ФГАОУ ВПО «УрФУ имени первого президента России Б.Н.Ельцина»
Институт математики и компьютерных наук

Кафедра математической экономики

Методы архивирования

Курсовая работа

студента 2 курса группы КБ-201
Кулакова Владислава Сергеевича

Научный руководитель
Асанов Магаз Оразкимович

Екатеринбург
2015

Теория
Различные Методы Архивирования

1. Введение

Архивация данных - это уменьшение физических размеров файлов, в которых хранятся данные. В настоящее время из-за быстрого увеличения памяти устройств архивирование потеряло свой первоочередной смысл в быту, однако до сих пор оно очень востребовано во многих сферах.

Активное использование архивирования можно наблюдать в сетевом трафике, так как передавать большое количество информации удобно именно в архивированном виде, слишком уж не хочется сидеть и ждать неделю, пока большой файл будет перекачиваться с одного устройства на другое, если он может быть уменьшен и передан быстрее.

Также архивирование применяется в разных базах данных, серверах, где даже очень большие накопители не в состоянии справиться с размером информации.

Ну и нельзя забывать о ценных данных, которые необходимо хранить несколькими копиями. Хранение их архивированных копий намного выгоднее, чем иметь несколько одинаковых файлов, даже если памяти много.

Так что архивирование до сих пор очень востребованное и нужное преобразование информации.

Выделяется два типа архивирования данных:

1) **Сжатие с потерями.** Наиболее применимое из них, конечно, сжатие с потерями, хотя многие даже не замечают, что они его используют. Такое сжатие применяется в большинстве мультимедийных форматов:

«mp3», «ogg» ... - для хранения музыкальных файлов;

«mpeg1», «mpeg2», «mpeg4» … - для сжатия видео,

«jpeg» … - для сжатия изображений.

Подобные способы архивирования используются для чувствительной информации (слуховой, зрительной …), когда влияние на значение этой информации не зависит от её полноты и точности.

К примеру: Человек слышит звук в диапазоне от 16—20 Гц до 15—20 кГц. Естественно, удаление частот выше или ниже диапазона приведёт к потере информации, но слух человека не почувствует разницы в звуковых дорожках до и после удаления, так как просто физически не способен её уловить.

2) **Сжатие без потерь.** Именно эти алгоритмы будут рассматриваться в курсовой работе. Эти алгоритмы имеют большую значимость в сфере архивирования, так как их главным свойством является сохранение точности данных. Именно эти алгоритмы используют для архивирования текстовой информации, которая при потере данных может потерять всякий смысл.

В алгоритмах без потерь выделяют два подхода:

а) **основан на повторах данных** — так как иногда данные имеют тенденцию повторяться, существуют алгоритмы, сжимающие информацию по повторяемым данным;

б) **основан на генерировании битовых последовательностей** — так как зачастую информация не расходует все возможности кодировки, тем более что кодировки, как правило, статичные, а информация распределяется неравномерно. Методы генерации битовых последовательностей очень актуальны, ведь они позволяют задействовать весь спектр предоставленных последовательностей и к тому же расходуют их в нужных пропорциях.

Алгоритмы основанные на анализе повторов

2. Алгоритм **RLE (run-length encoding)**

Самый простой алгоритм из этой серии. Основывается на кодировании длинных последовательностей символов.

Алгоритм:

1) Возьмём последовательность:
AAAAAAADDDDDAAAAAAAAAKKKKKKCCFFF
32 символа

2) Посчитаем количество повторяющихся символов:
7 букв A, 5 букв D, 9 букв A, 6 букв K, 2 буква C, 3 буквы F

3) Уберём слово «букв» из нашего подсчёта и получим легко декодируемую последовательность:
7A,5B,9A,6K,2C,3F

4) Дальше можно выделить два подхода в зависимости от тонкостей кодирования вашей информации.

