ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

А.Ю. Поляков А.Ю. Полякова Е.Н. Перышкова

ПРОГРАММИРОВАНИЕ

Практикум

Новосибирск
2015

УДК 004.4'4
ББК 32.973.26-018.2

Поляков А.Ю., Полякова А.Ю., Перышкова Е.Н. Программирование: Практикум. – Новосибирск/СибГУТИ, 2015. –54 с.

Практикум предназначены для студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника», профиль «Вычислительные машины, комплексы, системы и сети», изучающих дисциплины «Программирование» и «Языки программирования». Практикум содержит теоретический материал и набор практических заданий, предназначенных для выполнения курсовых работ по указанным дисциплинам с целью получения практических навыков в создании программ с использованием высокоуровневых методов программирования на языке С.

Кафедра вычислительных систем

Ил. – 28, табл. – 3, список лит. – 9 наим.

Рецензент: доцент кафедры прикладной математики и кибернетики ФГОБУ ВПО «СибГУТИ» доцент Галкина М.Ю.

Для студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» (профиль «Вычислительные машины, комплексы, системы и сети»), 02.03.02. Фундаментальная информатика и информационные технологии (профиль Супервычисления)

Утверждено редакционно-издательским советом ФГОБУ ВПО «СибГУТИ» в качестве практикума.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Дисциплины «Программирование» и «Языки программирования» предусматривают выполнение курсовой работы, включающей в себя проектирование, кодирование и тестирование программы, а также написание подробного отчета к ней.

Данные методические указания содержат теоретические сведения, необходимые для выполнения курсовой работы, а также варианты заданий. Также указаны требования к разрабатываемым программам и содержанию отчета.

ЦЕЛЬ И ЗАДАЧИ КУРСОВОЙ РАБОТЫ

Цель курсовой работы – получить практические навыки в создании новых типов данных и использовании современных образцов программирования при подготовке программных продуктов.

Порядок выполнения работы:

1. Изучить методические указания.
2. Познакомиться с предполагаемой структурой и содержанием курсовой работы.
3. Познакомиться с графиком выполнения курсовой работы.
4. Познакомиться с требованиями оформления отчета для курсовой работы.
5. Выполнить последовательно все этапы работы, предусмотренные поставленным заданием.
6. Распечатать отчет и сдать его для проверки.
7. Защитить работу преподавателю.

ТРЕБОВАНИЯ К КУРСОВЫМ РАБОТАМ

1. Все программы реализуются на языке С.

2. Необходима проверка всех входных данных на корректность. В случае ошибки должно выдаваться сообщение, поясняющее ее суть для пользователя. Далее программа должна завершаться с ненулевым кодом.

3. Для сдачи курсовой работы необходимо подготовить отчет, включающий приведенные ниже пункты:

   3.1. Титульный лист, оформленный согласно шаблону, размещенному на WEB-странице предмета. В поле название пишется название раздела, к которому относится вариант;

   3.2. Задание — содержит текст задания;

   3.3. Анализ задачи. Приводится развернутый анализ задачи, описываются методы и алгоритмы ее решения (особенно важные фрагменты представляются блок-схемами или псевдокодом). Если используются известные алгоритмы (кодирования, архивации, поиска) для них также необходимо привести описание и демонстрацию работы на примере;

   3.4. Тестовые данные: максимально полный набор тестовых данных, демонстрирующий работу разработанного ПО на различных данных (в том числе некорректных).

   3.5. Листинг программы. Раздел содержит исходные коды разработанного ПО. При оформлении можно использовать размер шрифта 8 пт.

ГЛАВА 1
ОБРАБОТКА ПОСЛЕДОВАТЕЛЬНОЙ ИНФОРМАЦИИ

В рамках данной главы рассматриваются различные задачи обработки последовательностей.

Получение информации через аргументы командной строки

В большинстве заданий входные данные должны передаваться через аргументы командной строки. Эта информация является частью «окружения» программы. Все аргументы командной строки передаются в виде строк и доступны через формальные параметры функции ***main***. Прототип функции ***main*** в программе, использующей аргументы командной строки, выглядит следующим образом:

```
int main(int argc, char **argv)
```

где *argc* – количество строк-аргументов, а *argv* – массив указателей на сами строки-аргументы.

Рассмотрим следующий пример:

```
$ ./prog first_arg “second arg” third arg
```

На вход программе будет передана информация, показанная на рисунке 1.

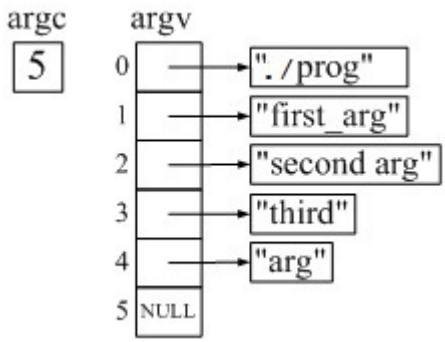


Рисунок 1. Аргументы командной строки

Обратите внимание, что в командной строке пробел является разделителем, поэтому строка third arg разбивается на два аргумента. Если требуется, чтобы входной аргумент содержал пробелы – строку необходимо заключать в кавычки, например, *"second arg"*. Первым элементом *argv* (*argv*[*0*]) всегда является имя исполняемого файла программы. Обработка входных аргументов может производиться напрямую через взаимодействие с массивом *argv*, однако существует ряд функций, позволяющих упростить процесс разбора входных параметров. Это функции ***getopt*** и ***getopt_long***. Более подробную информацию об использовании этих функций можно получить в справочном руководстве ОС GNU/Linux – "*man 3 getopt*", данная страница доступна на русском языке на сайте проекта opennet.ru:

http://www.opennet.ru/man.shtml?topic=getopt&category=3&russian=0.

Также примеры использования указанных функций есть на сайте FirstSteps: *http://www.firststeps.ru/linux/r.php?10.*

ВАРИАНТЫ 1.1-1.3 Разработка библиотеки сортировок

Задание

Реализовать динамическую библиотеку сортировок. Алгоритмы сортировок выбираются в соответствии с вариантом задания. Проанализировать эффективность алгоритмов сортировки. Разработать демонстрационную программу, использующую созданную библиотеку.

Критерии оценки

- **Оценка «удовлетворительно»**: алгоритмы реализованы в виде простой программы без применения библиотек, данные поступают на вход с клавиатуры. Тесты проведены только на небольших последовательностях, нет анализа и сравнения алгоритмов. Не предусмотрено динамическое выделение памяти под входные данные.
- **Оценка «хорошо»**: работа выполнена в полном соответствии с заданием. Обязательно динамическое выделение памяти под входные данные.
- **Оценка «отлично»**: помимо выполнения условий задания предусмотрена сортировка произвольных данных (по аналогии с функцией ***qsort*** библиотеки GNU C Library – GLibC). Обязательно динамическое выделение памяти под входные данные.

Указания к выполнению задания

Алгоритмы сортировки необходимо реализовать в подпрограммах. Подпрограммы выносятся в отдельную библиотеку, которая компилируется как динамическая. Информация о создании и использовании динамических библиотек может быть найдена на ресурсе FirstSteps: *http://firststeps.ru/linux/general1.html.*

Эффективность сортировок оценивать по времени работы алгоритмов. По полученным результатам сформулировать выводы о преимуществах и недостатках каждого алгоритма. Сравнить полученные результаты с теоретическими оценками вычислительной сложности реализованных алгоритмов.

Экспериментальные измерения необходимо провести как для упорядоченных данных (по возрастанию и по убыванию), так и случайных последовательностей, размер которых составляет $2^8 – 2^{15}$ элементов (с некоторым шагом). Построить графики полученных зависимостей. В случае, если время работы одного из алгоритмов превышает 15 мин., прекратить измерения по данному алгоритму и строить график не на всем интервале.

Таблица 1. Распределение вариантов задания по видам сортировок.

Вариант	**Алгоритмы сортировок**
1.1	Сортировка методом пузырька, быстрая сортировка
1.2	Сортировка вставками, пирамидальная сортировка.
1.3	Сортировка Шелла, сортировка слиянием.

ВАРИАНТ 1.4 Поикс палиндромов в тексте

Задание

Разработать программу ***palindrom***, выполняющую поиск всех палиндромов в заданном тексте. Команда ***palindrom*** принимает в качестве аргумента командной строки имя файла, содержащего текст на русском языке. Все найденные палиндромы распечатываются на экране.

Критерии оценки

- **Оценка «удовлетворительно»**: реализована проверка того, что весь текст входного файла целиком является палиндромом. Не предусмотрено динамическое выделение памяти под входные данные.
- **Оценка «хорошо»**: реализована предварительная обработка текста: из каждого предложения удаляются все знаки препинания. После этого осуществляется проверка каждого предложения на выполнение свойства палиндрома. Обязательно динамическое выделение памяти под входные данные.
- **Оценка «отлично»**: реализована предварительная обработка текста: из текста удаляются все пробелы и знаки препинания так, чтобы получилось одно большое слово. Для поиска подпалиндромов используется алгоритм, основанный на применении *динамического программирования* (*http://comp-science.narod.ru/WebPage/p6.html*). Обязательно динамическое выделение памяти под входные данные.

Указания к выполнению задания

Палиндромом называются слово (потоп, шалаш) или текст (а роза упала на лапу Азора), читающиеся одинаково в обоих направлениях.

Запуск программы должен производиться со следующими аргументами командной строки:

```
$ palindrom text.txt
```

Программа выполяет поиск всех палиндромов в файле *text.txt* и распечатывает результат работы на экране.

ВАРИАНТ 1.5 Разработка простейшего переводчика

Задание

Разработать программу ***translate***, выполняющую перевод текста с помощью словаря. Команда ***translate*** принимает на вход 3 файла. Первый содержит исходный текст, который необходимо перевести. Второй файл имеет вид простейшего словаря, где каждому слову на исходном языке соответствует слово на целевом. Третий файл необходимо создать и записать в него результат работы переводчика. Формат исходного текста должен быть сохранен.

Критерии оценки

- **Оценка «удовлетворительно»**: не реализована поддержка файла-словаря, словарь задается статически в программе. Не предусмотрено динамическое выделение памяти под входные данные.
- **Оценка «хорошо»**: программа реализована в полном соответствии с заданием. Обязательно динамическое выделение памяти под входные данные.
- **Оценка «отлично»** не предусмотрена, может быть предложен свой вариант усложнения.

Указания к выполнению задания

Запуск программы должен производиться со следующими аргументами командной строки:

```
$ translate text_rus.txt dictionary.txt text_eng.txt
```

Программа должна перевести текст в файле *text_rus.txt* с помощью словаря *dictionary.txt*, и записать результат в файл *text_eng.txt*. Примерное содержимое файлов и результат работы программы представлен на рисунке 2.

Text_rus.txt:

```
Тигр,
     Тигр,
жгучий страх.
Ты горишь в ночных лесах.
  Чей бессмертный взор,
любя,
Создал страшного тебя?
```

Dictionary.txt:

```
Тигр – Tyger
Страх – Fear
Ты – You
Взор – Eye
```

Text_eng.txt:

```
Tyger,
     Tyger,
жгучий fear.
You горишь в ночных лесах.
  Чей бессмертный Eye,
любя,
Создал страшного тебя?
```

Рисунок 2. Пример работы программы

Форматирование текста в файле text_rus.txt должно быть сохранено и в итоговом файле text_eng.txt, т.е. сохранены все сдвиги и переносы по тексту. Задание не предусматривает поиск однокоренных слов, поэтому замена слова происходит только по полному соответствию.

ВАРИАНТ 1.6 Частотный анализ текста

Задание

Разработать программу ***calcFrequency***, производящую частотный анализ слов в исходном тексте. На вход программы ***calcFrequency*** подается 2 файла. Первый файл содержит текст, который подлежит анализу. Второй файл необходимо создать, и записать все слова, встретившиеся в тексте с указанием частоты появления. Провести сортировку слов по частоте встречаемости (в порядке убывания).

Критерии оценки

- **Оценка «удовлетворительно»**: реализован подсчет количества символов и слов в тексте. Не предусмотрено динамическое выделение памяти под входные данные.
- **Оценка «хорошо»**: программа реализована в полном соответствии с заданием. Обязательно динамическое выделение памяти под входные данные.
- **Оценка «отлично»** не предусмотрена, может быть предложен свой вариант усложнения.

Указания к выполнению задания

Запуск программы должен производиться со следующими аргументами командной строки:

```
$ calcFrequency text1.txt text2.txt
```

Программе необходимо проанализировать текст в файле *text1.txt* и записать результаты работы в файл *text2.txt*. Примерное содержимое текстового файла и результат работы программы представлен на рисунке 3.

Text1.txt:	*Text2.txt:*
Кто рождается на свет Лишь для горести и бед, Кто рождается на вечность Лишь для радости беспечной. Кто для радости беспечной, Кто для ночи бесконечной.	Кто – 4 для – 3 рождается – 2 беспечной – 2 на – 2

Рисунок 3. Пример работы программы

Необходимо записывать во второй файл только те слова, частота встречаемости которых больше 1. Склонения и однокоренные слова считать разными словами.

ГЛАВА 2.
РАЗРАБОТКА ИНСТРУМЕНТОВ КОМАНДНОЙ СТРОКИ ОС GNU/LINUX

В заданиях данной главы требуется разработать программные утилиты командной строки в соответствии с вариантом задания. Все входные данные передаются через аргументы командной строки (с использованием аргументов функции ***main***, см. главу 1). Использование системного вызова ***system*** запрещено, допустимо использовать только системные вызовы ОС GNU\Linux.

Взаимодействие с файловой системой

Для выполнения некоторых заданий данного и последующих разделов потребуются инструменты, позволяющие получать информацию о содержимом директорий. В ОС GNU/Linux для решения этой задачи предусмотрены следующие функции:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
int closedir(DIR *dirp);


#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

Функция ***opendir*** "открывает" директорию для чтения, на ее вход передается строка, содержащая путь к директории, возвращаемым значением является указатель на структуру DIR, которая используется для дальнейших действий с открытой директорией.

Функция ***readdir*** выполняет последовательный проход по содержимому открытой директории. При каждом ее вызове она возвращает указатель на структуру типа ***struct dirent***, элементы которой приведены ниже:

```
struct dirent {
  ino_t d_ino;               /* i node number */
  off_t d_off;               /* offset to the next
                            dirent */
  unsigned short d_reclen;   /*length of this record*/
  unsigned char  d_type;     /* type of file; not
                            supported by all file
                            system types */
  char d_name[256];          /* filename */
};
```

Для выполнения заданий курсовой работы интерес представляют поля *d_name* и *d_type*. Первое содержит имя файла или директории, второе – позволяет определить тип элемента файловой системы, это поле представляет из себя целое число, которое устанавливается функцией ***readdir*** в одно из допустимых

значений, которые представлены константами, определенными в файле *dirent.h*:

```
DT_BLK      Блочное устройство
            (например, файл жесткого диска)
DT_CHR      Символьное устройство (клавиатура, мышь)
DT_DIR      Директория
DT_FIFO     Именованные программные каналы
            (для взаимодействия между программами).
DT_LNK      Символьная ссылка на файл
DT_REG      Обычный (регулярный) файл
DT_SOCK     Unix – сокет
            (для взаимодействия между программами).
DT_UNKNOWN  Неизвестный тип файла.
```

В рамках курсовой работы предусмотрена обработка только директорий (*DT_DIR*) и регулярных файлов (*DT_REG*), другие типы файлов должны игнорироваться.

Функция ***closedir*** закрывает директорию, после ее вызова функция ***readdir*** не должна использоваться с данной структурой типа *DIR*.

Дополнительную информацию об объектах файловой системы можно получить, используя функцию ***stat*** (самостоятельное изучение).

