Data Compression for use in the Short Messaging System

by
Måns Andersson

This thesis is presented as part of Degree of Bachelor of Science in Computer Science

Blekinge Institute of Technology
June 1, 2010
Abstract

Data compression is a vast subject with a lot of different algorithms. All algorithms are not good at every task and this thesis takes a closer look on compression of small files in the range of 100-300 bytes having in mind that the compressed output are to be sent over the Short Messaging System (SMS). Some well-known algorithms are tested for compression ratio and two of them, the Algorithm Λ, and the Adaptive Arithmetic Coding, are chosen to get a closer understanding of and then implement in the Java language. Those implementations are tested alongside the first tested implementations and one of the algorithms are chosen to answer the question "Which compression algorithm is best suited for compression of data for use in Short Messaging System messages?".
Acknowledgements

In particular, I would like to thank my advisor Mats-Ola Landbris for his continuous support and advice throughout my work. I am also grateful for Bengt Aspvall for giving me great input on my first test designs. Finally, a big thanks to NIBE AB for giving me the idea for the subject of this thesis.

Måns Andersson
Ronneby, June 2010
# Contents

1. Introduction 1  
   1.1. Problem statement and Scope 1  
   1.2. Structure of this thesis 1  

2. Compression Algorithms 3  
   2.1. Huffman Coding 3  
   2.2. Arithmetic Coding 5  
   2.3. Lempel-Ziv Coding 6  
      2.3.1. LZ77 6  
      2.3.2. LZ78 6  
      2.3.3. LZW 7  

3. Testing and Choosing 8  
   3.1. Test Design 8  
      3.1.1. Random files 8  
      3.1.2. Industry-Standard Corpus 9  
      3.1.3. Algorithm Implementations 9  
   3.2. Results 10  
   3.3. Decision 13  

4. Design and Implementation 14  
   4.1. Algorithmic Foundations 14  
      4.1.1. Algorithm A 14  
      4.1.2. Adaptive Arithmetic Coding 15  
   4.2. Implementation Design 16  
      4.2.1. Algorithm A 18  
      4.2.2. Adaptive Arithmetic Coding 18  

5. Testing and Results 19  
   5.1. Specification of testing methods 19  
      5.1.1. Difference Tests 19  
      5.1.2. Time Tests 19  
   5.2. Results 20  
      5.2.1. Compression Ratio 20  
      5.2.2. Time Tests 20  
      5.2.3. Space Complexity 20
Chapter 1.

Introduction

Begin - to begin is half the work,
    let half still remain;
    again begin this,
    and thou wilt have finished.
    - Marcus Aurelius

The use of advanced handheld devices increases rapidly. In relation to that fact, the users' expectations of what they can achieve with their devices is also increasing; the handheld devices is expected to work more or less as a replacement to their standalone computer. These facts put high demands on the mobile applications to take care of the devices' limited capacity; the algorithms used must be both effective and highly adapted.

1.1. Problem statement and Scope

In this bachelor thesis I will take a closer look on algorithms for compressing data. I will base my research on a scenario where communication between two devices will be done via the Short Messaging System [1]. This system sets a limit per message of 140 octets of data and the cost (in monetary terms) of sending one message is high; therefore it is important to keep the data to a bare minimum.

The thesis will try to answer the question "Which compression algorithm is best suited for compression of data for use in Short Messaging System messages?".

1.2. Structure of this thesis

I will begin by taking a look at some of the more common algorithms and types of algorithms in Chapter 2. Two of those will be chosen in Chapter 3 by the outcome of some tests and in Chapter 4 I will take a more in-depth look at how those two are functioning. I will also discuss how an effective implementation of them will look like and
implement them in the same chapter. Then I will go on and test these implementations together with some publicly available implementations in terms of both compression and computing efficiency in Chapter 5. In the 6:th and final chapter I will try to get an answer to which algorithm to use, when and why.
For some are sane and some are mad
And some are good and some are bad
And some are better, some are worse —
But all may be described in verse.
- T.S. Eliot

Data compression is a vast field comprised of a large set of algorithms and algorithmic variants. Each and everyone suggesting it has the most effective way of compressing your data. To give you a better understanding of the different aspects of data compression a short description of a few algorithms that can be defined as the backbone of this area will be given on the next pages.

