МИНОБР НАУКИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА — Российский технологический университет»

РТУ МИРЭА

Отчёт по выполнению практического задания 9
Тема: Поиск в тексте образца. Алгоритмы. Эффективность алгоритмов
Дисциплина Структуры и алгоритмы обработки данных

Выполнил студент Пак С.А.

группа ИКБО-05-20

Москва 2021

СОДЕРЖАНИЕ

Номер варианта: 5

ОТЧЁТ ПО ЗАДАНИЮ 1

1. Разработка программы

1. Постановка задачи

Дано предложение, состоящее из слов, разделённых одним пробелом. Удалить из него слова, встретившиеся более одного раза.

2. Подход к решению и алгоритмы

Для решения этой задачи нужно разработать и реализовать две дополнительные функции.

Первая функция *findFirst* будет искать индекс первого вхождения подстроки. Вторая функция *findLast* будет искать индекс последнего вхождения подстроки. Обе функции в случае неудачного поиска возвращают отрицательное значение (например, -1).

С помощью этих двух функций и возможностей функционального программирования, можно очень просто решить задачу.

Используя метод массива *filter*, который возвращает те элементы массива, для которых функция, переданная в качестве аргумента, возвращает логическое значение *true*.

Суть в том, что обе функции *findFirst* и *findLast* возвратят одно и то же значение, если слово входит в предложение только один раз. Собственно, это и будет ключевым условием для метода *filter*.

То есть, в качестве аргумента метода *filter* нужно передать функцию, которая возвращает значение *true*, если функции *findFirst* и *findLast* вернули одно и то же значение.

Алгоритм для функции *findFirst*:

1) Цикл for для i = 0, i < длина строки-длина образца+1;

2) Выстравиваем подстроку такой же длины, как и образец, от текущего индекса i;

3) Если выстроенная подстрока совпадает с образцом, то вернуть значение i;

4) Если цикл завершился, то вернуть значение -1.

Алгоритм для функции *findLast*:

1) Цикл for для i = длина строки-1, i >= длина образца-1;

2) Выстраиваем подстроку такой же длины, как и образец, от subIndex = i-длина образца+1;

3) Если выстроенная подстрока совпадает с образцом, то вернуть значение subIndex;

4) Если цикл завершился, то вернуть значение -1.

Пример: пусть предложение - «He is Sergey from Russia which is a cold country». Найти первое вхождение и последнее вхождение слова «is».

Функция *findFirst* проверяет каждую подстроку длины 2 в предложении и ищет совпадение. При этом проверка идёт сначала до конца.

Функция *findLast* проверяет каждую подстроку длины 2 в предложении и ищет совпадение. При этом проверка идёт с конца.

Все проверяемые подстроки для функции *findFirst*: [«He», «e », « i», «is»]. Следовательно, последняя подстрока совпадает с образцом, поэтому возвращается значение 3 (индекс символа «i»).

Все проверяемые подстроки для функции *findLast*: [«ry», «tr», «nt», …, «is»]. Следовательно, последняя подстрока совпадает с образцом, поэтому возвращается значение 31.

Функция *findFirst*:

1) Прототип: *(str: string, substr: string) => number*;

2) Предусловие: функция не может быть вызвана, если в качестве первого аргумента передана пустая строка;

3) Постусловие: после завершения работы функции ни один из параметров не меняет своего значения.

Функция *findLast*:

1) Прототип: *(str: string, substr: string) => number*;

2) Предусловие: функция не может быть вызвана, если в качестве первого аргумента передана пустая строка;

3) Постусловие: после завершения работы функции ни один из параметров не меняет своего значения.

3. Коды функций, реализующих алгоритмы

Код функции *findFirst:*

```
/**
 * Возвращает индекс первого вхождения подстроки. Если такой подстроки нет, то
возвращается отрицательное значение
 * @param str           строка, в которой происходит поиск
 * @param substr        подстрока
 */
const findFirst = (str: string, substr: string): number => {
  const words: Array<string> = str.split(" ");

  for (let i: number = 0; i < words.length; ++i) {
    if (words[i] === substr) {
      return i;
    }
  }

  return -1;
}
```

Код функции *findLast*:

```
/**
 * Возвращает индекс последнего вхождения подстроки. Если такой подстроки нет,
то возвращается отрицательное значение
 * @param str          строка, в которой происходит поиск
 * @param substr       подстрока
 */
const findLast = (str: string, substr: string): number => {
  const words: Array<string> = str.split(" ");

  for (let i: number = words.length - 1; i >= 0; --i) {
    if (words[i] === substr) {
      return i;
    }
  }

  return -1;
}
```