а) Мы можем убрать «,» и кодировать и декодировать информацию, как {число букв}{буква}, получим:
7A5B9A6K2C3F
(причём для кодирования нам не важно число 0, так что можно сдвинуть количество букв на n+1), то есть: 6A5B8A5K1C2F
Итог: 12 символов, сжатие = 37,5%

б) Подход второй использует управляющий символ, пусть это будет «,» (в теории может выступать любой, но желательно наименее встречаемый, или вообще не используемый), получим вид:
{управляющий символ}{число букв}{буква}
Теперь можно убрать запятую у всех символов входящих не более 3 раз, и сдвинуть на n+3, получим:
,4A,2B,6A,3KCCFFF
Итог: 17 символов, сжатие~53.1%

5) Для подхода с управляющим символом может быть нужно использование самого символа, тогда придётся экранировать и сам символ, но со смещением(n+1):
,0, - закодирована одна запятая.

Вывод: RLE кодирование не годится для литературных текстов, но оно нашло применение для кодирования простых графических изображений, таких как иконки.

3. Алгоритм **LZ77 (Abraham Lempel и Jacob Ziv 1977 год)**

Можно сказать, что алгоритмы семейства LZ* представляют собой более сложное обобщение простого и интуитивного способа сжатия данных, используемого в RLE.

Алгоритм использует две составляющие:

1) **Принцип «Скользящего окна» -** принцип, который использует уже встречавшуюся раннее информацию, то есть все последующие повторы данной информации ссылаются на эту.

2) **Механизмом кодирования совпадений** — (совпадение — 2 и более равных совпадающих элементов в последовательности) кодируются парой (смещение, длина)

Алгоритм:

1) Возьмём скороговорку:
Ripe·white·wheat·reapers·reap·ripe·white·wheat·right (52 символа)

2) Идём с начала предложения и ищем совпадения, найдя совпадение делаем ссылку на предыдущие в виде пары (смещение, длина):
смещение=индекс_второго - индекс_первого
(пошаговые преобразования)

1. Rip **e·wh** it **e·wh** eat·reapers·reap·ripe·white·wheat·right
(совпадение 4 символ и 10 символ, длины 4, (10-4,4))
2. Rip **e·wh** it(6,4) **e·wh** eat·reapers·reap·ripe·white·wheat·right
3. Ripe·whit(6,4)e·wh **ea** t·r **ea** pers·reap·ripe·white·wheat·right
4. Ripe·whit(6,4)e·wh **ea** t·r(5,2) **ea** pers·reap·ripe·white·wheat·right
5. Ripe·whit(6,4)e·wheat **·r**(5,2)**eap** ers **·reap** ·ripe·white·wheat·right
6. Ripe·whit(6,4)e·wheat **·r**(5,2)**eap** ers (8,5)**·reap** ·ripe·white·wheat·right
7. Ripe·whit(6,4)e·wheat·r(5,2)eapers(8,5) **·r** eap **·r** ipe·white·wheat·right
8. Ripe·whit(6,4)e·wheat·r(5,2)eapers(8,5) **·r** eap (5,2)**·r** ipe·white·wheat·right
9. R **ipe·whit**(6,4)**e·wheat·r**(5,2)eapers(8,5)·reap(5,2)·r **ipe·white·wheat·r** ight
10.R**ipe·whit**(6,4)**e·wheat·r**(5,2)eapers(8,5)·reap(5,2)·r(30,17)**ipe·white·wheat·r**ight

3) Убираем лишние повторы и получаем:
Ripe·whit(6,4)eat·r(5,2)pers(8,5)(5,2)(30,17)ight

4) Смещение и длина естественно определяется конкретными условиями. Для заданной последовательности можно взять вторую половину ASCII с кодами(128-255), чтобы они не пересекались с кодами символов. Также удобно разбить длину и смещение на два символа, и так как длина не может быть больше смещения, то её поставим на первое место, смещение на второе, и для смещения будем использовать весь байт (256).
Так (6,4) преобразуется в байты \x8406, где \x84 = 132 «132-128=4» \x06=6
Ripe·whit\x8406(6,4)eat·r\x8205(5,2)pers\x8508(8,5)\x8205(5,2)\x911E(30,17)ight
Ripe·whit\x8406eat·r\x8205pers\x8508\x8205\x911Eight
(32 символа, сжатие~61%)

//Так как в таком представлении не имеет смысла сжимать два символа, можно увеличить за счёт них длину, сдвинув её на n+2, как делали в RLE.

5) Существуют модификации алгоритма, когда длина может превышать смещение, к примеру, строку: «hahahahaha» можно закодировать как ha(2,8), зацикливаясь на нужном нам отрезке.