ВАРИАНТ 2.1 Интерпретатор скриптов

Задание

Разработать интерпретатор скриптов (ИС) - ***sinterp***. Команда ***sinterp*** принимает в качестве входных данных имя файла, содержащего скрипт.

Доступные операции: =, +, –, >, <, ==. На основе доступных операций допускается построение арифметических (операции +, –) и логических (операции >, <, ==) выражений. Арифметические выражения должны использоваться в операциях присваивания, логические выражения – в циклических конструкциях и конструкциях ветвления.

Операции присваивания допускают использование только одной бинарной операции + или –, например: $i = k + 10$, $i = 5 - x$. Выражения вида $i = k + 10 - x$ не допускаются, для их реализации необходимо сформировать два арифметических выражения: $i = k + 10$, $i = i - x$. Каждое арифметическое выражение записывается с новой строки.

Логические выражения могут содержать только одну из доступных операций сравнения, например: $i > 0$.

Скрипт может содержать следующие языковые конструкции:

1. ввод/вывод данных (**read/print**);
2. цикл (**while – do – done**);
3. ветвление (**if – then – else – fi**)

Критерии оценки

- **Оценка «хорошо»**: работа только с целыми числами, использование конструкций ввода-вывода и циклических конструкций. Вложенность циклических конструкций не предусматривается.
- **Оценка «отлично»**: работа с целыми и вещественными числами (отслеживание корректности использования переменных), должны быть реализованы все предусмотренные конструкции. Должна быть реализована возможность использования вложенных конструкций ветвление и цикл.

Формат доступных конструкций

Конструкции ввода-вывода:

read [тип данного: **-i** или **-f**] <имя переменной>

write <имя переменной>

При работе только с целыми числами спецификаторы формата реализовывать не нужно.

Циклическая конструкция:

while <логическое выражение> **do**

<тело цикла>

done.

Конструкция ветвления:

if <логическое выражение> **then**

<ветвь ветвления>
[**else**
<ветвь иначе>]
fi

Указания к выполнению задания

Запуск программы должен производиться со следующими аргументами командной строки:

```
$ sinterp script.txt
```

Программе необходимо выполнить скрипт, записанный в файл *script.txt* и выдать результат работы на экран. Примерное содержимое скриптов для программы представлено на рисунке 4.

```
Оценка «хорошо»
Вычисление суммы элементов последовательности
sum = 0
read n
i=0
while i < n do
   read x
   sum = sum + x
   i = i + 1
done
```

```
Оценка «отлично»
   Поиск минимального элемента последовательности
   read n
   read min
   i=0
   while i < n do
      read x
      if min > x then
         min = x
      fi
      i = i + 1
done
```

Рисунок 4. Пример скриптов

ВАРИАНТ 2.2 Калькулятор командной строки

Задание

Разработать программу-калькулятор ***cmdcalc***, предназначенную для вычисления простейших арифметических выражений с учетом приоритета операций и расстановки скобок. На вход команды ***cmdcalc*** через аргументы командной строки поступает символьная строка, содержащая арифметическое выражение. Требуется проверить корректность входного выражения (правильность расстановки операндов, операций и скобок) и, если выражение корректно, вычислить его значение.

Арифметическое выражение записывается следующим образом: **`A p B`** или **(`A p B`)**, где `A` – левый операнд, `B` - правый операнд, `p` – арифметическая операция. `A` и `B` – представляют собой арифметические выражения или вещественные числа, `p` = + | - | * | /.

Например: ((((1.1-2) +3) * (2.5*2)) + 10), (1.1 – 2) +3 * (2.5 * 2) + 10. Пример вызова команды ***cmdcalc*** (обратите внимание, что входное выражение необходимо взять в кавычки!):

```
$ cmdcalc "3 * 2 – 1 + 3"
```

Полученный ответ может отображаться на экране, а также сохраняться в файле.

Критерии оценки

- **Оценка «удовлетворительно»**: не предусмотрены скобки и приоритеты операций.
- **Оценка «хорошо»**: учитываются приоритеты операций.
- **Оценка «отлично»**: учитываются приоритеты операций, допускаются скобки.

Указания к выполнению задания

Одним из подходов к решению указанной задачи может быть использование структуры данных «стек». Данная структура представляет собой список элементов организованных по принципу LIFO (англ. last in — first out, «последним пришел — первым вышел»).

Одним из примеров стека может служить детская пирамидка: ось, на которую одеваются кольца. Первым с такой пирамидки снимается кольцо, размещенное на ней последним. Размещение элемента в стеке обозначают операцией ***push***, а извлечение элемента из стека – операцией ***pop***.

Для решения указанной задачи потребуется два стека, первый (***num***) будет использоваться для хранения операндов, тип элемента – вещественное число, второй (***ops***) – для хранения операций и скобок, тип элемента – символ.

Рассмотрим алгоритм на примере выражения 1.1 – (5 +2 * (2 + 3)) / 10.

Пошаговый разбор алгоритма представлен на рисунке 5.

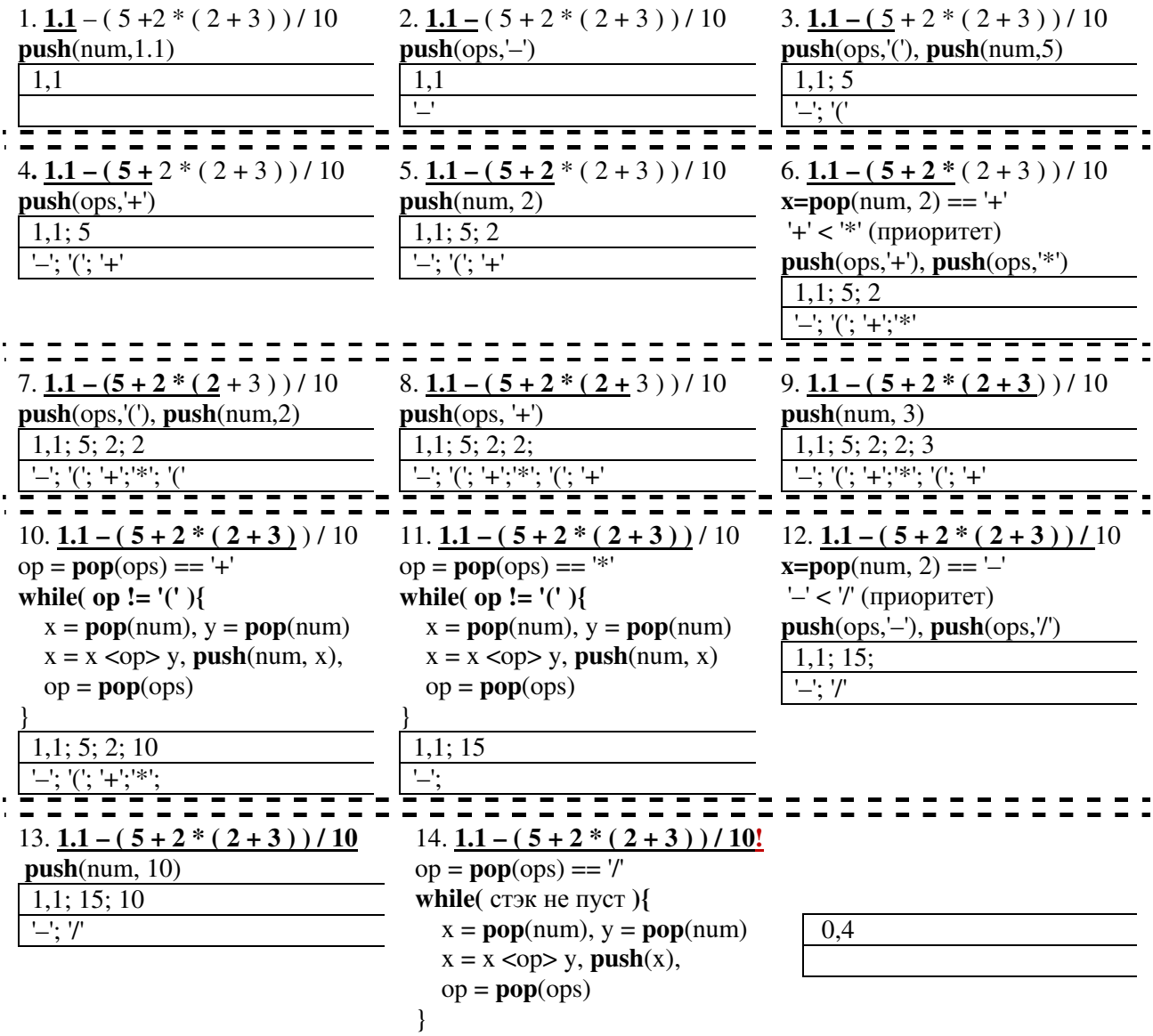


Рисунок 5. Пошаговы й алгоритм использование стека

ВАРИАНТ 2.3 Форматирование исходного кода программы на языке С

Задание

Разработка инструмента ***cformat***, обеспечивающего автоматическое форматирование программ на языке С. Команда ***cformat*** принимает на вход имя файла, содержащего программу на языке С. Содержимое файла должно быть отформатировано согласно требованиям, приведенным ниже. Отформатированная программа не должна терять корректность (компилируемость).

Критерии оценки

- **Оценка «удовлетворительно»**: предусмотрена только расстановка табуляции на основании блоков операторов (количество табуляций равно числу символов '{').
- **Оценка «хорошо»**: расстановка табуляций, изменение положения фигурных скобок и разбиение строк, превышающих максимально допустимую длину (80 символов).
- **Оценка «отлично»**: в дополнение к требованиям оценки «хорошо» должна осуществляется проверка корректности расстановки фигурных, круглых и квадратных скобок (количество и последовательность должны быть правильными).

Указания к выполнению задания

Проверка расстановки и очередность скобок осуществляется при помощи стека.

К форматированию программы устанавливаются следующие требования:

1. Строка должна быть сдвинута на *x* табуляций вправо, где *x* определяется глубиной вложенности блока операторов. Пример преобразования представлен на рисунке 6.

```
int main( ){                    int main( )
int x = 0;                      {
if( x > 0 )                     →int x = 0;
{                         =>    →if( x > 0 ){
x = 1;                          →→x = 1;
}}                              →}
                                }
```

Рисунок 6. Форматирование строк с помощью табуляций

2. Фигурная скобка, открывающая блок операторов тела функции должна располагаться с новой строки, открывающая и закрывающая скобки имеют одинаковый отступ. Фигурная скобка, начинающая блок операторов цикла или ветвления должна располагаться на одной строке с закрывающейся круглой скобкой логического выражения. Закрывающая скобка располагается на одном уровне с ключевым словом конструкции. Пример форматирования представлен на рисунке 7.

```
int main( ){                          int main( )
→if( x > 0 )                          {
{                                     →if( x > 0 ){
→→x = 1;                      =>      →→x = 1;
}                                     →}
}                                     }
```

Рисунок 7. Пример расположения фигурных скобок

3. Каждый оператор (выражение заканчивающееся ";") должен располагаться с новой строки. В циклах и ветвлениях тело (ветвь) также должны располагаться с новой строки. Пример преобразования на рисунке 8.

```
→int x = 0; if( x > 0 ){x = 1;}      =>   →int x = 0;
                                          →if( x > 0 ){
                                          →→x = 1;
                                          →}
```

Рисунок 8. Форматирование операторов

5. Строки, длина которых превышает 80 символов, должны разбиваться. При разбиении строк необходимо учитывать, что ключевые слова, имена переменных и т.д. разбивать нельзя. Разбиение должно производиться по лексемам. Пример разбиения представлен на рисунке 9 при максимальной длине строки 10 символов.

```
x=test_var + my_long_test_var + 15;    =>   x=test_var +
                                            →→my_long_test_var
                                            →→+ 15;
```

Рисунок 9. Разбиение строки по лексемам при ограничении длины стоки в 10 символов

Запуск команды должен производиться со следующими аргументами командной строки:

```
$ cformat test.c
```

На вход команда ***cformat*** принимает имя файла, в котором расположена программа на языке С, требующая форматирования.

ВАРИАНТ 2.4 Анализ исходного кода програмы на языке С

Задание

Разработать программу ***canalyze*** анализа программ на языке С. Команда ***canalyze*** принимает на вход имя файла, содержащего программу на языке С. Содержимое входного файла подвергается статическому анализу, результаты которого распечатываются на экране.

Критерии оценки

- **Оценка «удовлетворительно»**: Программа работает в интерактивном режиме. Пользователь задает с клавиатуры имя функции, а программа обнаруживает ее во входном файле и печатает номер строки и ее содержимое.
- **Оценка «хорошо»**: осуществляется автоматический поиск всех функций объявленных (через прототип) или определенных в указанном файле. Для каж-

дой из них производится подсчет количества ее вызовов в анализируемом файле.

▪ **Оценка «отлично»**: в дополнение к требованиям оценки «хорошо» должно осуществляться построение таблицы глобальных и локальных переменных. В случае обнаружения конфликта (например, две переменные с одинаковым именем в функции) или перекрытия (локальная переменная перекрывает глобальную) должны выдаваться соответствующие предупреждения.

Указание к выполнению задания

Запуск программы должен производиться со следующими аргументами командной строки:

```
$ canalyze lab.c
```

На вход команда ***canalyze*** принимает имя файла, в котором расположена программа на языке С.

Рассмотрим возможные варианты анализа файла, содержащего программу:

```
#include <stdio.h>
int var1, x; // глобальные переменные
int func1(int a){
    char z;
}
int func2(int a){
    int x, y, z;
    float x;
    func1(x);
}
int main( ){
    int a, var1;
    func1(a);
    func2(var1);
}
```

Работа программы зависит от выбранного уровня сложности. Возможные варианты статического анализа представлены на рисунке 10.

Оценка «удовлетворительно»:	*Оценка «хорошо»:*	*Оценка «отлично»:*

Input function: main
Result:
11: int main( ){
 	
Input function: func1
Result:
3: int func1(int a){
9: func1(x);
13: func1(a);
	Detected functions / call times:
	func1: called 2 times
	func2: called 1 time
	main: called 0 times
		Detected functions / call times:
		func1: called 2 times
		func2: called 1 time
		main: called 0 times
		
		WARNING: global variable x is shadowed by local variable in func2
		
		WARNING: global vari-

		able var1 is shadowed by local variable in main ERROR: conflicting variable x in func2, lines 7 and 8

Рисунок 10. Пример работы программы canalyze

ВАРИАНТ 2.5 Проверка целостности файлов

Задание

Реализовать программу ***integrctrl*** (Integrity Control) проверки целостности содержимого файловой системы. Использование программы состоит из двух шагов.

1. Запись информации о целостности в файл - базу данных. Для активации данного режима программа должна быть запущена с опцией *–s* (save integrity info). Если дополнительно указана опция *–r*, то рекурсивно анализируются все вложенные каталоги. Например:

```
$ integrctrl –s –f database /home/alex/          # анализ только файлов alex и
                                                   запись в database
```

Запуск программы с опцией *–r*:

```
$ integrctrl –s –r –f database /home/alex      # анализ файлов alex и всех вложенных
                                                 директорий, запись в database.
```

2. Сверка информации о целостности, расположенной в указанном файле. Для активации данного режима необходимо использовать опцию –c (check integrity):

```
$ integrctrl –c –f database /home/alex/     # проверка директории /home/alex,
                                              рекурсивность определяется
                                              содержимым database
```

Критерии оценки

- **Оценка «хорошо»**: проверка целостности ориентирована только на файлы, расположенные непосредственно в указанной директории (функционал опции **–r** не реализуется).
- **Оценка «отлично»**: программа позволяет выполнить проверку целостности файлов, расположенных в указанной директории и во всех вложенных в нее.

