ = '?', Vertex *parent_ = nullptr) :
vertex(vertex_), parent(parent_) {}
  char vertex = '?';
  bool isTerminal = false;
  Vertex *parent = nullptr;
  Vertex *suffixLink = nullptr;
  Vertex *terminalLink = nullptr;
  int number = 0;
  int stringLen = 0;
  std::map<char, Vertex *> child;
};

Vertex *createTrie(std::vector<std::string> pattern)
{
  Vertex *root = new Vertex();
  int count = 0;

  for (auto elem : pattern)
  {
    Vertex *current = root;
    for (int i = 0; i < elem.size(); ++i)
    {
      Vertex *previous = current;
              if  (current->child.find(elem[i])  ==  current-
>child.end())
               current->child[elem[i]]  =  new  Vertex(elem[i],
previous);
      current = current->child[elem[i]];

      if (elem.size() - 1 == i)
      {
```

```
            current->isTerminal = true;
            current->number = count++;
            current->stringLen = elem.size();
          }
        }
      }

      std::queue<Vertex *> queue;
      queue.push(root);

      while (queue.size())
      {
        Vertex *current = queue.front();
        queue.pop();

        for (auto elem : current->child)
        {
          queue.push(elem.second);
        }

        if (current->parent != nullptr)
        {
          Vertex *suffix = current->parent->suffixLink;
          char vertex = current->vertex;

           while (suffix != nullptr && suffix->child.find(vertex)
== suffix->child.end())
            suffix = suffix->suffixLink;

          if (suffix != nullptr)
            current->suffixLink = suffix->child[vertex];
          else
            current->suffixLink = root;

          if (current->suffixLink->isTerminal)
            current->terminalLink = current->suffixLink;
        }
      }

      return root;
    }

    void freeMemory(Vertex *root)
    {
      for (auto elem : root->child)
      {
```

```
        freeMemory(elem.second);
      }
      delete root;
    }

    std::set<std::pair<int, int>> ahoCorasick(std::string text,
std::vector<std::string> patterns)
    {
      std::set<std::pair<int, int>> result;
      Vertex *root = createTrie(patterns);

      Vertex *current = root;
      for (int i = 0; i < text.size(); ++i)
      {
        while (true)
        {
          Vertex *terminal = current->terminalLink;
          while (terminal != nullptr)
          {
                result.insert({i - terminal->stringLen, terminal-
>number});
            terminal = terminal->terminalLink;
          }

                if (current->child.find(text[i]) != current-
>child.end())
          {
            current = current->child[text[i]];
            if (current->isTerminal)
               result.insert({i - current->stringLen + 1, current-
>number});
            break;
          }
          else if (current == root)
            break;
          else
            current = current->suffixLink;
        }
      }

      freeMemory(root);

      return result;
    }

    int main()
```

```
    {
      std::string text;
      int numbers;
      std::vector<std::string> patterns;

      std::cin >> text >> numbers;

      for (int i = 0; i < numbers; ++i)
      {
        std::string temp;
        std::cin >> temp;
        patterns.push_back(temp);
      }

      auto result = ahoCorasick(text, patterns);

      for (auto elem : result)
      {
         std::cout << elem.first + 1 << ' ' << elem.second + 1 <<
'\n';
      }

      return 0;
}
```