Код функции *task1*:

```
/**
 * Удаляет слова в предложении, встретившиеся более одного раза
 * @param str       предложение
 */
export const task1 = (str: string): string => {
  return str
    .split(" ")
    .filter((word: string) => findFirst(str.toLowerCase(), word.toLowerCase())
=== findLast(str.toLowerCase(), word.toLowerCase()))
    .join(" ");
}
```

2. Таблица тестов

Ниже представлена таблица тестов для функций *findFirst*, *findLast* (табл.1):

Таблица 1

Входные данные	**Ожидаемые выходные данные для *findFirst***	**Ожидаемые выходные данные для *findLast***	**Фактические выходные данные для *findFirst***	**Фактические выходные данные для *findLast***
Предложение: «He is from Russia which is a cold country» Образец: «is»	3	31	3	31
Предложение: «He is from Russia which is a cold country» Образец: «hi»	-1	-1	-1	-1

Продолжение таблицы 1

Входные данные	**Ожидаемые выходные данные для *findFirst***	**Ожидаемые выходные данные для *findLast***	**Фактические выходные данные для *findFirst***	**Фактические выходные данные для *findLast***
Предложение: «Hello world using the best PL in the world» Образец: «world»	6	37	6	37
Предложение: «Hello world using the best PL in the world» Образец: «best»	4	4	4	4

3. Программа тестирования

Ниже представлен код программы тестирования алгоритма:

```
import * as fs from "fs";

import { task1 } from "./includes/tasks";
import { _INPUT_FILE_NAME, _PATH_TO_INPUT_FILE_ } from "./constants";

// Подобие стандартного потока ввода
const _STDIN_: string = fs.readFileSync(_PATH_TO_INPUT_FILE_ + _INPUT_FILE_NAME,
"utf-8");


/**
 * Основная функция
 */
const main = async (): Promise<void> => {
  const sentence: string = _STDIN_;

  console.log(`Изначальное предложение: ${sentence}`);
  console.log(`После удалений: ${task1(sentence)}`);
}


/* Вызов основной функции с отловом ошибок */
main()
  .catch((err: Error) => {
    console.log(`[ERROR]: ${err.message}`);
  });
```

Программа считывает из файла предложение. Затем применяет функцию удаления из предложения слов, которые входят в него более одного раза и выводит результат в консоль.

4. Практическая сложность алгоритма

Ниже представлена таблица, показывающая зависимость времени выполнения программы от длины предложения *n* (табл.2).

Таблица 2

n	**T**
100	9,762 мс
1000	22,945 мс
10000	360,335 мс

ОТЧЁТ ПО ЗАДАНИЮ 2

1. Разработка программы

1. Постановка задачи

Дано предложение, состоящее из слов, разделённых одним пробелом. Удалить из предложения все вхождения заданного слова, применяя для поиска слова в тексте метод Кнута-Мориса-Пратта.

2. Подход к решению и алгоритмы

Для начала необходимо определить две функции. Пусть первая функция называется *prefixFunction*, а вторая *knuthMorisPratt*.

Суть алгоритма Кнута-Мориса-Пратта заключается в использовании префикс-функции, которая возвращает массив максимальных длин префиксов.

Трюк состоит в том, что на вход этой функции подаётся не просто строка, а объединение образца и строки, которые разделены каким-либо символом, которого нет ни в образце, ни в строке.

Таким образом, после формирования префикс-функцией массива остаётся только пройтись по этому массиву и найти ту длину префикса, которая совпадает с длиной образца.

Однако на практике, можно не объединять образец со строкой, а передать их в качестве параметров отдельно.

Разберём алгоритм на примере. Пусть строка - «aabaabaaaabaabaaab», а образец - «aabaa».

Сначала с помощью префикс-функции заполняется массив [1, 2, 3, 4, 5, 3, 4, 5, 2, 2, 3, 4, 5, 3, 4, 5, 2, 2, 3].

Посмотрим на позиции 4, 7, 12, 15. Значения в массиве равны 5, это значит, что суффикс длиной 5 совпадает с 5 символами префикса. А 5 символов префикса — это и есть образец для поиска.

Функция *prefixFunction*:

1) Прототип: *(str: string) => Array<number>*;

2) Предусловие: если в функцию передать в качествве аргумента пустую строку, то будет возвращён пустой массив;

3) Постусловие: после вызова функции параметр *str* не меняет своего значения.