Указание к выполнению задания

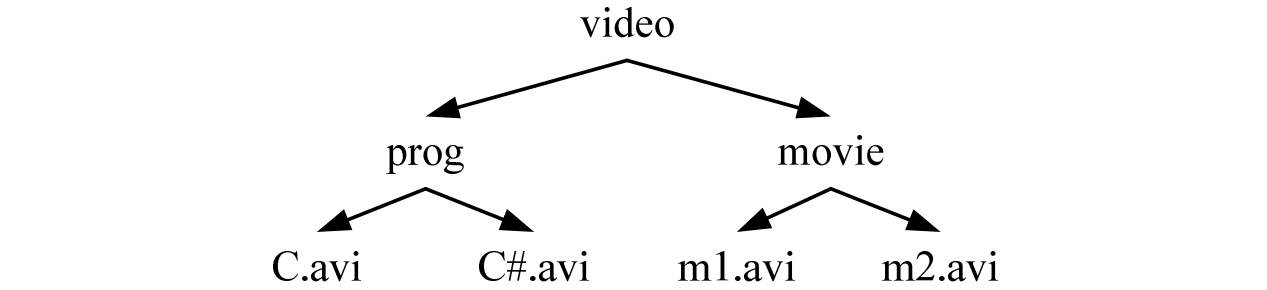
Проверка целостности файлов должна осуществляться с применением технологии хеширования. Для содержимого каждого обрабатываемого файла вычисляется значение хеш-функции с малой вероятностью коллизии. В рамках данной работы предлагается использование функции MD5. Для генерации хеш

кодов MD5 рекомендуется использовать исходные коды, расположенные на сайте Кафедры ВС (*http://cpct.sibsutis.ru/~artpol/downloads/prog/s2/MD5.tar.bz2*) или реализацию в библиотеке OpenSSL (описание ее использования расположено по адресу: *http://www.firststeps.ru/linux/r.php?18*).

Для хранения базы данных рекомендуется использовать бинарные файлы. Предполагается, что структура каталогов не может изменяться. Перемещение директории рассматривается как ее удаление и создание нового каталога в другом месте. Формат одной файловой записи представлен в виде таблицы 2. Пример записи приведен на рисунке 11.

Таблица 2. Рекомендованный формат одной файловой записи.

Название поля	Комментарии
id	Уникальный идентификатор записи в базе данных. Предназначен для описания структуры директорий. Назначается начиная с 1.
name	Имя директории или файла
type	Тип: директория, файл.
parent_id	Уникальный идентификатор родительской директории. Для исходного каталога (для которого формируется информация о целостности) устанавливается в 0.
md5	Если тип записи – файл, то данное поле содержит значение хеш-функции MD5, вычисленное для его содержимого.



id = 1	id = 2	id = 3	id = 4	id = 5	id = 6	id = 7
name=video type=dir parent_id=0 md5 = -	name=prog type=dir parent_id=1 md5 = -	name=C.avi type=file parent_id=2 md5 = <val>	name=C#.avi type=file parent_id=2 md5 = <val>	name=movie type=dir parent_id=1 md5 = -	name=m1.avi type=file parent_id=5 md5 = <val>	name=m2.avi type=file parent_id=5 md5 = <val>

Рисунок 11. Пример базы данных файловых записей

ГЛАВА 3.
ПОЛНОТЕКСТОВЫЙ ПОИСК ПО ШАБЛОНУ

Полнотекстовый поиск — поиск документа в базе данных текстов или в файловой системе (ФС) на основании содержимого этих документов, а также совокупность методов оптимизации этого процесса. Задания данной главы предусматривают разработку программного обеспечения (ПО), позволяющего выполнять полнотекстовый поиск строк по шаблону (wildcard) в файлах, находящихся в указанной директории.

Задача поиска подстроки в строке (string matching problem).

Задача поиска всех фрагментов *текста*, которые совпадают с заданным *образцом*, например:

Образец: `текст`

Текст:

```
Существуют две основных трактовки понятия «текст»: «имма-
нентная» (расширенная, философски нагруженная) и «репре-
зентативная» (более частная). Имманентный подход подразу-
мевает отношение к тексту как к автономной реальности, на-
целенность на выявление его внутренней структуры. Репре-
зентативный – рассмотрение текста как особой формы пред-
ставления знаний о внешней тексту действительности.
```

Результат полнотекстового поиска (в данном примере в виде выделения вхождений образца в тексте):

```
Существуют две основных трактовки понятия «текст»: «имма-
нентная» (расширенная, философски нагруженная) и «репре-
зентативная» (более частная). Имманентный подход подразу-
мевает отношение к тексту как к автономной реальности, на-
целенность на выявление его внутренней структуры. Репре-
зентативный – рассмотрение текста как особой формы представления знаний о внешней тексту действительности.
```

Формальная постановка задачи:

Пусть текст хранится в виде массива символов $T[1 \ldots n]$ длины n, а образец – в виде массива символов $P[1 \ldots m]$ длины m, $m \le n$. Элементы массивов P и T – символы конечного алфавита $\sum$.

Основные соглашения, понятия и обозначения.

Подстроку x строки P, которая начинается в i-м символе и заканчивается в j-м символе будем обозначать через $P[i, \ldots, j]$. Например, если P = “abcdefghijklmnop”, то $P[2 \ldots 6]$ = “bcdef”.

При изложении теоретического материала индексом первого элемента будет 1. При реализации алгоритмов необходимо учитывать, что в языке С индексом первого элемента является 0.

В большинстве кодировок (KOI-8, UTF-8, Windows-1251 и т.п.) коды первых 128 символов совпадают с таблицей ASCII. Для простоты изложения далее будем считать, то работа производится над текстами на английском языке, поэтому алфавит $\sum$ содержит символы таблицы ASCII. Все рассмотренные далее

понятия и методы могут быть распространены на все символы некоторой кодировки.

Будем говорить, что образец P встречается в тексте T со сдвигом s, если $0 \le s \le (n - m)$ и $T[(s + 1) \ldots (s + m)] = P[1 \ldots m]$ (или для любого $j = 1 \ldots m$ $T[s + j] = P[j]$). Также можно сказать, что образец P встречается в тексте, начиная с позиции $s + 1$. Далее будем считать операцию сравнения двух строк элементарной: $T[(s + 1) \ldots (s + m)] = P$, однако при реализации алгоритмов на языке программирования Си для сравнения строк необходимо использовать функцию ***strcmp***.

Если образец P встречается в тексте T со сдвигом s, то s называют допустимым сдвигом. В противном случае – недопустимым. Пример приведен на рисунке 12.1, *а*.

Множество строк конченой длины, которые можно составить из символов алфавита Σ, будем обозначать через Σ^*. Пустую строку (empty string) будем обозначать через ε. Длина строки x обозначается как $|x|$, конкатенация (склеивание, concatenation) строк x и y обозначается как xy, $|xy| = |x| + |y|$.

Строка ω называется префиксом (prefix) строки x ($\omega \sqsubset x$), если существует такая строка $y \in \Sigma^*$, что $\omega y = x$ (рисунок 12.1, б). Строка ω называется суффиксом (suffix) строки x ($\omega \sqsupset x$), если существует такая строка $y \in \Sigma^*$, что $y\omega = x$ (рисунок 12.1, *в*). Из $\omega \sqsubset x$ или $\omega \sqsupset x$ следует $|\omega| \le |x|$. Пустая строка является одновременно суффиксом и префиксом любой строки. Для любых строк x, y и символа a справедливо, что если выполняется $y \sqsupset x$, то также справедливо $ya \sqsupset xa$. Отношения $\sqsupset$ и $\sqsubset$ являются транзитивными: если $y \sqsupset x$ и $x \sqsupset z$, то $y \sqsupset z$. (рисунок 12.1, *г*).

Для краткости обозначим k-символьный префикс $P[1 \ldots k]$ образца $P[1 \ldots m]$ через P_k, тогда $P_0 = \varepsilon$, $P_m = P$.

Тогда задачу поиска подстроки в строке можно сформулировать, как задачу поиска всех сдвигов s таких, что $P \sqsupset T_{s+m}$.

Будем говорить, что строки ω и x сравнимы ($\omega \sim x$), если одна из строк является суффиксом другой. Примерами сравнимых строк будут строки x, y и z из рисунка 12.1, *г*, при этом справедливо как $y \sim x$, так и $x \sim y$.

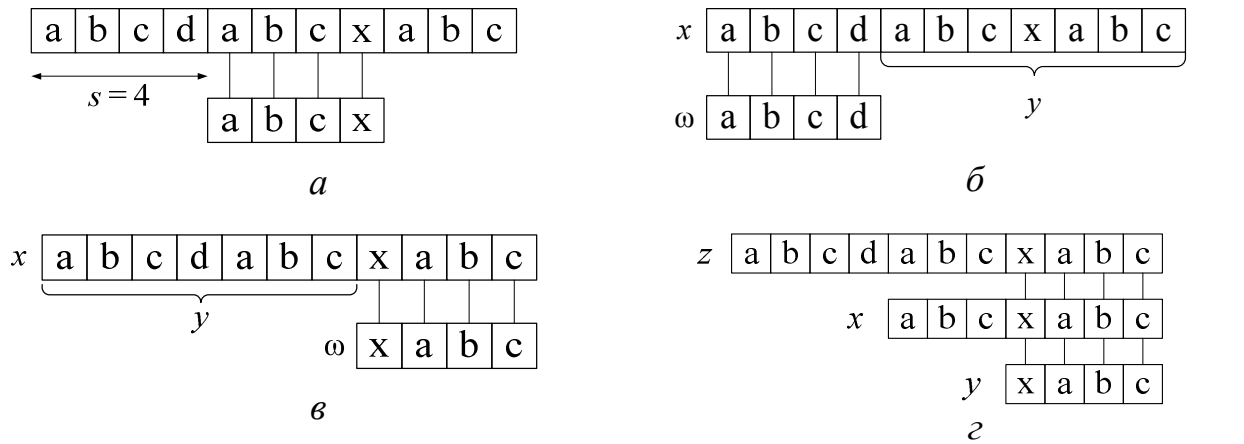
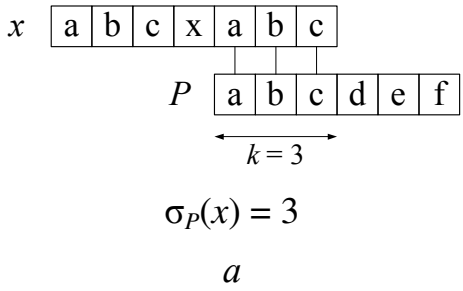


Рисунок 12.1 Основные обозначения

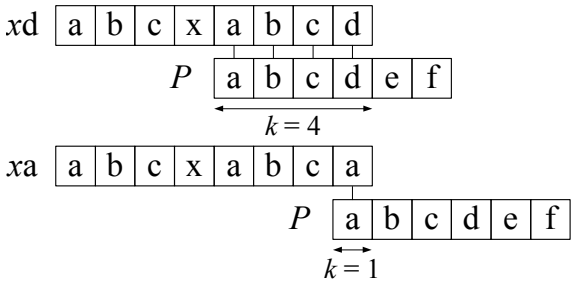
Суффиксной функцией (suffix function) $\sigma_P(x)$, связанной с образцом P, называется функция, ставящая в соответствие некоторой строке x максимальную длину префикса P, который является суффиксом x: $\sigma_P(x) = \max\{k : P_k \sqsupset x\}$ (рисунок 12.2, *а*).

Для любой строки x и символа a выполняется неравенство $\sigma_P(xa) \le \sigma_P(x) + 1$. Это значит, что появление нового символа может увеличить значение функции на 1, если $P_k a = P_{k+1}$. В противном случае максимальная длина префикса P, являющаяся суффиксом xa уменьшится по сравнению с x. На рисунке 12.2, *б* показано, что добавление к строке x символа 'd' приводит к увеличению $\sigma_P(x\text{d})$ на 1 по сравнению с $\sigma_P(x)$. Увеличение на большее значение невозможно, так как добавлен лишь один символ. Также на рисунке видно, что добавление символа 'a' значительно уменьшает значение $\sigma_P(x\text{a})$.



$\sigma_P(x) = 3$

а



$\sigma_P(x\text{d}) = 4$, $\sigma_P(x\text{a}) = 1$

б

Рисунок 12.2 Графическое доказательство неравенства суффикс-функции.

Префиксной функцией (prefix function) заданного образца P называют функцию π_P такую, что:

$$\pi_P[q] = \max\{k : k < q \text{ и } P_k \sqsupset P_q\}.$$

На рисунке 12.3 приведен пример вычисления значения функции π_P для элемента с номером 7.

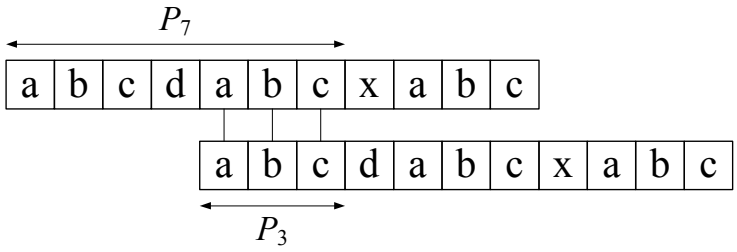


Рисунок 12.3. $\pi_P[7] = 3$

Рассмотрим прямой алгоритм вычисления значений функции π_P для P = "abcdabcxabc"

q	k	$P_k \sqsupset P_q$		$\pi_P[q]$
1	0	$P_0 \sqsupset P_1$		0
	$k < 2$	P_2 ab		
2	1	P_1 a		0
	0	P_0	$P_0 \sqsupset P_2$	
	$k < 3$	P_3 abc		
3	2	P_2 ab		0
	1	P_1 a		
	0	P_0	$P_0 \sqsupset P_3$	
	$k < 4$	P_4 abcd		
	3	P_3 abc		
4	2	P_2 ab		0
	1	P_1 a		
	0	P_0	$P_0 \sqsupset P_4$	
	$k < 5$	P_5 abcda		
	4	P_4 abcd		
5	3	P_3 abc		1
	2	P_2 ab		
	1	P_1 a	$P_1 \sqsupset P_5$	
	$k < 6$	P_6 abcdab		
	5	P_5 abcda		
6	4	P_4 abcd		2
	3	P_3 abc		
	2	P_2 ab	$P_2 \sqsupset P_6$	
	$k < 7$	P_7 abcdabc		
	6	P_6 abcdab		
7	5	P_5 abcda		3
	4	P_4 abcd		
	3	P_3 abc	$P_3 \sqsupset P_7$	
	$k < 8$	P_8 abcdabcx		
	7	P_7 abcdabc		
	6	P_6 abcdab		
	5	P_5 abcda		
8	4	P_4 abcd		0
	3	P_3 abc		
	2	P_2 ab		
	1	P_1 a		
	0	P_0	$P_0 \sqsupset P_8$	

q	k	$P_k \sqsupset P_q$		$\pi_P[q]$
	$k < 9$	P_9 abcdabcxa		
	8	P_8 abcdabcx		1
	7	P_7 abcdabc		
	6	P_6 abcdab		
9	5	P_5 abcda		
	4	P_4 abcd		
	3	P_3 abc		
	2	P_2 ab		
	1	P_1 a	$P_1 \sqsupset P_9$	
	$k < 10$	P_{10} abcdabcxab		
	9	P_9 abcdabcxa		
	8	P_8 abcdabcx		
	7	P_7 abcdabc		
10	6	P_6 abcdab		2
	5	P_5 abcda		
	4	P_4 abcd		
	3	P_3 abc		
	2	P_2 ab	$P_2 \sqsupset P_{10}$	
	$k < 11$	P_{11} abcdabcxabc		
	10	P_{10} abcdabcxab		
	9	P_9 abcdabcxa		
	8	P_8 abcdabcx		
11	7	P_7 abcdabc		3
	6	P_6 abcdab		
	5	P_5 abcda		
	4	P_4 abcd		
	3	P_3 abc	$P_3 \sqsupset P_{11}$	

Полученная функция π_P:

q	1	2	3	4	5	6	7	8	9	10	11
$\pi_P[q]$	0	0	0	0	1	2	3	0	1	2	3

Приведенный алгоритм вычисления π_P не является эффективным, так как осуществляет лишние проверки. Более эффективным будет использовать ранее полученные результаты (значения функции π_P для меньших q). Рассмотрим шаг алгоритма при q = 7 и 8. К моменту обработки $q = 7$ уже получены значения $\pi_P[1] – \pi_P[6]$. Если длина максимального префикса продолжает увеличиваться, то первым необходимо проверить префикс, полученный для P_6, удлиненный на один символ:

7	$k < 7$	P_7 abcdabc		3
	$\pi_P[6]+1$	P_3 abc	$P_3 \sqsupset P_7$	

К моменту обработки $q = 8$ уже получены значения $\pi_P[1] – \pi_P[7]$. Если длина максимального префикса продолжает увеличиваться, то первым необходимо проверить префикс, полученный для P_7, удлиненный на один символ. Далее, если совпадения не происходит, стоит проверить префикс длины $(\pi_P[\pi_P[7]] + 1)$ и т.д.:

8	$k < 8$	P_8 abcdabcx		0
	$\pi_P[7]+1$	P_4 abcd		
	$\pi_P[4]+1$	P_1 a		
	$!\pi_P[1]+1 = \pi_P[1]!$ 0	P_0	$P_0 \sqsupset P_8$	

Псевдокод алгоритма вычисления префикс-функции приведен в листинге 1.