2.1. Huffman Coding

Huffman Coding, also known as variable-size coding, is probably the best known compression algorithm there is. It is easy to understand and to implement which makes it a standard algorithm to be taught in many algorithm courses.

The complexity of the huffman algorithm does not lay in the compression or decompression parts but rather in how we define our Minimum-Redundancy codes. Basically the algorithm consists of a simple lookup table where each symbol in the alphabet is mapped against a minimum-redundancy code. A minimum-redundancy code is a code which can easily be distinguished from all other codes in a set of codes and has the lowest possible average message length [8].

Huffman [8] came up with a way to easily define those minimum-redundancy codes based on the probability of each symbol to appear. A more probable symbol would then get a shorter message code than a symbol appearing more sparse. Huffman’s algorithm is based on binary trees and can be seen described in figure 2.1. By walking left in the tree you add a 0 to your message code and by going right a 1. When reaching a leaf node you have found the message code corresponding to the symbol of that node.
Figure 2.1: Creation of a Huffman Tree. Consider each symbol as a binary tree root. For each step, add the two trees with the smallest probability to one binary tree until only one root exists.
Huffman Coding is a fast and easy to implement algorithm. On its downside it depends on the probability of each symbol to appear. It can be hard to know beforehand how often each symbol appears which might give a bad compression ratio. There exists two different ways to handle this problem, a two-pass algorithm and an adaptive huffman algorithm.

A two-pass algorithm does exactly what its name implies, it parses the uncompressed message two times. The first time it counts the number of occurrences of each symbol and creates a huffman tree which is used in the second parsing to compress the message. This algorithm will require the calculated huffman tree to be saved together with the compressed data which can give the data a quite hefty overhead, especially when small data sets is used.

The adaptive huffman algorithms on the other hand does this in only one pass. It starts out with an empty huffman tree and for each symbol it reads it updates the tree. This gives a better compression ratio for each symbol than a static huffman coding but requires more computing power to be effective. There exists variants which updates the tree every $N$:th symbol to make it more power efficient. Since the huffman tree is calculated during compression and decompression there will be no huffman tree to save together with the data. Good examples of this technique is the Algorithm Λ [6] and the FGK algorithm [3].

### 2.2. Arithmetic Coding

Arithmetic Coding [5] is a more complex algorithm than the Huffman algorithm. It is based on the probabilities of each symbol to appear and for each symbol read it calculates the probability of the whole message, which ends up as the compressed data.

It begins with an interval $[0, 1)$ which are subdivided into one interval for each symbol that may appear next, the size of each interval based on the probability of that symbol. After reading the next symbol the corresponding interval is picked as the new starting interval. Thereafter it starts all over again, subdividing this new interval and picking another one until the whole message has been read. The last interval symbolizes the compressed message and the shortest binary number that can distinguish this particular interval from every other are outputted.

Arithmetic Coding is hard to implement because it is based on high-precision float values. Therefore it is often implemented using fixed decimal values and by letting the interval expand when starting to get too small. A well-known version of this is given by Witten et. al [5].

Since arithmetic coding is based on probabilities it has the same flaws as Huffman coding, a need to save the calculated probabilities with the data, either as an overhead or coded
Chapter 2. Compression Algorithms

into the algorithm. To solve these issues there also exists adaptive arithmetic coding algorithms which calculates the probabilities on the fly [4].

2.3. Lempel-Ziv Coding

The Lempel-Ziv Coding is not one algorithm but actually a collection of algorithms, all based on the same technique. Basically it is a dictionary compression technique where strings are switched to their corresponding index in the dictionary. The big difference between the Lempel-Ziv methods is how the dictionary is created. A short description of three common Lempel-Ziv-algorithms will be given.

2.3.1. LZ77

The LZ77 algorithm uses a technique called sliding window to search for similarities in the message string. It works as a buffer which slides over the message. This buffer is divided into two parts, the search buffer and the look-ahead buffer. The search buffer consists of the last bytes that has been compressed, often up to some 1000 bytes [4]. It is this search buffer that works as the dictionary: this is where the search for a matching string is done. The look-ahead buffer on the other hand is the following bytes that will get compressed, often less than a 100 bytes long [4].