Для алгоритма LZ77 не сразу можно понять, как же декодировать эту мистическую строчку обратно, так что приведём процесс декодирования нашей фразы:

1) Берём закодированную строку:

Ripe·whit(6,4)eat·r(5,2)pers(8,5)(5,2)(30,17)ight

2) Пошагово восстанавливаем её, то есть копируем символы и для пары (смещение, длина) копируем символы из восстановленной строки:

1.**Ripe·whit** (6,4)eat·r(5,2)pers(8,5)(5,2)(30,17)ight
2.Rip**e·wh**it **(6,4)**eat·r(5,2)pers(8,5)(5,2)(30,17)ight
3.Ripe·whit**e·wh** eat·r(5,2)pers(8,5)(5,2)(30,17)ight
4.Ripe·white·wh**eat·r** (5,2)pers(8,5)(5,2)(30,17)ight
5.Ripe·white·wh**ea**t·r **(5,2)**pers(8,5)(5,2)(30,17)ight
6.Ripe·white·wheat·r**ea** pers(8,5)(5,2)(30,17)ight
7.Ripe·white·wheat·rea**pers** (8,5)(5,2)(30,17)ight
8.Ripe·white·wheat**·reap**ers **(8,5)**(5,2)(30,17)ight
9.Ripe·white·wheat·reapers**·reap** (5,2)(30,17)ight
10.Ripe·white·wheat·reapers**·r**eap **(5,2)**(30,17)ight
11.Ripe·white·wheat·reapers·reap**·r** (30,17)ight
12.R**ipe·white·wheat·r**eapers·reap·r **(30,17)**ight
13.Ripe·white·wheat·reapers·reap·r**ipe·white·wheat·r** ight
14.Ripe·white·wheat·reapers·reap·ripe·white·wheat·r**ight**

Так происходит декодирование строки закодированной алгоритмом LZ77.

Вывод: Алгоритм LZ77 подходит для кодирования текстовой информации, так как в отличие от RLE он ищет схожие последовательности, однако размер текстовой информации должен быть довольно большой, чтобы в нём было много повторений. (Также подходит для кодирования скороговорок, ведь в них часто повторяется текст)

Также существует множество других алгоритмов анализа повторов, но они основаны на этих же идеях. К примеру: Преобразование Барроуза — Уилера переводит текст в удобный для сжатия по RLE-подобным, LZ78 и LZW и т. д. - алгоритмы ориентируются на данные, которые только будут получены (модернизация алгоритма LZ77) ...

Алгоритмы основанные на генерации битовых последовательностей

4. Алгоритм **Шеннона — Фано**

Алгоритмы, основанные на генерации битовых последовательностей используют для построения **коды переменной длины** (так как в основном кодировки статические, и даже если это не так, то далеко не все символы используются для построения отдельного текста) и **частоты встречаемости символов**. Естественно, возникла идея сгенерировать такие коды символов, чтобы размер информации уменьшился, а содержательность осталась.

Все алгоритмы этого типа требуют вначале построения **частот встречаемости символов.**

Наиболее простой алгоритм этой группы, это алгоритм Шеннона—Фано.

Алгоритм:

1) Возьмём строку:

AAAAAADDDDDDAAAAKKKKKKKKKFFCCFFF

32 символа

2)Строим таблицу встречаемости символов:

Буквы	A	D	K	C	F
Количество	10	6	9	2	5

Так как символов 5, то наименьший статичный код строился бы из 3 бит 2^2=4<5<2^3, размер в битах 3*32 = 96 бит, а в ASCII занял бы 32 Байта

3)Упорядочим таблицу по не возрастанию. И объединяем, чтобы начать преобразования.

Буквы	A	K	D	F	C
Количество	10	9	6	5	2

4) Стартовый шаг, 1 блок из всех букв, постепенно разделяя блоки на 2 по убыванию количества букв так, чтобы разница сумм была наименьшая, преобразуем нашу таблицу вплоть до разделения на все символы. Получается построение двоичного дерева. При разделении присваиваем буквам из левого блока 0, из правого 1.