Листинг 1. Алгоритм вычисления префиск-функции.

```
COMPUTE_PREFIX_F(P)
    m ← len(P)
    π_P[1] ← 0
    k ← 0
    for q ← 2 to m do
        while k > 0 и P[k + 1] ≠ P[q] do
            k ← π_P[k]
        if P[k + 1] ≠ P[q] then
            k ← k + 1
        π_P[q] ← k
return π_P
```

Эвристика[1] стоп-символа. Для эффективного использования эвристики проверка образца должна производиться справа налево. Пусть в процессе сравнения образца с текстом по смещению s обнаруживается несовпадение символов $P[i]$ и $T[s+i]$. Пусть первое вхождение символа $T[s+i]$ в P находится по индексу $k < i$. Тогда сдвиг s может быть увеличен на значение $(i – k)$ без риска пропустить допустимый сдвиг. Для применения эвристики стоп-символа необходимо для образца P построить таблицу λ_P (рисунок 12.4, *а*). Пример применения эвристики стоп-символа приведен на рисунке 12.4.

[1] Эвристика - алгоритм решения задачи, не имеющий строгого обоснования, но, тем не менее, дающий приемлемое решение задачи в большинстве практически значимых случаев.

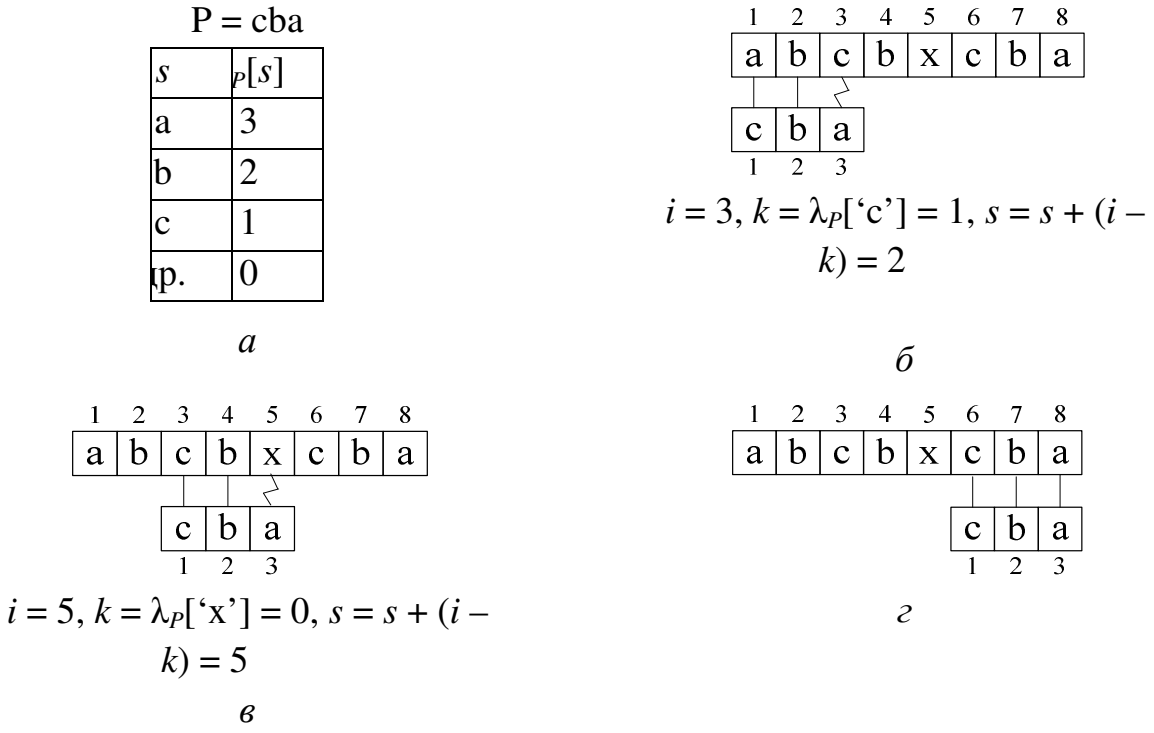


Рисунок 12.4 Пример использования эвристики стоп-символа

Эвристика безопасного суффикса

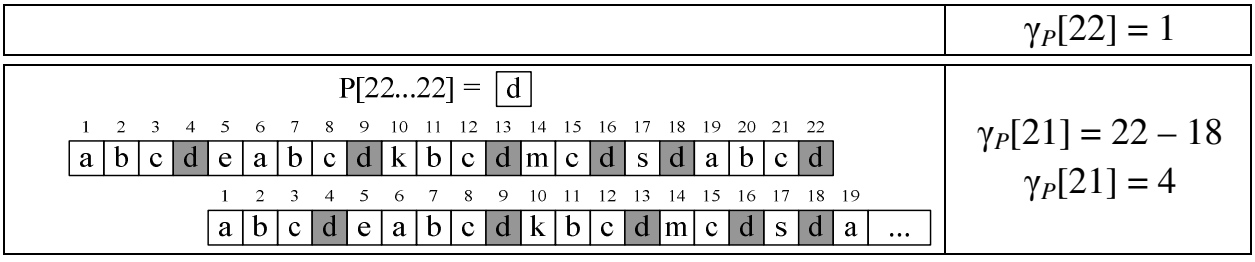
Для эффективного использования эвристики проверка образца должна производиться справа налево. Пусть в процессе сравнения образца с текстом (справа налево) по смещению s обнаруживается несовпадение символов $P[i]$ и $T[s+i]$, $0 < i < m$, т.е. символы образца с $(i+1)$ по m совпадают с соответствующими символами текста. Эвристика безопасного суффикса предполагает использование знаний о содержимом образца для вычисления максимально возможного сдвига, при котором нет риска пропустить допустимый сдвиг. Для использования эвристики безопасного суффикса необходимо по заданному образцу построить функцию безопасного суффикса (good suffix function) γ_P, которая определяется следующим образом:

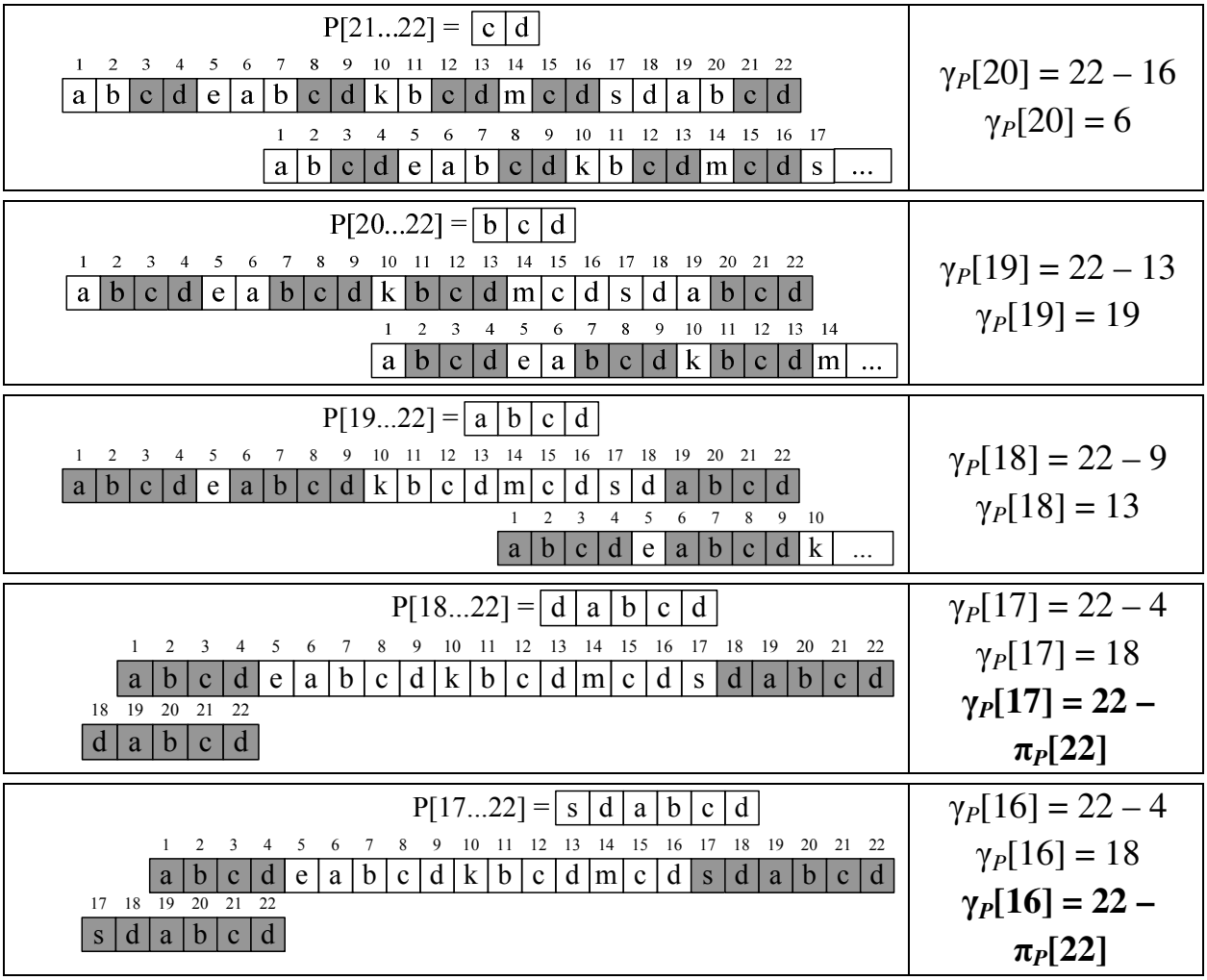
$$\gamma_P[j] = m - \max\{k: 0 \le k < m, P[(j+1) \ldots m] \sim P_k\}$$

Рассмотрим на примере прямой алгоритм построения функции γ_P (рисунок 12.5).

P = "abcdeabcdkbcdmcdsdabcd"

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$\pi_P[i]$	0	0	0	0	0	1	2	3	4	0	0	0	0	0	0	0	0	0	1	2	3	4





. . .

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$\gamma_P[\,i\,]$	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	13	9	6	4	0

Рисунок 12.5 Пример вычисления функции безопасного суффикса

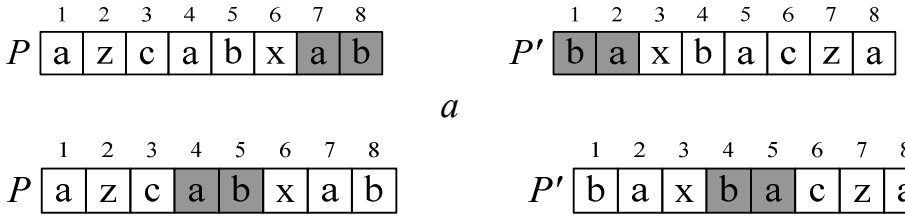
На рисунке 12.5 видно, что когда очередной суффикс $P[i \ldots m]$ не может быть найден в образце, то значение функции безопасного суффикса становится равно значению префикс функции для аргумента i ($\pi_P[i]$). Другими словами мы сдвигаем образец так, чтобы его начало максимально совпало с его окончанием, длина совпадения определяется согласно формуле:

$$\max\{\, k : 0 < k < m\,, P_k \sqsupset P\}.$$

Следовательно, функцию γ_P можно определить следующим образом:

$$\gamma_{\mathrm{P}}[i] = m - \max\{\ \{\pi_P[m]\} \cup \{k : \pi_P[m] < k < m, P[(i+1) \ldots m] \sim P_k\}$$

Рассмотрим более эффективный алгоритм вычисления γ_P, для этого введем строку P', которая представляет собой обращенный образец P (для любых $0 < i < m$ $P'[i] = \mathrm{P}[m - i]$). Если ω – суффикс строки P, то обращение ω является префиксом строки P' (рисунок 12.6, *а*) а предпоследнее вхождение некоторого набора символов x в P – это второе вхождение обращенного набора x в P' (рисунок 12.6, *б*).



k	1	2	3	4	5	6	7	8
$\pi_P[k]$	0	0	0	1	0	0	1	0

k	1	2	3	4	5	6	7	8
$\pi_{P'}[k]$	0	0	0	1	2	0	0	0

в

k	$i = m - k$	$K' = \{ k' : \pi_{P'}[k'] = i\}$	$\min\{K'\} - (m-k)$ $\min\{K'\} - (8-k)$	$m - \pi_P[k]$	$\gamma_P[k]$
1	7	$K = \{\}$	-	8	8
2	6	$K = \{\}$	-	8	8
3	5	$K = \{\}$	-	8	8
4	4	$K = \{\}$	-	8	8
5	3	$K = \{\}$	-	8	8
6	2	$K = \{5\}$	3	8	3
7	1	$K = \{4\}$	3	8	3
8	0	$K = \{1, 2, 3, 6, 7\}$	1	8	1

г

Рисунок 12.6 Соотношения между образцом P и его обращением P'

Таким образом, возможно использование префикс-функции для строки P'. Рассмотрим алгоритм вычисления функции безопасного суффикса, на основе проведенных наблюдений. Псевдокод алгоритма представлен в листинге 2.

Листинг 2. Псевдокод алгоритма вычисления безопасного суффикса.

```
COMPUTE_GOOD_SUFFIX(P)
    π_P ← COMPUTE_PREFIX_F(P)
    P′ = revert(P) // обратить образец
    π_P′ ← COMPUTE_PREFIX_F(P′)
    for k ← 1 to m do
        γ_P[k] ← m – π_P[m]
    for k ← 1 to m do
        j ← m – k
        k′ ← 1
        while (k′ ≤ m) и (π_P′[k′] ≠ j) do
            k′ ← k + 1
        if (π_P′[k′] = j) then
            γ_P[k] ← k′ – (m – k)
    return γ_P
```

Шаблон поиска

Шаблон поиска (wildcard) – метод описания поискового запроса с использованием метасимволов (символов-джокеров). В данной работе будет рассматриваться три метасимвола: '.', '*' и '\', которые имеют следующие значения:

1. '.' означает вхождение произвольного символа один раз. Например, шаблону "т.ст" соответствуют слова "тест" и "тост", но не соответствует слово "текст", так как между "т" и "ст" расположен не один, а два символа.

