Оглавление

[Тема 1](#_Toc28587170)

[Преамбула 1](#_Toc28587171)

[Задачи 1](#_Toc28587172)

[Отчет 4](#_Toc28587173)

# Тема

Разработка алгоритма упрощенного алгоритма Бойера-Мура (Бойера-Мура-Хорспула). Сравнение со стандартным find. Исследование зависимости сложности от алфавита, длины образца и текста.

# Преамбула

В этом задании вы разработаете функцию поиска подстроки в тексте упрощенным алгоритмом Бойера-Мура, рассмотренном в лекции 3 с модификацией в духе Чжу-Такаоки.

Упрощенный алгоритм использует только правило плохого символа. А модификация Чжу-Такаоки заключается в использовании пары символов вместо одного. Это существенно улучшает эффективность на коротких алфавитах.

Проведете исследование быстродействия алгоритма.

Прежде чем приступать к выполнению задания, необходимо прочесть часть лекции 3 относительно алгоритма Бойера-Мура (BM).

# Задачи

1. Начните новый проект SimpleBMХZT. Подключите к нему вывод-вывод и модуль *list*.
2. Добавьте в проект класс BadCharTbl. Его удобно сделать inline, т.е. полностью определить в заголовочном файле. Для этого, если вы пользовались мастером добавления классов, нужно или указать, что класс будет inline или удалить из проекта файл BadCharTbl.cpp, а непосредственно в BadCharTbl.h реализовать пока пустой конструктор.
3. Пусть нашим алфавитом будут все ASCII символы, коды которых меньше 128. Тогда в классе BadCharTbl понадобится лишь массив целых размером 128\*128. Добавьте соответствующее (*массиву*) поле с именем badChar.

Других данных в классе не предвидится. Что можно сказать о конструкторах и деструкторах? Доработайте класс в соответствие с вашими соображениями.

1. Массив будет заполняться в методе void Init(const char \*s). Добавьте его. Пусть метод пока просто инициализирует массив badChar значениями len - 1, где len – длина s. (*Почему именно им? Что в лекции?*)

Создайте объект типа BadCharTbl в main и вызовите Init. Убедитесь, что проект компилируется и выполняется.

1. Добавьте в класс оператор преобразования к целому, который просто возвращает длину массива, и оператор [], который позволит читать значение таблицы для определенной пары символов (int operator[] (int c) const). Он должен просто возвращать значение массива с индексом, заданным параметром c.
2. Вернитесь к методу Init. Выясните из лекции, как заполняется таблица стоп-символов. Добавьте соответствующий комментарий перед Init.  
     
   Реализуйте соответствующий алгоритм в Init после цикла инициализации. (см. лекцию 3).

Здесь индексами будут пары символов. Превратить их в целое число можно, например, так \*(unsigned short\*)p, где p указывает на первый символ!

В badChar достаточно изменить только те значения, которые соответствуют имеющимся парам. Последняя пара не учитывается!

1. В main проинициализируйте объект таблицы плохих символов строкой “**karakara**”, выведите в цикле все значения таблицы, которые не равны len - 1. Это можно сделать таким оператором:

ctbl.Init("karakаra");

for (int i = 0; i < ctbl; i++)

if (ctbl[i] != len-1)

cout << (char)i<<(char)(i>>8) << '\t' << ctbl[i] << endl;

Должны появиться все пары соседних символов, которые встречаются в образце и их смещение от конца строки, кроме последней пары!

1. Займемся упрощенным алгоритмом Бойера-Мура-Хорспула с модификацие в духе Чжу-Такаоки. Реализуйте его в виде функции   
   list <int> BMHZT (const char\*t, const char\*T). Функция будет возвращать список найденных вхождений t в T.

Здесь используется шаблон списка STL. Он определяется в модуле <list>. Соответственно, объявить переменную типа “список целых” можно так:

list<int> r;

Добавлять элементы в список можно так:

r.push\_back(x);

Пробежаться по списку можно так:

for(int e: r)

cout<< e << ‘ ’;

cout<<endl;

* 1. Объявите в BMHZT список целых *result*. В нем буду содержаться результаты поиска;
  2. Объявите объект BadCharTbl и проинициализируйте его строкой t;
  3. Реализуйте алгоритм Бойера-Мура согласно лекции, но используйте в нем лишь таблицу плохих символов!

**Важно!** В качестве “плохого символа” в БМХЧТ используется пара символов строки T, которая совмещена с окончанием шаблона t. Именно поэтому “эвристика плохих символов” всегда работает! Это приводит к тому, что сдвиг вычисляется иначе, чем указано в лекции. Проявите осторожность!

В случае нахождения t добавляйте в result индекс в строке T;

* 1. Верните список result как результат работы функции.

1. Пора проверить наш алгоритм. Для этого придется научиться генерировать случайные строки. Это можно сделать по следующему примеру.

const int N = 100;

const int alphaBetSize = 'z' - 'a' + 1;

char \*s = new char[N + 1]{0};

for (int i = 0; i < N; i++)

s[i] = rand() % alphaBetSize + 'a';

Здесь сгенерирована строка длиной 100 из алфавита ‘a’…’z’.