Блок	AKDFC				
Количество	32				
Блоки	AK — 0		DFC — 1		
Количество	10+9 = 19		6+5+2=13		
Блоки	A — 00	K — 01	D — 10	FC — 11	
Количество	10	9	6	5+2=7	
Блоки	A — 00	K — 01	D — 10	F — 110	C — 111
Количество	10	9	6	5	2

5) Таким образом, построены коды символов для наших букв

A — 00, K — 01, D — 10, F — 110, C — 111

Эти коды относятся к префиксным, так как ни один не начинается с кода другого.
Тем самым кодируемая последовательность может быть однозначно декодируема.

Итак, последовательность:
AAAAAADDDDDDAAAAKKKKKKKKKFFCCFFF
Будет записана как:
00000000000010101010100000000001010101010101011101101111111110110110
Разделим по байтам:
00000000 = \x00
00001010 = \x0A
10101000 = \xA8
00000001 = \x01
01010101 = \x55
01010101 = \x55
11011011 = \xDB
11111101 = \xFD
10110
Получаем последовательность из 8*8+5 = 69 бит, 8.625 Байта
Сжатие относительно 3-битного кода ~ 72%
относительно ASCII ~ 27-28%
Для точного разбиения на байты, в построение можно ввести символ конца данных.
Тогда все оставшиеся биты просто заполним 0 (или можно передавать длину последовательности).

Вывод: Алгоритм Шеннона-Фано довольно хорошо сжимает текстовую информацию, но существуют последовательности, на которых строятся неоптимальные коды, поэтому для построения используется более эффективный алгоритм Хаффмана.

4. Алгоритм **кодирования Хаффмана**

Алгоритм Хаффмана очень похож на предыдущий алгоритм Шеннона — Фано, только построение дерева идёт не с корня, а с листьев.

Алгоритм:

1) Возьмём последовательность (из предыдущего алгоритма):

AAAAAADDDDDDAAAAKKKKKKKKKFFCCFFF

32 символа

2) Строим таблицу частот букв:

Буквы	A	D	K	C	F
Количество	10	6	9	2	5

Так же размер в статическом 3-битном коде 96 бит, в ASCII 32 байта.

3)Стартовый шаг алгоритма: от частот букв строим листья дерева по правилу: Берём два блока наименьшего количества, левому присваиваем 0, правому присваиваем 1, удаляем блок из просматриваемых и добавляем суммарный блок. Так пока не дойдём до корня, потом восстанавливаем коды.

Блоки	A	K	D	**F — 0**	**C — 1**
Количество	10	9	6	**5**	**2**
Блоки	A	K	**D — 0**	**FC — 1**	
Количество	10	9	**6**	**5+2=7**	
Блоки	**A — 0**	**K — 1**	DFC		
Количество	**10**	**9**	6+5+2=13		
Блоки	**AK — 0**		**DFC — 1**		
Количество	**10+9 = 19**		**6+5+2=13**		
Блоки	AKDFC				
Количество	10+9+6+5+2=32				

4) Восстанавливаем коды букв в порядке, идущем от корня:

A — 00, K — 01, D — 10, F — 110, C — 111

Коды совпали с кодами Шеннона — Фано, поэтому статистические данные те же.

Получаем последовательность из 8*8+5 = 69 бит, 8.625 Байта

Сжатие относительно 3-битного кода ~ 72%

относительно ASCII ~ 27-28%

В большинстве случаев коды этих двух алгоритмов одинаковы, но для таблицы:

Буквы	A	K	D	F	C
Количество	9	6	5	3	2

Коды будут различны, у Хаффмана они не изменятся, а у Шеннона — Фано будут:

A — 0, K — 01, D — 011, F — 0111, C — 1111

Конечно, коды Хаффмана будут оптимальнее ведь:

Длина Хаффмана = 9*2+6*2+5*2+3*3+2*3 = 55

Длина Шеннона — Фано= 9*1+6*2+5*3+3*4+2*4 = 56

Вывод: Алгоритм Хаффмана работает лучше, чем алгоритм Шеннона — Фано, поэтому именно он активно применяется для сжатия фото- и видеоизображений, основанного на генерации битовых последовательностей.

5. Небольшое обобщение.

Мной были рассмотрены несколько основных алгоритмов сжатия информации без потерь, алгоритмы работают по двум основным подходам, с анализом повторов информации и с изменением размера, выделяемого под единицу информации.