2. '*' означает вхождение символа, стоящего непосредственно перед '*', ноль, один и более раз. Например, шаблону "go*gle" соответствуют строки "ggle", "gogle", "google", "goooogle", gooooogle" и т.п. Обратите внимание, что повторяться может и метасимвол '.': "g.*le" соответствуют строки "**g**i**gle**","**g**oo**gle**", "**g**angnam st**yle**".

3. '\' является символом экрана. Он используется для того, чтобы указать в шаблоне один из используемых метасимволов в его обычном значении. Например, шаблон "текст закончен\." Говорит о том, что в тексте после слов "текст закончен" должна быть точка. Другой пример: "a*b=c" – в данном контексте мы просим осуществить поиск строки "a*b=c" и хотим, чтобы символ '*' воспринимался как символ умножения. Также можно экранировать сам символ '\': "C:\\\\Windows\\system" соответствует строке "C:\\Windows\system".

Взаимодействие с файловой системой

Для организации рекурсивного обхода могут быть использованы функции **opendir/readdir/closedir** (см. главу 2).

Выделение образца в тексте

Для того, чтобы сделать результаты поиска более наглядными можно использовать цветовыделение текста в консоли ОС GNU/Linux. Демонстрационная программа приведена в листинге 3. Более подробную информацию о возможностях представления текста в консоли можно получить по следующим адресам:

1) http://en.wikipedia.org/wiki/ANSI_escape_code#Colors

2) http://tiebing.blogspot.ru/2010/05/add-color-to-your-linux-console-program.html

Листинг 3. Демонстрационная программа использования цветовыделения текста в консоли ОС GNU/Linux.

```
#include <stdio.h>
#define CSI "\x1B\x5B"

char colors[][5] = {
   "0;30", /* Black  */ "1;30", /* Dark Gray */,
   "0;31", /* Red    */ "1;31", /* Bold Red  */,
   "0;32", /* Green  */ "1;32", /* Bold Green */,
   "0;33", /* Yellow */ "1;33", /* Bold Yellow */
   "0;34", /* Blue   */ "1;34", /* Bold Blue */
   "0;35", /* Purple */ "1;35", /* Bold Purple */
   "0;36", //* Cyan */ "1;36" /*Bold Cyan */ };
```

Листинг 3. Демонстрационная программа использования цветовыделения текста в консоли ОС GNU/Linux.

```
int colors_sz = sizeof(colors)/sizeof(colors[0]);

int main()
{
    char text1[] = "This is demonstration text\n";
    int i;

    // 1. Print out text with different colors
    for(i=0;text1[i] != '\0';i++){
        // never use black and dark gray
        int color = i%(colors_sz-2) + 2;
        printf("%s%sm%c%s0m",CSI,colors[color],text1[i],CSI);
    }
    printf("\n");

    // 2. highlight "demo" word with red in output text
    // this word starts at 8th position and ends at
    // 20th position
    for(i=0;i<8;i++){
        printf("%c",text1[i]);
    }
    printf("%s%sm",CSI,colors[2]); // setup text color to red
    for(i=8;i<=20;i++){
        printf("%c",text1[i]);
    }
    printf("%s0m",CSI); // return normal text color
    for(i=21;text1[i] != '\0'; i++){
        printf("%c",text1[i]);
    }
    printf("\n");
   // 3. highlight with RED and ITALIC in output text
    // word starts at 8th position and ends at 20th position
    for(i=0;i<8;i++){
        printf("%c",text1[i]);
    }
    printf("%s%sm",CSI,colors[2]); // setup text color to red
    printf("%s4m",CSI); // setup text color to bold

    for(i=8;i<=20;i++){
        printf("%c",text1[i]);
    }
    printf("%s0m",CSI); // return normal text color

    for(i=21;text1[i] != '\0'; i++){
        printf("%c",text1[i]);
    }
    printf("\n");

 // 4. highlight "demonstration" word
```

Листинг 3. Демонстрационная программа использования цветовыделения текста в консоли ОС GNU/Linux.

```
      //with RED and WHITE BACKGROUND
      // this word starts at 8th position and ends
      //at 20th position
         for(i=0;i<8;i++){
             printf("%c",text1[i]);
         }
         printf("%s%sm",CSI,colors[2]); // setup text color to red
         printf("%s47m",CSI); // setup background color to white
         for(i=8;i<=20;i++){
             printf("%c",text1[i]);
         }
         printf("%s0m",CSI); // return normal text color

         for(i=21;text1[i] != '\0'; i++){
             printf("%c",text1[i]);
         }

         printf("\n");


}
```

ВАРИАНТ 3.1 Алгоритм Рабина-Карпа

Задание

Реализовать программу ***rkmatcher*** (Rabin-Karp string MATCHER) полнотекстового поиска по шаблону. Шаблон и имя файла (директории) в которой осуществляется поиск, передаются через аргументы командной строки в следующем порядке:

$ **rkmatcher** "g*.le" ~/mydir	#Анализ всех файлов, расположенных в ~/mydir.
$ **rkmatcher -r** "g*.le" ~/mydir	#Рекурсивный поиск во всех директориях, расположенных ниже ~/mydir.

Критерии оценки

- **Оценка «отлично»**: разработанная программа обеспечивает поиск текста по шаблону рекурсивно в заданной директории. Под рекурсивным поиском понимается анализ всех текстовых файлов в текущей директории, а также во всех вложенных директориях.
- **Оценка «хорошо»**: разработанная программа не предусматривает поиск по шаблону ИЛИ не способна выполнять рекурсивный поиск в дереве каталогов (поиск только в одном файле).
- **Оценка «удовлетворительно»**: реализован только алгоритм Рабина-Карпа.

Указание к выполнению задания

Алгоритм Рабина-Карпа предусматривает предварительную обработку образца перед его поиском в тексте. Для этого образец рассматривается как целое число по модулю q, где q – параметр алгоритма. Далее в процессе поиска фрагменты текста также представляются в виде целых чисел, с применением которых производится первое (приближенное) сравнение. Сначала рассмотрим задачу поиска цифровой последовательности на рисунке 13.

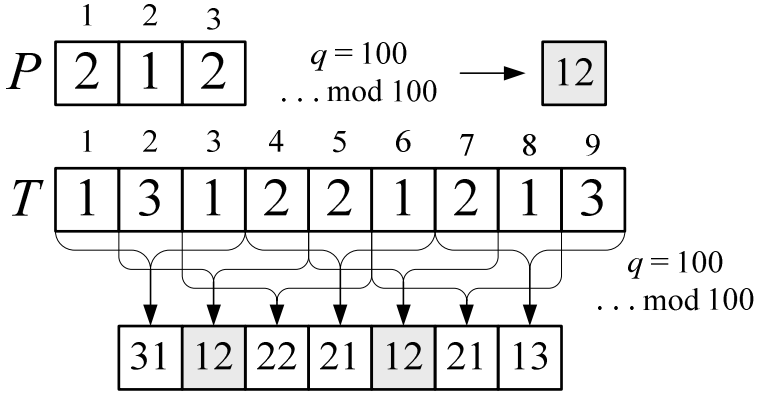


Рисунок 13. Поиск цифровой последовательности

Производится поиск трехсимвольного сочетания цифр "212". Соответствующее целое число – 212. Если q будет равно 1000, то для поиска образца достаточно будет выполнить сравнение чисел, получаемых из подпоследовательностей текста.

Однако в данном примере $q = 100$, поэтому возможны "коллизии", когда несколько трехсимвольных сочетаний соответствуют одному числу. В примере это "312" и "212".

Таким образом, при совпадении чисел необходимо дополнительно выполнить посимвольное сравнение образца с фрагментом текста.

Пусть преобразование некоторой строки x в число в n-ичной системе счисления производится функцией $h(x)$. Если известно число h_1, соответвующее строке $T[i \ldots (i + m)]$ ($h_1 = h(\mathrm{T[i \ldots i+m]})$), то для вычисления числа h_2, соответствующего $T[(i + 1) \ldots ((i + 1) + m)]$, достаточно вычесть из h_1 вклад символа $T[i]$, который больше не присутствует во фрагменте текста, далее сдвинуть число h_1 на один n-ичный разряд влево, после этого добавить вклад нового символа $T[(i + 1) + m]$. Рассмотрим пример:

T = "131221213", $T[3 \ldots 5]$ = "122", $T[4 \ldots 6]$ = "221", $h_1 = h(T[3 \ldots 5])$, для вычисления h_2 необходимо отнять вклад $T[3]$: $h_2 = h_1 - 10^2 \cdot \text{'1'} = 122 - 100 = 22$, далее сдвинуть h_2 на один n-ичный разряд влево: $h_2 = h_2 \cdot 10 = 220$. Следующим шагом требуется прибавить вклад символа $T[6]$: $h_2 = h_2 + \text{'1'} = 221$.

Для вычисления числового представления строки, содержащей произвольные символы таблицы ASCII, обычно используется тот факт, что каждому символу ставится в соответствие ASCII-код, который находится в диапазоне от 0 до 255. Таким образом, строка может быть рассмотрена как число в 256-ричной системе счисления. Более эффективным будет использование следующей функции:

$h(x) = \text{code}(x[i]) \cdot d^m + \text{code}(x[i+1]) \cdot d^{(m-1)} + \ldots + \text{code}(x[i+j]) \cdot d^{(m-j)} + \ldots + \text{code}(x[i+m]) \cdot d^0$,

где d – основание используемой системы счисления, при $d = 256$ каждая строка будет представлена уникально, но предпочтительнее использование значений d = 7, 13, 17, 23, 29, 31, 37 (простые числа).

В качестве q в работе предлагается использовать ограничения, накладываемые типом unsigned int и стандартную обработку переполнений (отбрасывание старших разрядов), которая реализована в современных процессорах. Рассмотрим обработку переполнения для ячейки размером 1 байт, предствленную на рисунке 14.

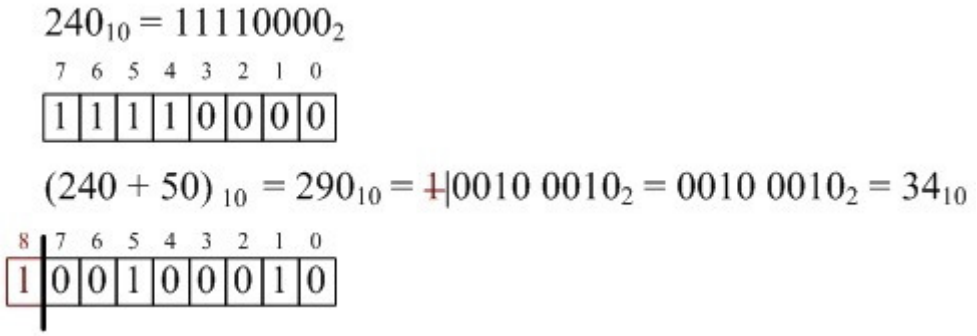


Рисунок 14. Обработка переполнения ячейки

Обработка переполнений для многобайтовых ячеек производится аналогичным образом.

ВАРИАНТ 3.2 Автоматы поиска подстрок

Задание

Реализовать программу ***fsmatcher*** (Finit State Machine string mATCHER) полнотекстового поиска по шаблону. Шаблон и имя файла (директории), в которой осуществляется поиск, передаются через аргументы командной строки в следующем порядке:

$ **fsmatcher** "g*.le" ~/mydir	#Анализ всех файлов, расположенных в ~/mydir.
$ **fsmatcher -r** "g*.le" ~/mydir	#Рекурсивный поиск во всех директориях, расположенных ниже ~/mydir.

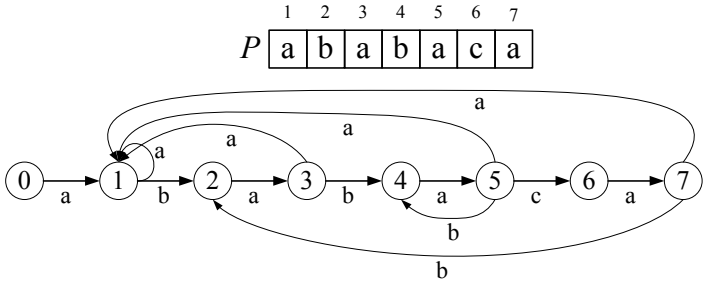
Критерии оценки

- **Оценка «отлично»**: разработанная программа обеспечивает поиск текста по шаблону рекурсивно в заданной директории. Под рекурсивным поиском понимается анализ всех текстовых файлов в текущей директории, а также во всех вложенных директориях.
- **Оценка «хорошо»**: разработанная программа не предусматривает поиск по шаблону ИЛИ не способна выполнять рекурсивный поиск в дереве каталогов (поиск только в одном файле).
- **Оценка «удовлетворительно»**: реализован алгоритм поиска с помощью конечного автомата.

Указание к выполнению задания

Алгоритм поиска, использующий конечные автоматы (КА), основан на следующем принципе: для заданного шаблона $P[1 \ldots m]$ строится конечный автомат $M(Q,q_0,A,\sum,\delta)$, где $Q = \{0, 1, 2, \ldots, m\}$ – конечное множество состояний (states), $q_0 = 0$ – начальное состояние, $A = m$ - конечное множество заключительных состояний. $\sum$ – входной алфавит, $\delta(q,a)$ – функция переходов, которая показывает, в какое состояние переходит КА, находящийся в состоянии q, при появлении символа a входного алфавита.

Функция δ строится для образца, для ее построения используется суффикс-функция, рассмотренная в общем материале к данному разделу. Она определяется следующим образом: $\delta(q,a) = \sigma(P_q a)$, т.е. если КА находился в состоянии q и был обнаружен символ a, то КА переходит в состояние $\sigma(P_q a)$, т.е. определяет максимальную длину k префикса строки $P[1 \ldots q]$, к которой добавлен символ a, совпадающего со своим суффиксом ($k < q, P[1 \ ... \ k] \sqsupset P_q a]a$). Пример работы алгоритма представлен на рисунке 15.



Соответствующая графу таблица переходов:

δ	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
a	1	1	3	1	5	1	7	1
b	0	2	0	4	0	4	0	2
c	0	0	0	0	0	6	0	0

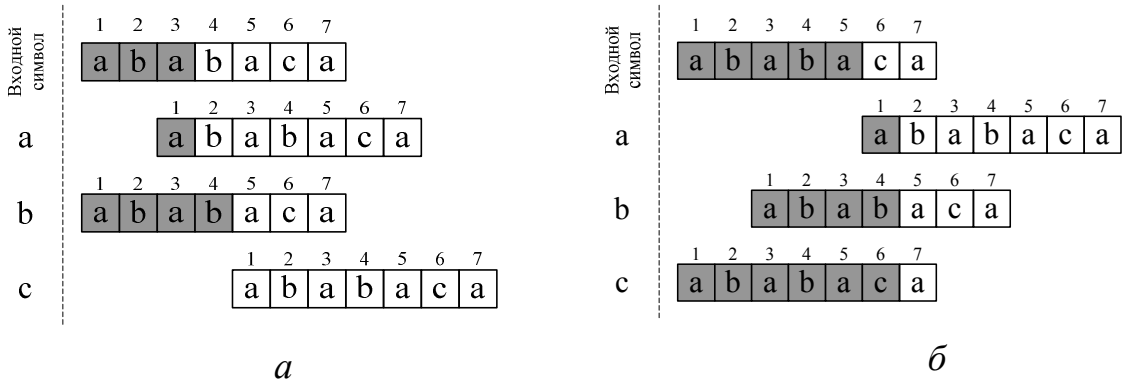
Рисунок 15. Алгоритм поиска, использующий конечные автоматы

Рассмотрим подробнее некоторые этапы построения полученной таблицы:

1. Если КА находится в начальном состоянии ($q = 0$), то поступление любого символа, отличного от 'a' оставляет КА в исходном состоянии. Если получен символ 'a', то КА переходит в состояние 1, что означает, что в тексте найдены символы P[1 … 1].