Функция *knuthMorisPratt*:

1) Прототип: *(str: string, substr: string) => number*;

2) Предусловие: функция в принципе не имеет никаких ограниченний и будет работать стабильно при любых входных данных;

3) Постусловие: на выходе может быть только положительное целое число (в случае удачного поиска), либо число -1.

3. Коды функций, реализующих алгоритмы

Код функции *prefixFunction*:

```
/**
 * Префикс-функция для алгоритма Кнута-Мориса-Пратта
 * @param str
 */
const prefixFunction = (str: string): Array<number> => {
  const N: number = str.length;
  let prefix: Array<number> = new Array(N);

  for (let i: number = 1; i < N; ++i) {
    let prevPrefix: number = prefix[i - 1];

    while ((prevPrefix > 0) && (str[i] !== str[prevPrefix])) {
      prevPrefix = prefix[prevPrefix - 1];
    }

    if (i === prevPrefix) {
      ++prevPrefix;
    }

    prefix[i] = prevPrefix;
  }

  return prefix;
}
```

Код функции *knuthMorisPratt*:

```
/**
 * Реализует алгоритм Кнута-Мориса-Пратта
 * @param str           строка, в которой происходит поиск
 * @param substr        подстрока
 */
const knuthMorisPratt = (str: string, substr: string): number => {
  let k: number = 0;
  let index: number = -1;
  let prefixArray: Array<number> = prefixFunction(substr);

  for (let i: number = 0; i < str.length; ++i) {
    while (k > 0 && substr[k] !== str[i]) {
      k = prefixArray[k - 1];
    }

    if (substr[k] === str[i]) {
      ++k;
    }

    if (k === substr.length) {
      index = i - substr.length + 1;
      break;
    }
  }
```

```
  return index;
}
```

Код функции *task2*:

```
/**
 * Удаляет из предложения str все вхождения слова word, используя для поиска
слова в тексте алгоритм Кнута-Мориса-Пратта
 * @param str           строка, в которой всё происходит
 * @param word          удаляемое слово
 */
export const task2 = (str: string, word: string): string => {
  for (let i: number = 0; i < str.length; ++i) {
    const result: number = knuthMorisPratt(str, word);

    if (result > 0) {
      str = str.slice(0, result) + str.slice(result + word.length, str.length);
      continue;
    }
  }

  return str;
}
```

2. Таблица тестов

Ниже представлена таблица тестов для функции *knuthMorisPratt* (табл.3):

Таблица 3

Входные данные	**Ожидаемые выходные данные**	**Фактические выходные данные**
Строка: «aabaabaaaabaabaaab» Образец: «aabaa»	0	0
Строка: «aabaabaaaabaabaaab» Образец: «ссс»	-1	-1
Строка: «there`s a fire starting in my heart» Образец: «a»	8	8

3. Программа тестирования

Ниже представлен код программы тестирования функции.

```
import * as fs from "fs";

import { task2 } from "./includes/tasks";
import { _INPUT_FILE_NAME, _PATH_TO_INPUT_FILE_ } from "./constants";

// Подобие стандартного потока ввода
const _STDIN_: string = fs.readFileSync(_PATH_TO_INPUT_FILE_ + _INPUT_FILE_NAME,
"utf-8");
```

```
/**
 * Основная функция
 */
const main = async (): Promise<void> => {
  const [sentence, word] = _STDIN_.split("\n");

  console.log(`Изначальное предложение: ${sentence}`);
  console.log(`После удалений: ${task2(sentence, word)}`);
}


/* Вызов основной функции с отловом ошибок */
main()
  .catch((err: Error) => {
    console.log(`[ERROR]: ${err.message}`);
  });
```

Программа считывает из файла предложение. Затем применяет функцию удаления из предложения всех вхождений заданного слова и выводит результат в консоль.

4. Практическая сложность алгоритма

Ниже представлена таблица, показывающая зависимость времени выполнения программы от длины предложения *n* (табл.4).

Таблица 4

n	**T**
100	0,573 мс
1000	0,641 мс
10000	0,886 мс

ОТЧЁТ ПО ЗАДАНИЮ 3

1. Разработка программы

1. Постановка задачи

Посчитать количество различных подстрок у строки длинной *n = 1000*. Это обычный бор. Решить ту же задачу на сжатом боре. Оценить практическую сложность. Сравнить с асимптотической сложностью по бору.