Все алгоритмы имеют свои преимущества и свои недостатки: как алгоритм LZ77 может сжимать лучше Хаффмана, если там много символов, но и много повторов, так и обратно алгоритм Хаффмана будет выигрывать на малом количестве повторов. Алгоритм RLE бесспорно обойдёт всех в сжимании огромных количеств повторов одного символа. Каждый алгоритм подходит для определённых нужд.

Также существуют более сложные алгоритмы и в той, и в другой группе, которые улучшают разные аспекты алгоритмов.

Основной проблемой алгоритмов генерации последовательностей является перенос самой таблицы генерации, её надо либо генерировать отдельно, либо переносить вместе с архивом, что увеличивает объём данных. Но сжатие этих алгоритмов, как правило, выше, чем у других.

В анализе повторов очень мешает ограниченность смещения и длины, да и недостаток самих повторов, поэтому их невыгодно применять для цифровых файлов, однако очень удобно для текстовой информации, ведь для языков характерны одинаковые последовательности символов и одинаковые созвучия.

Выделим ещё третью группу алгоритмов, они используют возможности обоих подходов. Так, например, можно использовать таблицу, генерируемую по частоте встречаемости пар символов и использовать не вошедшие символы кодировки для замещения пар, тем самым тоже получить сжатие без потерь.
Или, к примеру, алгоритм Deflate, который использует комбинацию алгоритма LZ77 и алгоритма Хаффмана.

Практика

Реализация метода архивирования.

Задача 1307 «Архиватор» из архива **Timus Online Judge** (Приложение №1)

Для реализации этой задачи первым делом надо выбрать алгоритм архивирования, но, так как это задача из категории ACM, её надо решать наиболее доступным способом. Поэтому я приведу алгоритм, который не рассматривался в теоретической части, и, более того, этот алгоритм интуитивен и легко реализуем.

Алгоритм, который будет реализован, использует оба подхода, но он далеко не самый эффективный по сжиманию, зато гарантирует сжатие больших объёмов литературных текстов.

Итак, алгоритм:

1) Первым делом наш алгоритм воспользуется **анализом повторов.** Возьмём какой-нибудь литературный текст и проверим вхождение пар и даже троек букв
(можно проверять и более большие вхождения).

Проверка проводилась на статье «The GNU Project»(См. Источники)
Алгоритм проверки реализован на языке Python3 (Приложение №2)
(Суть: перемещаясь от 1 до n-3 с шагом 1 по символам, записываем новые вхождения в словарь с количеством 1 и увеличиваем повторяющиеся на 1, потом сортируем и выводим в файл)

2) Получаем таблицу: (\s — пробел(space))

1.e\s	2.\st	3.th	4.s\s	5.\sa	6.re	7.he	8.t\s	9.d\s	10.in	11.\ss
1588	1119	851	802	747	728	727	700	609	570	536

3) Возьмём первые 11 вхождений (можно брать больше для увеличения сжатия)
(Также не рекомендуется брать много пересекающихся вхождений).
Добавим в наш список символы, которые по умолчанию экранируются в языке (в нашем случае C++): \n, \t, \", \', \\

4) Теперь сопоставим нашим символам номера из второй половины ASCII(128-255), которые точно не используются для представления входного текста. Достаточно символов (128-143), добавим нашу таблицу в конечный архив. Теперь мы можем гарантировать, что наш текст точно не увеличится больше, чем на объём таблицы и кода нужного для декодирования.

5)Кодирование:

a) Дан текст для архивирования.

b) Будем считывать текст по 1000 символов, так как у нас есть ограничения компилятора на длину строки.

c) Для каждой партии в 1000 символов:

запускаем цикл с шагом 1 и заменяем пары символов (шаг+1) и экранируемые символы на символы не использующиеся(по нашей таблице) и записываем их в архив, остальные записываем в архив, как есть.

d) Архив построен.

6) Декодирование:

a) Возьмём наш архив (состоит из блоков <= 1000 символов)

b) Берём каждый блок и заменяем символы по нашей таблице. Символы, которых там нет, оставляем такими же.

c) Таким образом, получаем исходный текст.

7) Алгоритмы Кодирования и Декодирования построены.