* 1. Объявите 2 целочисленных константы N – длина T и M – длина t.
  2. Сгенерируйте соответствующие строки. Вначале алфавит можно ограничить символами ‘a’ и ’b’ (*так будет проще отлаживать программу*). N можно взять 50, а M = 3 или 4.
  3. Вызовите в main BMHZT, а результат ее работы (*позиции вхождения t в T*) выведите на консоль в одну строку.
  4. Проверьте корректность поиска, используя, например, notepad.

1. В дальнейшем для переключения режимов работы программы будет удобно использовать препроцессор:
   1. Перед вызовом BMHZT поставьте инструкцию препроцессора  
       #ifdef \_BM\_
   2. После вызова BMHZT  
       #else  
       #endif
   3. Добейтесь правильности работы программы. Скорее всего, ошибки будут связаны с тем, что список, в который получен результат работы BMHZT, сохраняется в переменной, объявленной под #ifdef.

В таком случае достаточно объявить аналогичный список между #else и #endif. Так, вне зависимости от того, определена \_BM\_ или нет будет объявлена переменная списка.

1. Теперь нам предстоит повторить поиск, но уже обычным методом класса string – find:
   1. Между #else и #endif объявите строку sT и проинициализируйте ее массивом T;
   2. Реализуйте поиск всех вхождений t в sT, используя метод find класса string.

Метод s.find(t, off) возвращает индекс вхождения строки t в строку s, начиная со смещения off. Если таких вхождений нет, то -1. Очевидно, что для поиска всех вхождений, если обнаружено вхождение в j, то необходимо продолжить поиск со смещением j+1.

Индексы вхождений добавляйте в список результатов.

* 1. Убедитесь, что поиск выполняется корректно.

1. Давайте воспользуемся удобствами препроцессора. Добавьте перед main определение   
   #define \_BM\_  
   и снова выполните программу.

Теперь поиск выполнялся с помощью BMHZT. Таким образом, для переключения метода поиска нам достаточно закомментировать или раскомментировать определение   
#define \_BM\_.

Убедитесь, что оба метода поиска дают одинаковый результат!

1. Прежде чем оценивать эффективность нашего алгоритма, давайте его немного оптимизируем.
   1. Если в нескольких местах кода используется strlen от одной и той же строки, вычислите ее длину 1 раз и запомните в переменной;
   2. Обязательно убедитесь, что после оптимизации сдвиги не изменились и результат остался таким же, как и до.
2. Теперь нам необходимо научиться измерять время выполнения программы.
   1. Включите заголовочный файл <chrono>;
   2. Объявите перед измеряемым участком (*перед #ifdef* ) переменную

auto start = chrono::steady\_clock::now();

* 1. Сразу за кодом поиска, еще до вывода результатов на консоль (*сразу после #endif*) объявите  
     auto stop = chrono::steady\_clock::now();
  2. Теперь время в микросекундах можно вычислить следующим образом:  
     auto dt = chrono::duration\_cast<chrono:: microseconds>(stop - start).count();  
     можно также переводить в nanoseconds, milliseconds, microseconds, seconds, minutes и hours.

1. Проверьте, что в области измерения находится ТОЛЬКО КОД ПОИСКА, нет никакого вывода на консоль (*это ОЧЕНЬ медленно и исказит результат*) в методах и пр. Проверьте код main на предмет утечек памяти и внесите необходимые изменения.
2. **Пора приступать к исследованиям**. Убедитесь, что программа компилируется в Release моде. Скорее всего, компиляция для x86 будет эффективнее чем для x64.
3. Выберите M = 6, а N в диапазоне от 128 до 2^13, каждый раз увеличивая N в 4 раза. Алфавит при случайном построении строк не должен быть большим. Типично, это 4-5 символов, иначе поиск будет, как правило, пустым. Измеряйте время поиска для каждого значения N одним и вторым методом.

При измерении времени нужно иметь ввиду, что Windows – многозадачная среда. ОС во время засечки может передавать CPU другим процессам. Поэтому, для корректного измерения нужно выполнять программу для каждой конфигурации 3-5 раз и выбирать наименьшее из полученных значений.

1. Проанализируйте, как ведет себя тот и другой метод поиска в зависимости от длины **T**.
2. Исследуйте зависимость времени от длины **t** при максимальном фиксированном **T** (N=2^13).
3. Исследуйте зависимость времени поиска от размера алфавита (2-4-8-16-28) при фиксированных длинах **t** (M=64) и **T**(N=2^13).

# Отчет

В отчете кратко опишите результаты ваших исследований. Отмечайте только суть. Общий объем не должен превышать 1 стр.

Очистите решение от вспомогательных файлов и результатов компиляции и поместите каталог в архив.

Архив с решением и отчет в Word отправляйте как ответ на выполнение задания.