2. Подход к решению и алгоритмы

К сожалению для человека, решающего эту задачу, задание не представляется возможным решить с помощью бора. Поэтому решим задачу с помощью комбинаторики.

Алгоритм достаточно простой. Пусть *n* — длина строки. Для начала нужно посчитать количество различных подстрок длины 1. Очевидно, таких будет *n* штук. Затем посчитаем количество различных подстрок длины 2, 3 и т.д.

Для решения задачи нужно знать формулу числа размещений из *n* элементов по *k*. С помощью данной формулы можно посчитать количество комбинаций из *k* элементов, составленных из *n* элементов. При этом порядок элементов в комбинации имеет значение.

Формула имеет следующий вид (формула 1):

$$A_n^k = \frac{n!}{(n-k)!} \quad (1)$$

Соответственно, чтобы посчитать количество различных подстрок длины *k* для строки длинной *n*, нужно применить формулу 1.

Следовательно, в цикле для *k* от 1 до длины строки *n* включительно к счётчику нужно прибавить число размещений из *n* по *k*. Вот и весь алгоритм.

3. Коды функций, реализующих алгоритмы

Код функции *factorial*:

```
/**
 * Считает факториал для заданного числа
 * @param n           число, факториал которого нужно вычислить
 */
const factorial = (n: number): bigint => {
  let fact: bigint = 1n;

  for (let i: number = 2; i <= n; ++i) {
    fact *= BigInt(i);
  }

  return fact;
}
```

Код функции *countPlacements*:

```
/**
```

```
 * Подсчитывает число размещений из n элементов по k
 * @param n         количество элементов во множестве
 * @param k         количество элементов в комбинации
 */
const countPlacements = (n: number, k: number): bigint => {
  return factorial(n) / factorial(n - k);
}
```

Код функции *task3*:

```
/**
 * Возвращает количество различных подстрок строки длинной length
 * @param length        количество символов в строке
 */
export const task3 = (length: number): bigint => {
  let countSubStrings: bigint = BigInt(length);

  for (let k: number = 2; k <= length; ++k) {
    countSubStrings += countPlacements(length, k);
  }

  return countSubStrings;
}
```

Стоит отметить, что здесь используется тип данных *bigint*, который предназначен для работы с большими числами.

2. Таблица тестов

Ниже представлена таблица тестов для функции *task3*, которая подсчитывает количество различных подстрок для строки длинной *n* (табл.5):

Таблица 5

Входные данные	**Ожидаемые выходные данные**	**Фактические выходные данные**
n = 3	15	15
n = 5	325	325
n = 100	25368695556012729741527074821228022044514757856629814223277518598744925390838644651894048542515204979326740773232800349360951349984969417670976449032316399200	25368695556012729741527074821228022044514757856629814223277518598744925390838644651894048542515204979326740773232800349360951349984969417670976449032316399200

Для *n = 3*, существует 15 различных подстрок. Действительно, если есть строка «abc». То у неё будут следующие подстроки: «a», «b», «c», «ab», «ac», «bc», «ba», «ca», «cb», «abc», «acb», «bac», «bca», «cab», «cba». Всего 15 штук.

Для *n = 1000* результат оказался настолько большим, что число не умещается в пределы одной страницы. Поэтому результат не был включён в таблицу.

3. Программа тестирования

Ниже представлена программа тестирования функции *task3*:

```
import * as fs from "fs";

import { task3 } from "./includes/tasks";
import { _INPUT_FILE_NAME, _PATH_TO_INPUT_FILE_ } from "./constants";

// Подобие стандартного потока ввода
const _STDIN_: string = fs.readFileSync(_PATH_TO_INPUT_FILE_ + _INPUT_FILE_NAME,
"utf-8");


/**
 * Основная функция
 */
const main = async (): Promise<void> => {
  const strLength: number = +_STDIN_;

  console.time("counting");
  const subStrings: bigint = task3(strLength);
  console.timeEnd("counting");

  console.log(`Для строки длинной ${strLength} существует ${subStrings}
различных подстрок`);
}


/* Вызов основной функции с отловом ошибок */
main()
  .catch((err: Error) => {
    console.log(`[ERROR]: ${err.message}`);
  });
```

4. Практическая сложность алгоритма

Ниже представлена таблица зависимости времени исполнения программы от объёма входных данных (табл.6):

Таблица 6

n	**T**
10	0,167 мс
100	4,544 мс
1000	966,054 мс

ОТВЕТЫ НА ВОПРОСЫ

1. Что называют строкой?