Пример на скороговорке:

Ripe·white·wheat·reapers·reap·ripe·white·wheat·right (52 символа)

Закодируем: (код = (номер в таблице из пункта 2) + 132)

Rip\133whit\133w\139a\140\138aper\136\138aprip\133whit\133w\139a\140right

(41 символ сжатие ~ 79%)

Конечно, на маленьких последовательностях этот алгоритм будет работать не слишком хорошо. Более того, было бы лучше строить таблицу по входному тексту, но всё равно этот алгоритм сжимает даже данную скороговорку. И он значительно выиграет в скорости у всех остальных (кроме, быть может, RLE).

Данный алгоритм был Реализован на языке C++

- Кодирование (Приложение № 3)
- Пример Декодируемого кода (Приложение №4)

Данный алгоритм прошёл все тесты на Timus Online Judge и, значит, является решением данной задачи.

ID	Дата	Автор	Задача	Язык	Результат проверки	№ теста	Время работы	Выделено памяти
6262604	12:10:30 30 май 2015	Forecoding	1307. Архиватор	G++ 4.9	Accepted		0.031	348 КБ

Источники

1. http://algolist.ru/compress/standard/

2. http://habrahabr.ru/post/141827/ (Алгоритмы RLE и LZ77)

3. http://neerc.ifmo.ru/wiki/index.php?title=Алгоритмы_LZ77_и_LZ78

4. **Ананий В. Левитин. Глава 9. Жадные методы: Алгоритм Хаффмана**

5. http://www.gnu.org/gnu/thegnuproject.en.html (текст для анализа)

Приложение №1

1307. Архиватор

Ограничение времени: 1.0 секунды
Ограничение памяти: 4 МБ

Как правило, жюри олимпиад по программированию норовит дать к каждой задаче «сказку», цели которой — наметить мнимую связь задачи с реальностью, а также замутить воду вокруг сути задачи, особенно если условие задачи кажется слишком простым для понимания. Однако в данной задаче сказки не будет – во-первых, потому, что условие является достаточно непростым и необычным, а во-вторых, потому что сама задача — о краткости.

Пусть дан некоторый текст. Архивом этого текста называется текст, удовлетворяющий следующим требованиям:

1. Архив является программой на одном из языков программирования, допустимых правилами соревнований.
2. Первой строкой архива является либо строка «{PAS}», либо строка «/*C*/», либо строка «//CPP».
3. Если архив скомпилировать и выполнить, то программа выдаст на стандартный выход текст, в точности совпадающий с исходным.
4. Размер архива строго меньше размера исходного текста.

Требуется написать программу, которая по заданному тексту строит его архив. При проверке посланного командой решения, архив компилируется и исполняется с использованием тех же ограничений, параметров и условий, что и решения участников. Архив не обязательно должен быть программой на том же языке, что и решение. При проверке, жюри определяет язык архива по его первой строке («{PAS}» — Pascal/Delphi, «/*C*/» — C, «//CPP» — C++).

Исходные данные

Текст длиной не менее 20000 и не более 200000 символов. Текст может содержать большие и маленькие латинские буквы, цифры, знаки препинания, пробелы, переводы строк, кавычки. Гарантируется, что все тексты, использованные в качестве тестов к задаче, являются литературными текстами на английском языке.

Примечание. Пример входа (см. ниже) является условным, поскольку не удовлетворяет требованиям литературности и минимальной длины.

Результат

Архив исходного текста.

исходные данные
12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565 7
результат
//CPP #include<iostream.h> int main() {for(int i=1;i<58;i++)cout<<i;return 0;}

Автор задачи: Идея — Леонид Волков, подготовка — Павел Егоров, Леонид Волков
Источник задачи: VIII Командный студенческий чемпионат Урала по программированию. Екатеринбург, 11-16 марта 2004 г.