2. Если КА находится в состоянии $q = 1$ (обнаружены символы P[1 … 1]) и поступает символ 'a', то КА остается в состоянии $q = 1$ (шаблон передвигается вправо на 1 позицию). Действительно, образец P = "ababaca", а прочитаны символы "aa", вхождение образца в текст возможно только со 2-й или более поздней позиции. Символ 'b' переводит КА в состояние 2, а символ 'c' – в состояние $q = 0$ (такие дуги не обозначены на рисунке для упрощения его чтения).

На рисунке показаны действия, предпринимаемые при обнаружении допустимых символов, если КА находится в состоянии $q = 3$ и 5 (рисунок 16 а и б), а также в листинге 4 предствлен псевдокод алгоритма вычисления $\delta(q,a)$.



а *б*

Рисунок 16. Пример работы КА в различных состояниях

Листинг 4. Псевдокод алгоритма вычисления δ(*q*,*a*).

```
COMPUTE_TRANSITION_F(P, ∑)
    m ← len(P)
    for q ← 0 to m do
        k ← min(m+1, q+2)
        while не ( P_k ⊐ P_q a)
            k ← k – 1
        δ(q,a) ← k
return δ
```

ВАРИАНТ 3.3 Алгоритм Кнута-Морриса-Пратта

Задание

Реализовать программу ***kmpmatcher*** (Knuth–Morris–Pratt string MATCHER) полнотекстового поиска по шаблону. Шаблон и имя файла (директории), в которой осуществляется поиск, передаются через аргументы командной строки в следующем порядке:

$ **kmpmatcher** "g*.le" ~/mydir	#Анализ всех файлов, расположенных в ~/mydir.
$ **kmpmatcher -r** "g*.le" ~/mydir	#Рекурсивный поиск во всех директориях, расположенных ниже ~/mydir.

Критерии оценки

- **Оценка «отлично»**: разработанная программа обеспечивает поиск текста по шаблону рекурсивно в заданной директории. Под рекурсивным поиском понимается анализ всех текстовых файлов в текущей директории, а также во всех вложенных директориях.
- **Оценка «хорошо»**: разработанная программа не предусматривает поиск по шаблону ИЛИ не способна выполнять рекурсивный поиск в дереве каталогов (поиск только в одном файле).
- **Оценка «удовлетворительно»**: реализован только алгоритм Кнута-Морриса-Пратта.

Указание к выполнению задания

Алгоритм Кнута-Морриса-Пратта (КМП) основан на применении префикс-функции π_P, подробно описанной в общей информации к данному разделу. В листинге 5 приведен псевдокод алгоритма КМП.

Листинг 5. Псевдокод алгоритма КМП

```
KMP_MATCHER(T, P)
    m ← len(P)
    n ← len(T)
    π_P ← COMPUTE_PREFIX_F(P)
    q ← 0
    for i ← 1 to n do
        while q > 0 и P[q + 1] ≠ T[i] do
            q ← π_P[q]
        if P[q + 1] = T[i] then
            q ← q+1
        if q = m then
            print "Образец обнаружен при сдвиге" i – m
        q ← π_P[q]
```

ВАРИАНТ 3.4 Алгоритм Бойнега-Мура

Задание

Реализовать программу ***bmatcher*** (Boyer–Moore string mATCHER) полнотекстового поиска по шаблону. Шаблон и имя файла (директории), в которой осуществляется поиск, передаются через аргументы командной строки в следующем порядке:

$ **bmatcher** "g*.le" ~/mydir	#Анализ всех файлов, расположенных в ~/mydir.
$ **bmatcher -r** "g*.le" ~/mydir	#Рекурсивный поиск во всех директориях, расположенных ниже ~/mydir.

Критерии оценки

- **Оценка «отлично»**: разработанная программа обеспечивает поиск текста по шаблону рекурсивно в заданной директории. Под рекурсивным поиском понимается анализ всех текстовых файлов в текущей директории, а также во всех вложенных директориях.
- **Оценка «хорошо»**: разработанная программа не предусматривает поиск по шаблону ИЛИ не способна выполнять рекурсивный поиск в дереве каталогов (поиск только в одном файле).
- **Оценка «удовлетворительно»**: реализован только алгоритм Бойера-Мура с эвристикой стоп-символа.

Указание к выполнению задания

Алгоритм Бойера-Мура (БМ) основан на применении эвристики стоп-

символа и эвристики безопасного суффикса, которые были подробно рассмотрены в общей информации к данному разделу. В листинге 6 приведен псевдокод алгоритма БМ.

Листинг 6. Псевдокод алгоритма БМ

```
BM_MATCHER(T, P)
    m ← len(P)
    n ← len(T)
    // построить таблицу эвристики стоп-символа
    λ_P ← COMPUTE_LAST_OCCURENCE(P)
    γ_P ← COMPUTE_GOOD_SUFFIX(P) // построить таблицу γ_P
    s ← 0
    while s < (n – m) do
        j ← m
        while j > 0 и P[j] = T[s + j] do
            j ← j – 1
        if j = 0 then
            print "Образец обнаружен при сдвиге" i – m
            q ← γ_P[0]
        else
  s ← s + max(γ_P[j], j – λ_P[ T[s + j] ])
```

ГЛАВА 4.
СЖАТИЕ ДАННЫХ

Заданиями данной главы предусматривается разработка программ, обеспечивающих сжатие данных на основе известных алгоритмов:

1) Шенона-Фано;
2) Хаффмана;
3) Лемпеля – Зива (LZ77, LZ78, LZSS, LZW).

Коэффициент сжатия

Коэффициент сжатия – основная характеристика алгоритмов сжатия. Она определяется как отношение объёма исходных несжатых данных к объёму сжатых, то есть:

$$k = \frac{s_t}{s_c},$$

где k — коэффициент сжатия, s_t — объём исходного сообщения, а s_c — объём сжатого сообщения. Таким образом, чем выше коэффициент сжатия, тем алгоритм эффективнее. Если $k = 1$, то алгоритм не производит сжатия, то есть выходное сообщение оказывается по объёму равным входному. Если $k < 1$, то алгоритм порождает сообщение большего размера, нежели несжатое, то есть, совершает «вредную» работу.

Работа с битовым массивом

Алгоритмы Шенона-Фано и Хаффмана предполагают построение новых кодов для символов входного алфавита, основанных на частоте встречаемости этих символов. При этом длина кодов часто встречающихся символов будет наименьшей (1 – 4 бит), в то время, как коды редко встречающихся символов могу составлять 8, 9, 10 и более бит. Хранение символов с короткими (< 8 бит) кодами в 8-битном байте оказывается бесполезным, так как эффект от сжатия кода нивелируется наличием неиспользуемых бит. С другой стороны не существует возможности сохранить 10-битное слово в одном байте, потребуется использовать ячейки двухбайтного типа.

Таким образом, возникает необходимость *компактного* хранения кодов, имеющих различную длину в массивах, элементы которых имеют фиксированный размер. Для решения этой задачи может использоваться *битовый массив*. Для работы с этим массивом необходимо разработать две функции: ***setbits***(array, offset, value, value_len) и ***getbits***(array, offset, len). Рассмотрим далее принцип действия этих функций.

Пусть дана таблица α кодов, построенная некоторым алгоритмом сжатия (см. Таблицу 3).

Таблица 3. Таблица α кодов.

Символ	Код (code)	Длина кода (len)
a	010	3
b	1100	4
c	010111	6
d	1110001111	10

Необходимо закодировать сообщение: "abaabcd". На рисунке 17 показано исходное сообщение.

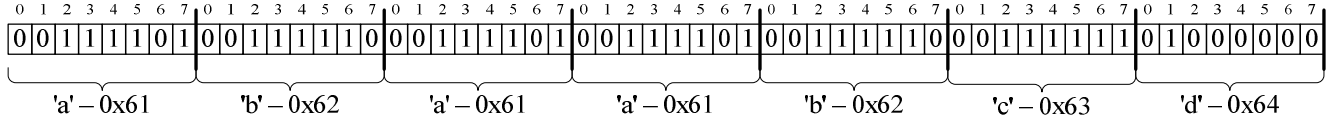


Рисунок 17. Исходное сообщение в формате ASCII

Тогда закодированное сообщение представлено на рисунке 18.

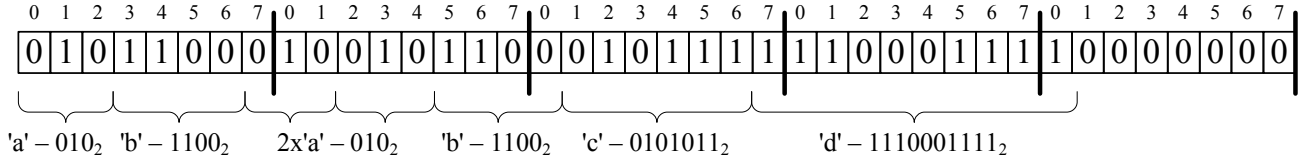


Рисунок 18. Закодированное сообщение

Видно, что исходное сообщение занимает 7 байт, а сжатое – 5 байт, при этом 7 бит последнего байта свободны, т.е. можно сказать, что длина сжатого сообщения близка к 4-м байтам. Таким образом коэффициент сжатия для данного кодирования будет:

$$k = \frac{7}{4\frac{1}{8}} = \frac{56}{33} = 1{,}6969(69) \approx 1{,}7.$$

Далее в листинге 7 представлен псевдокод алгоритма формирования битового массива *C*, хранящего закодированное (сжатое) сообщение *T*.

Листинг 7. Псевдокод алгоритма формирования битового массива *C*

```
ENCODE_MSG(T, α)
    offs ← 0   // смещение в битовом массиве устанавливается в ноль
    C ← "" // массив бит исходно пустой
    n ← len(T)
    for  n ← 1 to n do
        v ← α[ T[i] ].code
        l ← α[ T[i] ].len
        offs ← setbits(C, offs, v, len)
    return C
```

Функция ***setbits*** должна по номеру бита определить номера байтов, в которых будет сохранено значение *v*. Например, если *offs* = 10, а *len* = 15, то первый байт, в который будут записаны разряды *v*, имеет номер *offs* **div** 8, при этом если *offs* **mod** 8 ≠ 0, то необходимо прибавить еще один байт.

Для рассмотренного примера 10 **div** 8 = 1, т.к. 10 **mod** 8 = 2 ≠ 0, то номер первого байта будет 2. Далее необходимо разместить биты значения *v* в байтах массива как показано на рисунке 19.

Для решения задач по выделению фрагментов значения и помещения их в соответствующие байты необходимо использовать поразрядные операции.

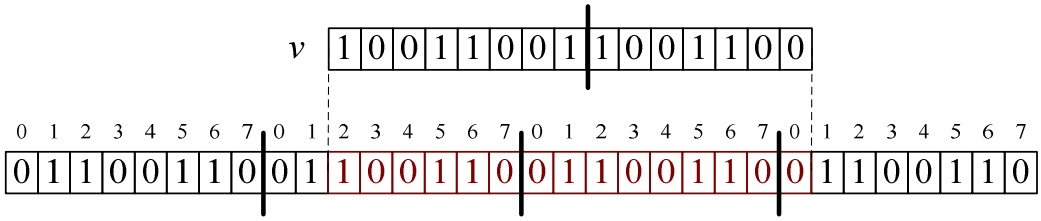


Рисунок 19. Этап декодирования

Процесс декодирования сообщения представлен в листинге 8.

Листинг 8. Процесс декодирования сообщения.

```
DECODE_MSG(C, α)
    offs ← 0    // смещение в битовом массиве устанавливается в ноль
    T ← ""      // массив декодированного сообщения исходно пустой
    n ← len(C) // размер сообщения в байтах
    tmp ← α
    v ← 0
    for  i ← 1 to n·8 do
        v ← v<<1 | getbits(C, i, 1)
        tmp ← clear(tmp,v) // убрать из tmp коды, начало которых не совпадает с v
        if find_code(tmp, v, s) = 1 then // в tmp есть символ s такой, что tmp[s].code = v
            T = Ts // конкатенация декодированного текста с новым символом
            tmp = C // tmp снова равен множеству всех доступных кодов
            v ← 0 // v сбрасывается в 0, чтобы в ней формировался код след. символа.
        else if |tmp| = 0  then // tmp пусто – не удалось найти такого кода – ошибка!
        print "Ошибка декодирования, кода v не существует!"
```

Бинарные файлы

При реализации архиваторов возникает необходимость хранения в файлах битовых массивов, а также структур данных. Представление этой информации с помощью текста не является компактным. Для хранения на диске информации, отличной от текстовой используются двоичные (бинарные) файлы. Для работы с бинарными файлами обычно используются функции *fread* и *fwrite*. Эти функции позволяют читать и записывать блоки данных любого типа. Прототипы этих функций следующие:

```
size_t fread (void *buffer, size_t size, size_t count,
FILE *fp);
size_t fwrite (const void *buffer, size_t size,size_t
count, FILE *fp);
```

Буфер (*buffer*) – указатель на область памяти, в которую будут прочитаны или из которой будут записаны данные. Счетчик *count* определяет, сколько считывается и записывается элементов данных, причем длина каждого элемента в байтах равна *size*. Функция *fread* возвращает количество прочитанных элементов. Если достигнут конец файла или произошла ошибка, то возвращаемое значение может быть меньше, чем счетчик. Функция *fwrite* возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению счетчика. Одним из самых полезных применений функций fread() и fwrite() является чтение и запись данных пользовательских типов,

особенно структур. Пример использования функций fread() и fwrite() представлен в листинге 9.

Листинг 9. Пример использования функций fread() и fwrite()

```
FILE * fp;
char filename[] = "some_file";
struct some_struct buff[64]; // массив структур из
                             // 64-х элементов
fp =fopen(filename, "wb")
if( fp == NULL ){
printf("Error opening file %s\n",filename);
}else{
fwrite(buff, sizeof(struct some_struct), 64, fp);
fclose(fp);
}
. . . . . .
fp =fopen(filename, "rb")
if( fp == NULL ){
printf("Error opening file %s\n",filename);
}else{
fread(buff, sizeof(struct some_struct), 64, fp);
fclose(fp);
}
```

ВАРИАНТ 4.1 Алгоритм Шеннона-Фано (Shannon-Fano)

Задание

Реализовать программу ***sfcompress*** сжатия текстовых файлов на английском языке алгоритмом Шенона-Фано. Сжатие осуществляется с аргументом командной строки *-c* (compress), а распаковка – с аргументом *-d* (decompress). Опция *-o* указывает имя выходного файла. Например:

$ **sfcompress** -c -o file.sfc file.txt	# сжатие file.txt в file.sfc
$ **sfcompress** -d -o file1.txt file.sfc	# распаковка file.sfc в file1.txt

Критерии оценки

- **Оценка «хорошо»**: реализован алгоритм сжатия, не обеспечено компактное расположение сжатого текста в выходном файле – код каждого символа занимает целое число байт.
- **Оценка «отлично»**: закодированный текст компактно упаковывается в файл без учета байтовых границ (с помощью битового массива, как описано в общей информации к разделу).

Указание к выполнению задания

Алгоритм Шеннона-Фано использует коды переменной длины. Часто встречающийся символ кодируется кодом меньшей длины, редко встречающийся – кодом большей длины. Например, исходный текст:

«ббббмдддуууууууууу». Частоты встречающихся в исходном тексте символов: б – 4, м – 1, ‘д’ – 3, ‘у’ – 6. Основные этапы алгоритма Шеннона-Фано:

1. Список X символов упорядочивается по убыванию частоты встречаемости.