Строка — тип данных, значениями которого является произвольная последовательность символов алфавита.

2. Что называют префиксом строки?

Префикс строки — подстрока определённой длины, которая начинается с начала исходной строки.

3. Что называют суффиксом строки?

4. Асимптотическая сложность последовательного поиска подстроки в строке.

Асимптотическая сложность последовательного поиска подстроки в строке — $O(N*M)$, где N — длина исходной строки, M — длина образца.

5. В чём особенность поиска образца алгоритмом Бойера-Мура?

Особенность поиска образца алгоритмом Бойера-Мура является тот факт, что сравнение символов начинается с конца образца, а не с начала, то есть сравнение отдельных символов происходит справа налево.

Суффикс строки — подстрока определённой длины, которая заканчивается на последнем символе исходной строки.

6. Приведите асимптотическую сложность алгоритма Бойера-Мура поиска подстроки в строке по времени и памяти.

Асимптотическая сложность алгоритма Бойера-Мура — $O(N+M)$, где N — длина исходной строки, M — длина образца.

Асимптотическая сложность алгоритма Бойера-Мура по памяти — $O(N+M+P)$, где P — длина таблицы сдвигов.

7. Приведите пример входных данных для реализации эффективного метода прямого поиска подстроки в строке.

Исходная строка: «aabcdf». Образец: «a». Тогда первая же проверка найдёт нужную подстроку.

8. Приведите пример строки, для которой поиск подстроки «aaabaaa» будет более эффективным, если делать его методом Кнута, Морриса и Пратта, чем, если делать его методом Бойера-Мура. И наоборот.

9. Объясните, как влияет размер таблицы кодов в алгоритме Бойера-Мура на скорость поиска.

Алгоритм поиска Бойера-Мура наиболее эффективен, если образец длинный, а мощность алфавита (таблицы кодов) достаточно велика. То есть, чем больше размер таблицы кодов в алгоритме, тем выше скорость работы алгоритма.

10. За счёт чего в алгоритме Бойера-Мура поиск оптимален в большинстве случаев?

В большинстве случаев поиск оптимален, потому что при несовпадении символов выполняется сдвиг по тексту на установленное значение.

11. Поясните влияние префикс-функции в алгоритме Кнута, Морриса и Пратта (КМП) на организацию поиска подстроки в строке.

Префикс-функция в алгоритме Кнута-Морриса-Пратта нужна для вычисления смещений для каждого символа в подстроке. Следовательно, когда длина суффикса совпадает с длиной префикса подстроки, вхождение образца в строку найдено.

12. Приведите пример префикс-функции для поиска образца в тексте для алгоритма КМП.

Пусть исходая строка «abcabcd», а образец - «ab». Соответственно, формируем строку «ab@cdabcd». То есть только, что произошло склеивание образца с исходной строкой. Ниже представлен массив, который возвращается префикс-функцией (табл.7).

Таблица 7

a	**b**	**@**	**c**	**d**	**a**	**b**	**c**	**d**
0	0	0	0	0	1	2	0	0

13. В чём особенность поиска образца алгоритмом Рабина и Карпа?

Особенность поиска образца алгоритмом Рабина и Карпа состоит в том, что для поиска используется хеширование, хеш-функции.

14. Приведите асимптотическую сложность алгоритма Рабина и Карпа поиска подстроки в строке.

Эффективность в лучшем случае — *O(n)*, а в худшем случае — *O(n*m)*.

15. Что такое бор?

Бор представляет собой т-арное дерево. Каждый узел уровня *h* — это множество всех ключей, начинающихся с определённой последовательности из *h* символов.

16. Какие структуры хранения данных используются для реализации простого бора?

Для реализации простого бора нужно создать собственную структуру данных, которая содержит поле массива из символов. Таким образом, можно реализовать структуру дерева.

17. Приведите пример бора и реализуйте его одним из способов. Объясните алгоритм поиска образца с использованием бора.

18. Поясните применение алгоритма Ахо-Корасик. Приведите его вычислительную и ёмкостную сложность.

Задача алгоритма Ахо-Корасик состоит в том, чтобы построить бор и из него построить автомат. Полученный автомат уже может использоваться в различных задачах, простейшая из которых — нахождение всех вхождение каждой строки из данного набора в некоторый текст за линейное время.

Вычислительная сложность — *O(m)*, где *m* — длина исходной строки. Ёмкостная сложность — *O(m*k)*, где *k* — длина подстрок.