Приложение №2

Анализ файла на повторы пар и троек символов

Python3

```
fin = open("learn_text.txt", encoding="ascii")
f = fin.read()
fin.close()
d = [{}, {}, {}]
s = [0, 0, 0, 0]
for i in range(len(f)-3):
    for j in range(3):
        k = f[i:i+j+1].replace("\n", "\\n")
        k = k.replace(" ", "\\s")
        if k not in d[j]:
            d[j][k] = 1
            s[j] += 1
            s[3] += 1
        else:
            d[j][k] += 1
            s[j] += 1
            s[3] += 1
fout = open("stat_file.txt", "w")
for j in range(3):
    p = []
    for i in d[j]:
        p.append((i, d[j][i]))
    while p:
        k = 0
        for i in range(1, len(p)):
            if p[i][1]>p[k][1]:
                k = i
        fout.write("  {}: {}\n".format(p[k][0], p[k][1]))
        p.remove(p[k])
    fout.write(" Итого: {}\n".format(s[j]))
fout.write("Всего: {}\n".format(s[3]))
fout.close()
```

Приложение №3

Алгоритм Архивирования
C++

```
#include <iostream>
using namespace std;

void encode(char * c)
{
    cout<<"f(\"";
    for(int i=0; c[i]!=0; ++i)
    {
        if(c[i] == '\r') continue;
        else if(c[i] == '\n') cout<<char(128);
        else if(c[i] == '\t') cout<<char(129);
        else if(c[i] == '\"') cout<<char(130);
        else if(c[i] == '\'') cout<<char(131);
        else if(c[i] == '\\') cout<<char(132);
        else if((c[i] == 'e')&&(c[i+1] == ' ')) {
                              cout<<char(133); ++i;}
        else if((c[i] == ' ')&&(c[i+1] == 't')) {
                              cout<<char(134); ++i;}
        else if((c[i] == 't')&&(c[i+1] == 'h')) {
                              cout<<char(135); ++i;}
        else if((c[i] == 's')&&(c[i+1] == ' ')) {
                              cout<<char(136); ++i;}
        else if((c[i] == ' ')&&(c[i+1] == 'a')) {
                              cout<<char(137); ++i;}
        else if((c[i] == 'r')&&(c[i+1] == 'e')) {
                              cout<<char(138); ++i;}
        else if((c[i] == 'h')&&(c[i+1] == 'e')) {
                              cout<<char(139); ++i;}
        else if((c[i] == 't')&&(c[i+1] == ' ')) {
                              cout<<char(140); ++i;}
        else if((c[i] == 'd')&&(c[i+1] == ' ')) {
                              cout<<char(141); ++i;}
        else if((c[i] == 'i')&&(c[i+1] == 'n')) {
                              cout<<char(142); ++i;}
        else if((c[i] == ' ')&&(c[i+1] == 's')) {
                              cout<<char(143); ++i;}
        else cout<<c[i];
    }
    cout<<"\");\n";
}
```

```cpp
int main()
{
    cout<<"//CPP\n#include <iostream>\n#include
<string>\nusing namespace std;\n";
    cout<<"const char*d[] = {\"\\n\", \"\\t\", \"\\\"\",
\"\\'\", \"\\\\\", \"e \", \" t\", \"th\", \"s \", \" a\",
\"re\", \"he\", \"t \", \"d \", \"in\", \" s\"};\n";
    cout<<"void f(string c){for(int i=0; i<c.length(); ++i)
{\nunsigned char p = c[i];if(p > 127) cout<<d[int(p-128)];else
cout<<p;}}";
    cout<<"int main(){\n";
    char c[1001]={0}, d;
    int i=0;
    while(cin.get(d))
    {
        c[i] = d;
        c[++i] = 0;
        if(i == 1000)
        {
            encode(c);
            i = 0;
        }
    }
    if (i) encode(c);
    cout<<"return 0;}";
    return 0;
}
```

Приложение №4

Пример Алгоритм Декодирования

C++ (кодировка cp1251)
(Текст форматирован для удобного просмотра, на самом деле большинство символов \t и \n нет в полученном коде)

```
//CPP
#include <iostream>
#include <string>
using namespace std;
const char*d[] = {"\n", "\t", "\'", "\"", "\\", "e ", " t", "th", "s ", " a", "re", "he", "t ", "d ", "in",
" s"};

void f(string c)
{
    for(int i=0; i<c.length(); ++i)
    {
        unsigned char p = c[i];
        if(p > 127) cout<<d[int(p-128)];
        else cout<<p;
    }
}

int main()
{
    f("Rip…whit…w‹aЊЉaper€Љap rip…whit…w‹aЊrightЂ");
    return 0;
}
```