2. Список X делится на две части X_1 и X_2 так, чтобы разница сумм частот обеих частей (Σ_1 и Σ_2) была минимальна, например $\Sigma_1 = 6$ (симв. ‘у’), $\Sigma_2 = 8$ (симв. ‘б’, ‘м’, ‘д’).

3. Кодовой комбинации каждого символа дописывается 1 (X_1) или 0 (X_2).

4. Если X_1 содержит **один** символ, код этого символа построен, перейти к шагу 5. Иначе – рекурсивно перейти к шагу 2, считая $X = X_1$.

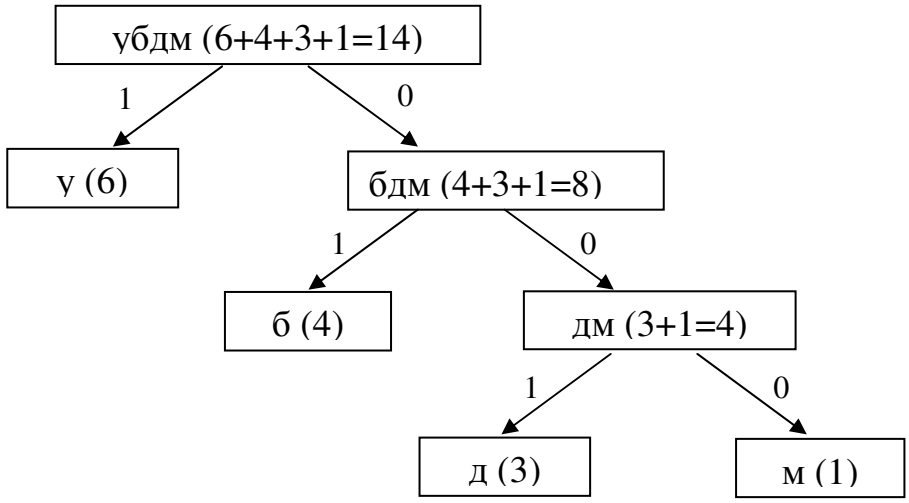
5. Если X_2 содержит один символ, код этого символа построен – завершить данный шаг рекурсии (на предыдущих шагах рекурсии могут быть необработанные данные). Если символов больше одного – рекурсивно перейти на шаг 2, считая, что $X = X_2$.

В листинге 10 приведен высокоуровневый псевдокод алгоритма.

Листинг 10. Псевдокод алгоритма Шеннона–Фано

```
SFCOMPRESS(text, start, end)
    index = distribute(text, start, end)
    if (index – start) > 1 then SFCOMPRESS(text, start, index)
    else code_finished(text[index])
    if (end – index) > 1 then SFCOMPRESS(text, index + 1, end)
    else code_finished(text[index])
```

Для распаковки сжатых данных необходимо знать все использованные коды, поэтому требуется сохранять полученную таблицу соответствия «символ-код» в сжатом файле. Сжатие сообщения, приведенного выше, представлено на рисунке 17.



Будут получены следующие коды символов:

у – 1

б - 01

д - 001

м - 000

Рисунок 20. Пример получения пары символ - код

Текст «ббббмдддуууууууууу» будет закодирован в виде двоичной последовательности, представленной на рисунке 21.

01|01|01|01|000|001|001|001|1|1|1|1|1|1|1|1|1|1|1

Рисунок 21. Закодированная двоичная последовательность

ВАРИАНТ 4.2 Алгоритм Хаффмана

Задание

Реализовать программу ***hcompress*** сжатия текстовых файлов на английском языке алгоритмом Хаффмана. Сжатие осуществляется с аргументом командной строки *-c* (compress), а распаковка – с аргументом *-d* (decompress). Опция *-o* указывает имя выходного файла. Например:

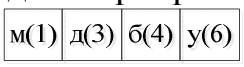
$ **hcompress** -c -o file.hc file.txt	# сжатие file.txt в file.hc
$ **hcompress** -d -o file1.txt file.hc	# распаковка file.hc в file1.txt

Критерии оценки

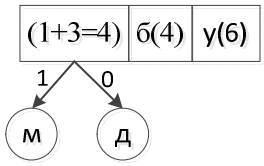
- **Оценка «хорошо»**: реализован алгоритм сжатия, не обеспечено компактное расположение сжатого текста в выходном файле – код каждого символа занимает целое число байт.
- **Оценка «отлично»**: закодированный текст компактно упаковывается в файл без учета байтовых границ (с помощью битового массива, как описано в общей информации к разделу).

Указание к выполнению задания

В алгоритме Хаффмана на основании таблицы частот встречаемости символов в сообщении строится дерево кодирования Хаффмана (Н-дерево). Например, исходный текст: «ббббмдддуууууууууу». Частоты встречающихся в исходном тексте символов: б – 4, м – 1, ‘д’ – 3, ‘у’ – 6. Основные этапы алгоритма Хаффмана:

1. Список *X* символов упорядочивается по возрастанию частоты встречаемости и записывается в очередь с приоритетом (каждый элемент очереди - узел дерева Хаффмана). 

2. Выбираются два элемента очереди ω_1 и ω_2 с наименьшими весами (частотами встречаемости) . Для рассматриваемого примера ω_1 = ‘м’ и ω_2 = ‘д’.
3. Создается их родитель ω с весом, равным их суммарному весу (Σ =1+3=4). Родитель добавляется в соответствующее место очереди, а два его потомка удаляются оттуда.
4. Кодовой комбинации символов дописывается 1 (ω_1) или 0 (ω_2).



5. Если в очереди находится один элемент - дерево Хаффмана построено, иначе переходим на шаг 2.

В листинге 11 приведен высокоуровневый псевдокод алгоритма построения дерева Хаффмана.

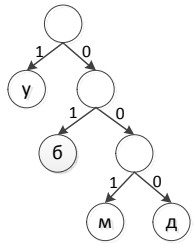
Листинг 11. Псевдокод алгоритма построения дерева Хаффмана

```
HTREE(queue, sequence)
    init_queue(queue) // заполнить очередь символами сообщения и их весами
    while size(queue) > 1 do
```

```
ω1=dequeue_min(queue) // извлечь (с удалением!) элемент с наименьшим весом
ω2=dequeue_min(queue) // извлечь (с удалением!) элемент с наименьшим весом
ω.left = ω1
ω.right = ω2
ω.weight = ω1.weight + ω2.weight
 enqueue(queue, ω)
```

Для формирования таблицы кодов необходимо выполнит обход построенного дерева в глубину.

Для распаковки сжатых данных необходимо знать все использованные коды, необходимо сохранять полученную таблицу соответствия «символ-код» в файл со сжатой информацией. Итоговое дерево Хаффамана для сообщения, приведенного выше, изображено на рисунке 22:



Будут получены следующие коды символов:
у – 1
б – 01
д – 000
м – 001

Рисунок 22. Итоговое деверо Хаффмана

Текст «ббббмдддуууууууууу» будет закодирован в виде двоичной последовательности:

01|01|01|01|001|000|000|000|1|1|1|1|1|1|1|1|1

Рисунок 23. Закодированная двоичная последовательность

ВАРИАНТ 4.3 Алгоритм Лемпела – Зива (Lempel – Ziv) LZ77

Задание

Реализовать программу ***lz77compress*** сжатия текстовых файлов на английском языке алгоритмом Зива-Лемпела. Сжатие осуществляется с аргументом командной строки *-c* (compress), а распаковка – с аргументом *-d* (decompress). Опция *-o* указывает имя выходного файла. Например:

$ **lz77compress** -c -o file.lz77 file.txt	# сжатие file.txt в file.lz77
$ **lz77compress** -d -o file1.txt file.lz77	# распаковка file.lz77 в file1.txt

Критерии оценки

- **Оценка «хорошо»**: реализован алгоритм сжатия, для записи кодов в файл используются структуры данных.
- **Оценка «отлично»**: можно задать любой размер словаря и буфера, для формирования файлового элемента используется битовый массив (как описано в общей информации к разделу 4).

Указание к выполнению задания

LZ77 использует скользящее по сообщению окно. Метод кодирования согласно принципу скользящего окна учитывает уже ранее встречавшуюся ин-

формацию, то есть информацию, которая уже известна для кодировщика и декодировщика (второе и последующие вхождения некоторой строки символов в сообщении заменяются ссылками на ее первое вхождение). Окно состоит из двух частей – словаря (большая часть) и буфера. Первая, большая по размеру, включает уже просмотренную часть сообщения. Вторая, меньшая по размеру, содержит еще незакодированные символы входного потока. Алгоритм пытается найти в словаре фрагмент, совпадающий с содержимым буфера. Алгоритм LZ77 выдает коды, состоящие из трех элементов:

- смещение подстроки, совпадающей с началом содержимого буфера, относительно начала словаря;
- длина этой подстроки;
- первый символ буфера, следующий за подстрокой.

В конце итерации алгоритм сдвигает окно на длину равную длине подстроки, обнаруженной в словаре.

Рассмотрим алгоритм на примере. Пусть нам необходимо закодировать строку: «про проверку и проведение проводки». При кодировании будем использовать окно размером 23 символа, где первые 15 символов будут словарем, а следующие 8 – буфером.

Словарь															Буфер								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	1	2	3	4	5	6	7	8	Код
															п	р	о		п	р	о	в	<0,0,‘п’>
														п	р	о		п	р	о	в	е	<0,0,‘р’>
													п	р	о		п	р	о	в	е	р	<0,0,‘о’>
												п	р	о		п	р	о	в	е	р	к	<0,0,‘ ’>
											п	**р**	**о**		**п**	**р**	**о**	в	е	р	к	у	<11,3,‘в’>
							п	р	о		п	р	о	в	е	р	к	у		и		п	<0,0,‘е’>
						п	**р**	о		П	р	о	в	е	**р**	к	у		и		п	р	<7,1,‘к’>
				п	р	о		п	р	О	в	е	р	к	у		и		п	р	о	в	<0,0,‘у’>
			п	р	о		п	р	о	В	е	р	к	у		и		п	р	о	в	е	<6,1,‘и’>
	п	р	о		**п**	**р**	**о**	**в**	**е**	Р	к	у		и		**п**	**р**	**о**	**в**	**е**	д	е	<4,6,‘д’>
о	в	**е**	р	к	у		и		п	Р	о	в	е	д	**е**	н	и	е		п	р	о	<2,1,‘н’>
е	р	к	у		**и**		п	р	о	В	е	д	е	н	**и**	е		п	р	о	в	о	<5,1,‘е’>
к	у		и		**п**	**р**	**о**	**в**	е	Д	е	н	и	е		**п**	**р**	**о**	**в**	о	д	к	<4,5,‘о’>
р	о	в	е	**д**	е	н	и	е		П	р	о	в	о	**д**	к	и						<4,1,‘к’>
в	е	д	е	н	**и**	е		п	р	О	в	о	д	к	**и**								<5,0,‘и’>

Закодированный текст «про проверку и проведение проводки» представлен на рисунке 24.

0 0 ‘п’|0 0 ‘р’|0 0 ‘о’|0 0 ‘ ’|11 3 ‘в’|0 0 ‘е’|7 1 ‘к’|0 0 ‘у’|6 1 ‘и’|4 6 ‘д’|2 1 ‘н’|5 1 ‘е’|4 5 ‘о’|4 1 ‘к’|5 0 ‘и’

Рисунок 24. Закодированный текст

ВАРИАНТ 4.4 Алгоритм Лемпела – Зива (Lempel – Ziv) LZSS

Задание

Реализовать программу ***lzsscompress*** сжатия текстовых файлов на английском языке алгоритмом Зива-Лемпела. Сжатие осуществляется с аргументом

командной строки *–c* (compress), а распаковка – с аргументом *-d* (decompress). Опция *-o* указывает имя выходного файла. Например:

$ **lzsscompress** -c -o file.lzss file.txt	# сжатие file.txt в file.lzss
$ **lzsscompress** -d -o file1.txt file.lzss	# распаковка file.lzss в file1.txt

Критерии оценки

- **Оценка «хорошо»**: реализован алгоритм сжатия, для записи кодов в файл используются структуры данных.
- **Оценка «отлично»**: можно задать любой размер словаря и буфера, для формирования файлового элемента используется битовый массив (как описано в общей информации к разделу 4).

Указание к выполнению задания

Алгоритм LZSS разработан на базе алгоритма LZ77 и отличается от него форматом закодированной информации. Код, выдаваемый LZSS, начинается с однобитного префикса, отличающего код от незакодированного символа. Код состоит из пары: смещение и длина, такими же как и для LZ77. В LZSS окно сдвигается ровно на длину найденной подстроки или на 1, если не найдено вхождение подстроки из буфера в словарь. Длина подстроки в LZSS всегда больше нуля, поэтому длина двоичного кода для длины подстроки - это округленный до большего целого двоичный логарифм от длины буфера.

Рассмотрим алгоритм на примере. Пусть нам необходимо закодировать строку: «про проверку и проведение проводки». При кодировании будем использовать окно размером 23 символа, где первые 15 символов будут словарем, а следующие 8 – буфером.

Словарь															Буфер								Код
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	1	2	3	4	5	6	7	8	
															п	р	о		п	р	о	в	<0,0,‘п’>
														п	**р**	о		п	р	о	в	е	<0,0,‘р’>
													п	р	**о**		п	р	о	в	е	р	<0,0,‘о’>
												п	р	о		п	р	о	в	е	р	к	<0,0,‘ ’>
											п	**р**	**о**		**п**	**р**	**о**	в	е	р	к	у	<1,11,3>
								п	р	О		п	р	о	**в**	е	р	к	у		и		<0,0,‘в’>
							п	р	о		п	р	о	в	**е**	р	к	у		и		п	<0,0,‘е’>
						п	**р**	о		П	р	о	в	е	**р**	к	у		и		п	р	<1,7,1>
					п	р	о		п	Р	о	в	е	р	**к**	у		и		п	р	о	<0,0,‘к’>
				п	р	о		п	р	О	в	е	р	к	**у**		и		п	р	о	в	<0,0,‘у’>
			п	р	о		п	р	о	В	е	р	к	у		и		п	р	о	в	е	<1,6,1>
		п	р	о		п	р	о	в	Е	р	к	у		**и**		п	р	о	в	е	д	<0,0,‘и’>
	п	р	о		п	р	о	в	е	Р	к	у		и		п	р	о	в	е	д	е	<1,4,6>
р	о	в	е	р	к	у		и		П	р	о	в	е	**д**	е	н	и	е		п	р	<0,0,‘д’>
о	в	**е**	р	к	у		и		п	Р	о	в	е	д	**е**	н	и	е		п	р	о	<1,2,1>
в	е	р	к	у		и		п	р	О	в	е	д	е	**н**	и	е		п	р	о	в	<0,0,‘н’>
е	р	к	у		**и**		п	р	о	В	е	д	е	н	**и**	е		п	р	о	в	о	<1,5,1>
р	к	у		и		п	р	о	в	**Е**	д	е	н	и	**е**		п	р	о	в	о	д	<1,10,1>
к	у		и		**п**	**р**	**о**	**в**	е	Д	е	н	и	е		**п**	**р**	**о**	**в**	о	д	к	<1,4,5>
п	р	**о**	в	е	д	е	н	и	е		п	р	о	в	**о**	д	к	и					<1,2,1>

р	о	в	е	**д**	е	н	и	е		П	р	о	в	о	**д**	к	и						<1,4,1>
о	в	е	д	е	н	и	е		п	Р	о	в	о	д	**к**	и							<0,0,‘к’>
в	е	д	е	н	**и**	е		п	р	О	в	о	д	к	**и**								<1,5,1>

Закодированный текст «про проверку и проведение проводки» представлен на рисунке 25.

0 0 ‘п’|0 0 ‘р’|0 0 ‘о’|0 0 ‘ ’|1 11 3|0 0 ‘в’|0 0 ‘е’|1 7 1|0 0 ‘к’|0 0 ‘у’|1 6 1|0 0 ‘и’|1 4 6|0 0 ‘д’|1 2 1|0 0 ‘н’|

1 5 1|1 10 1|1 4 5|1 2 1|1 4 1|0 0 ‘к’|1 5 1|

Рисунок 25. Закодированный текст

ВАРИАНТ 4.5 Алгоритм Лемпела – Зива (Lempel – Ziv) LZ78

Задание

Реализовать программу ***lz78compress*** сжатия текстовых файлов на английском языке алгоритмом Зива-Лемпела. Сжатие осуществляется с аргументом командной строки *-c* (compress), а распаковка – с аргументом *-d* (decompress). Опция *-o* указывает имя выходного файла. Вызов программы с аргументами командрой строки может выглядить следующим образом:

$ **lz78compress** -c -o file.lz78 file.txt	# сжатие file.txt в file.lz78
$ **lz78compress** -d -o file1.txt file.lz78	# распаковка file.lz78 в file1.txt

Критерии оценки

- **Оценка «хорошо»**: реализован алгоритм сжатия, размер словаря 65536 элементов, при переполнении словаря он полностью сбрасывается.
- **Оценка «отлично»**: реализован алгоритм сжатия, размер словаря может быть задан пользователем, при переполнении словаря удаляется наименее часто используемая фраза. Дополнительным плюсом будет использование кодов переменной длины (размер ссылки на словарь сначала 1 байт, когда 1 байт переполняется – 2 байта и т.д.).

Указание к выполнению задания

В алгоритме LZ78 используется словарь из уже просмотренных фраз, который в процессе сжатия постоянно пополняется. При старте алгоритма этот словарь содержит только одну пустую строку (строку длины ноль). Алгоритм считывает символы сообщения до тех пор, пока накапливаемая подстрока входит целиком в одну из фраз словаря. Как только эта строка перестанет соответствовать хотя бы одной фразе словаря, алгоритм генерирует код, состоящий из индекса строки в словаре, которая до последнего введенного символа содержала входную строку, и символа, нарушившего совпадение. Затем в словарь добавляется полученная подстрока:

X = <номер строки из словаря><символ, нарушивший совпадение>.

Например, пусть мы кодируем следующий текст:”ппр”. Заносим в словарь первый символ ‘п’ и начинаем поиск новой фразы, для этого считываем следующий символ (‘п’). Поскольку он совпадает с фразой из словаря, считываем очередной символ. Получившейся фразы ‘пр’ нет в словаре – в соответствии с алгоритмом генерируем код для нее и записываем ее в словарь.

Если словарь уже заполнен, то из него предварительно удаляют наиболее редко используемую фразу либо очищают полностью.

Ключевым для размера получаемых кодов является размер словаря во фразах, потому что каждый код при кодировании по методу LZ78 содержит номер фразы в словаре. Из последнего следует, что эти коды имеют постоянную длину, равную округленному в большую сторону двоичному логарифму размера словаря плюс количество бит в байт-коде ASCII.

Рассмотрим алгоритм на примере. Пусть нам необходимо закодировать строку: «про проверку и проведение проводки». Пошаговый алгоритм кодирования представлен на рисунке 26.

Строка	Фраза, добавляемая в словарь	Код
про проверку и проведение проводки	0 - ‘’	
про проверку и проведение проводки	1 - ‘п’	<0,‘п’>
п**р**о проверку и проведение проводки	2 - ‘р’	<0,‘р’>
пр**о** проверку и проведение проводки	3 - ‘о’	<0,‘о’>
про проверку и проведение проводки	4 - ‘ ’	<0,‘ ’>
про **пр**оверку и проведение проводки	5 - ‘пр’	<1,‘р’>
про пр**ов**еркуи проведение проводки	6 - ‘ов’	<3,‘в’>
про пров**е**рку и проведение проводки	7 - ‘е’	<0,‘е’>
про прове**рк**у и проведение проводки	8 - ‘рк’	<2,‘к’>
про проверк**у** и проведение проводки	9 - ‘у’	<0,‘у’>
про проверку **и** проведение проводки	10 - ‘ и’	<4,‘и’>
про проверку и **п**роведение проводки	11 - ‘ п’	<4,‘п’>
про проверку и п**ро**ведение проводки	12 - ‘ро’	<2,‘о’>
про проверку и пров**е**дение проводки	13 - ‘в’	<0,‘в’>
про проверку и прове**д**ение проводки	14 - ‘ед’	<7,‘д’>
про проверку и провед**ен**ие проводки	15 - ‘ен’	<7,‘н’>
про проверку и проведени**е** проводки	16 - ‘и’	<0,‘и’>
про проверку и проведение проводки	17 - ‘е ’	<7,‘ ’>
про проверку и проведение **про**водки	18 - ‘про’	<5,‘о’>
про проверку и проведение пров**о**дки	19 - ‘во’	<13,‘о’>
про проверку и проведение провод**к**и	20 - ‘д’	<0,‘д’>
про проверку и проведение проводк**и**	21 - ‘к’	<0,‘к’>
про проверку и проведение проводки	22 - ‘и’	<0,‘и’>

0,‘п’|0,‘р’|0,‘о’|0,‘ ’|1,‘р’|3,‘в’|0,‘е’|2,‘к’|0,‘у’|4,‘и’|4,‘п’|2,‘о’|0,‘в’|7,‘д’|7,‘н’|0,‘и’|7,‘ ’|5,‘о’|13,‘о’|0,‘д’|0,‘и’

Рисунок 26. Пошаговый алгоритм кодирования

При декодировании данной строки словарь формируется динамически. Из входных данных считываются пары <номер фразы-начала из словаря, завершающий символ>, в выходные данные дописывается конкатенация этих двух элементов. Так например, пусть в качестве входных данных получен набор кодов: <0,‘п’><1,‘р’>. Считываем первый код - <0,‘п’>, поскольку номер фразы-начала из словаря 0, что соответствует пустой строке, после конкатенации в словарь под номером 1 добавляется фраза ‘п’. Считываем очередной код-<1,‘р’>. В нем номер фразы-начала из словаря равен 1, что соответствует фразе ‘п’, в словарь под номером 2 мы добавляем конкатенацию этой фразы и символа ‘р’ – фразу ‘пр’

Фрагмент закодированной строки	Трактовка кода	Фраза, добавляемая в словарь	Раскодированная строка
<0,‘п’>	-	1 - ‘п’	‘п’
<0,‘р’>	-	2 - ‘р’	‘пр’
<0,‘о’>	-	3 - ‘о’	‘про’
<0,‘ ’>	-	4 - ‘ ’	‘про ’
<1,‘р’>	1→п:п+р	5 - ‘пр’	‘про пр’
<3,‘в’>	3→о:о+в	6 - ‘ов’	‘про пров’
<0,‘е’>	-	7 - ‘е’	‘про прове’
<2,‘к’>	2→р:р+к	8 - ‘рк’	‘про проверк’
<0,‘у’>	-	9 - ‘у’	‘про проверку’
<4,‘и’>	4→ : +и	10 - ‘ и’	‘про проверку и’
<4,‘п’>	4→ : +п	11 - ‘ п’	‘про проверку и п’
<2,‘о’>	2→р:р+о	12 - ‘ро’	‘про проверку и про’
<0,‘в’>	-	13 - ‘в’	‘про проверку и пров’
<7,‘д’>	7→е:е+д	14 - ‘ед’	‘про проверку и провед’
<7,‘н’>	7→е:е+н	15 - ‘ен’	‘про проверку и проведен’
<0,‘и’>	-	16 - ‘и’	‘про проверку и проеведени’
<7,‘ ’>	7→е:е+	17 - ‘е ’	‘про проверку и проведение ’
<5,‘о’>	5→пр:пр+о	18 - ‘про’	‘про проверку и проведение про’
<13,‘о’>	13→в:в+о	19 - ‘во’	‘про проверку и проведение прово’
<0,‘д’>	-	20 - ‘д’	‘про проверку и проведение провод’
<0,‘к’>	-	21 - ‘к’	‘про проверку и проведение проводк’
<0,‘и’>	-	22 - ‘и’	про проверку и проведение проводки’

Рисунок 27. Пошаговый процесс декодирования сообщения

ВАРИАНТ 4.6 Алгоритм Лемпела – Зива – Велча, LZW

Задание

Реализовать программу ***lzwcompress*** сжатия текстовых файлов на английском языке алгоритмом Лемпела – Зива – Велча. Сжатие осуществляется с аргументом командной строки *-c* (compress), а распаковка – с аргументом *-d* (decompress). Опция *-o* указывает имя выходного файла. Вызов программы с аргументами командной стоки может выглядить следующим образом:

$ **lzwcompress** -c -o file.lzw file.txt	# сжатие file.txt в file.lzw
$ **lzwcompress** -d -o file1.txt file.lzw	# распаковка file.lzw в file1.txt

Критерии оценки

- **Оценка «хорошо»**: реализован алгоритм сжатия, размер словаря 65536 элементов, при переполнении словаря он полностью сбрасывается (за исключением односимвольных фраз).
- **Оценка «отлично»**: реализован алгоритм сжатия, размер словаря может быть задан пользователем, при переполнении словаря удаляется наименее часто используемая фраза. Дополнительным плюсом будет использование кодов переменной длины (размер ссылки на словарь сначала 1 байт, когда 1 байт переполняется – 2 байта и т.д.).

Указание к выполнению задания

Алгоритм LZW основан на алгоритме LZ78, он используется в программе сжатия compress ОС Unix, а также в формате GIF.

Также как и в LZ78, в алгоритме LZW для декомпрессии не нужно сохранять всю таблицу кодов в файл для распаковки. Алгоритм построен таким образом, что весь словарь можно восстановить, зная все односимвольные фразы и пользуясь потоком кодов. Основные этапы алгоритма LZW:

1)Инициализация словаря всеми возможными односимвольными фразами (обычно 256 символами расширенного ASCII). Инициализация новой фразы ω первым символом сообщения.

2)Считать очередной символ *k* из кодируемого сообщения.

3)Если конец сообщения, записать код для ω в выходной поток, кодирование завершено.

4)Если фраза ω*k* уже есть в словаре, присвоить ω = ω*k* и перейти на шаг 2. Иначе, записать код для ω в выходной поток, добавить ω*k* в словарь, присвоить ω = *k* и перейти на шаг 2.

Для LZW ключевым для размера получаемых кодов является размер словаря во фразах: LZW-коды имеют постоянную длину, равную округленному в большую сторону двоичному логарифму размера словаря. При переполнении словаря, из него удаляют либо наиболее редко используемую фразу, либо все фразы, отличающиеся от одиночного символа.

Рассмотрим алгоритм на примере. Пусть нам необходимо закодировать строку: «про проверку и проведение проводки». На рисунке 28 жирным выделено ω*k*, причем ω дополнительно выделено цветом фона.

Строка	Фраза, добавляемая в словарь	Результирующий код
	1 ‘п’ 2 ‘р’ 3 ‘о’ 4 ‘в’ 5 ‘е’ 6 ‘к’ 7 ‘у’ 8 ‘ ’ 9 ‘и’ 10 ‘д’ 11 ‘н’	
про проверку и проведение проводки	12 ‘пр’	<1>
п**ро** проверку и проведение проводки	13 ‘ро’	<1,2>
про проверку и проведение проводки	14 ‘о ’	<1,2,3>
про **п**роверку и проведение проводки	15 ‘ п’	<1,2,3,8>
про **пр**оверку и проведение проводки	16 ‘про’	<1,2,3,8,12>
про пр**ов**ерку и проведение проводки	17 ‘ов’	<1,2,3,8,12,3>
про пров**е**рку и проведение проводки	18 ‘ве’	<1,2,3,8,12,3,4>
про пров**ер**ку и проведение проводки	19 ‘ер’	<1,2,3,8,12,3,4,5>
про пров**ерк**у и проведение проводки	20 ‘рк’	<1,2,3,8,12,3,4,5,2>
про пров**ерку** и проведение проводки	21 ‘ку’	<1,2,3,8,12,3,4,5,2,6>
про пров**еру** и проведение проводки	22 ‘у ’	<1,2,3,8,12,3,4,5,2,6,7>
про проверку **и** проведение проводки	23 ‘ и’	<1,2,3,8,12,3,4,5,2,6,7,8>
про проверку **и** проведение проводки	24 ‘и ’	<1,2,3,8,12,3,4,5,2,6,7,8,9>
про проверку и **пр**оведение проводки	25 ‘ пр’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15>
про проверку и п**ров**едение проводки	26 ‘ров’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13>
про проверку и пров**ед**ение проводки	27 ‘вед’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18>
про проверку и прове**де**ние проводки	28 ‘де’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10>
про проверку и провед**ен**ие проводки	29 ‘ен’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5>
про проверку и проведе**ни**е проводки	30 ‘ни’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11>
про проверку и проведен**ие** проводки	31 ‘ие’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9>
про проверку и проведение проводки	32 ‘е ’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5>
про проверку и проведение **про**водки	33 ‘ про’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5,25>
про проверку и проведение пр**ово**дки	34 ‘ово’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5,25,7>
про проверку и проведение пров**од**ки	35 ‘од’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5,25,7,3>
про проверку и проведение провод**к**и	36 ‘дк’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5,25,7,3,10>
про проверку и проведение провод**ки**	37 ‘ки’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5,25,7,3,10,6>
про проверку и проведение проводк**и**	38 ‘и’	<1,2,3,8,12,3,4,5,2,6,7,8,9,15,13,18,10,5,11,9,5,25,7,3,10,6,9>

Рисунок 28. Пошаговый процесс кодирования сообщения

СПИСОК ЛИТЕРАТУРЫ

1. Подбельский В.В, Фомин С.С. Программирование на языке Си: Учеб. пособие. – 2-е доп. изд. – М.: Финансы и статистика, 2004. – 600 с.
2. Белецкий Я. Энциклопедия языка Си. – М.: Мир, 1992.
3. Богатырев А. Хрестоматия по программированию на Си в Unix. – 1992
4. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона / Пер. с англ. Ткачев Ф.В. – М.: ДМК Пресс, 2012 г. – 272 с., ил.
5. Шень А. Программирование. Теоремы и задачи – М.: МЦНМО, 2011 г. – 296 с. – ISBN 978-5-94057-696-9.
6. Керниган Б., Ритчи Д., Фьюер А. Язык программирования Си. Задачи по языку Си: Пер. с англ. – М.: Финанасы и статистика, 1985.
7. Кормен Т., Лейзерсон Ч., Ривест Р. Штайн К. Алгоритмы: построение и анализ, 2-е изд.: Пер. с англ. – М.: Издательский дом "Вильямс", 2012. – 1296 с.: ил. – Парал. тит. англ. ISBN 978–5–8459–0857–5
8. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ, М.:МЦНМО, 2002, 960 с.
9. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск (The Art of Computer Programming): Пер. с англ. – М.: Издательский дом "Вильямс", 2012 г. – 824 с. – ISBN 978-5-8459-0082-1

Артем Юрьевич Поляков
Александра Юрьевна Полякова
Евгения Николаевна Перышкова

Программирование

Практикум

Редактор: О.В. Молдованова
Корректор: В.В.Сиделина

Подписано в печать 25.06.2012г.
Формат бумаги 60 х 84/16, отпечатано на ризографе, шрифт № 10,
изд. л. 8,4 заказ № 40 , тираж – 100 экз., СибГУТИ.
630102, г. Новосибирск, ул. Кирова, д. 86
54