When compressing a symbol it begins searching for occurrences of that symbol from right to left in the search buffer. For each similar symbol it finds in the search buffer it starts to compare the following symbols (left to right) in the search buffer with the following in the look-ahead buffer until a difference is found (or until it has reached the end of the look-ahead buffer). It picks out the occurrence with the longest similarity with the look-ahead buffer and outputs its information as a token, \( (offset, length, following\ symbol) \) e.g. \( (4,2,’e’) \). If the symbol can’t be found in the search buffer the offset and length is set to 0 and the following symbol is the symbol which wasn’t found, eg. \( (0,0,’a’) \).

2.3.2. LZ78

The LZ78 algorithm uses a more traditional dictionary where past strings are added. It begins with a dictionary containing only one string, the null string or the empty string, and an empty buffer string. For every symbol read it searches the dictionary for any occurrences of the string buffer concatenated with the new symbol. If found, the symbol is added to the buffer and the next symbol is read. If not found, the string is added to the dictionary and a token consisting of two parts is created, the number of the dictionary string corresponding to the the string buffer and the symbol which was read, e.g. \( (3,’e’) \). This token is added to the compressed data and the string is stored in the dictionary.
together with the token. Before reading the next symbol the string buffer is emptied. For an example of how the dictionary can look like, see Table 2.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Character</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>(0,'a')</td>
</tr>
<tr>
<td>2</td>
<td>p</td>
<td>(1,'p')</td>
</tr>
<tr>
<td>3</td>
<td>e</td>
<td>(0,'e')</td>
</tr>
</tbody>
</table>

Table 2.1.: **LZ78 Dictionary** for the string *aaape*

### 2.3.3. LZW

The LZW algorithm is heavily based on LZ78. It uses a similar dictionary but starts out with all the single-byte symbols included. When a symbol sequence not found in the dictionary is encountered it stores it in the dictionary and outputs the dictionary entry recognized as the sequence up to the last read symbol. The string buffer is set to the last read symbol and the algorithm continues.
Chapter 3.

Testing and Choosing

Nothing is more difficult, and therefore more precious, than to be able to decide.
- Napoleon Bonaparte

In this thesis I will take a closer look on two of the algorithms mentioned in the previous chapter. To make sure those two are interesting in the focus area of this thesis I began everything by testing existing implementations of a wide range of algorithms. This chapter describes these tests and tries to give a good look into which two I have chosen and why.

3.1. Test Design

These tests have focused on the compression ratio for some of the most common compression algorithms. They have been designed by me using both files of random data and some widely used corpuses. Although this thesis is primarily focused on small files (around 100-300 bytes) of random data I will by testing more files get a better understanding of what each algorithm is good at.

All these files have been compressed and measured using a simple bash shell script which compresses each file, measures its length and computes a compression ratio for it. The full bash script can be found in Section A.3

3.1.1. Random files

The random files used for testing has been created using a C program using the internal random function of the c-library. This gives us a pseudo-random distribution of each character in the file’s specified alphabet.
These files have been separated into two distinguished tracks; one that looks in the interval of 100-300 bytes (one to two SMS-messages) and one which spans over a larger interval (32-262144 bytes). The small interval consists of 21 different sizes, each separated by 10 bytes which will give us a good look on how the algorithms perform in this focus interval. My larger interval consists of 14 files, each separated by the factor of 2 (e.g. $s_n = 2 \cdot s_{n-1}$ where $s_1 = 32$).

I have also chosen to use seven different alphabets to symbolize three different scenarios, only numbers and some separator, full english alphabet, and the full ASCII table. To make sure the usage of an alphabet which isn’t a power of 2, or divisible by 2, will affect the outcome of my tests I have decided to add one alphabet below and one above my calculated scenario alphabets for the small and medium interval.

**Small** 8, 11, and 16 symbols;

**Medium** 64, 69, and 72 symbols;

**Large** 256 symbols.

Finally, to minimize errors each and every combination of file and alphabet size have been outputted in ten different files. The average compression ratio of these collections of files has been calculated.

### 3.1.2. Industry-Standard Corpus

Alongside my own corpus of random files I have decided to run all tests on some publicly available corpuses from the internet [2]. In these corpuses you can find a wide range of files, from ones that may show an algorithm’s worst-case behaviour to files that resembles the data files found on an average computer. Therefore are these corpuses used all over the world to measure compression algorithms and give developers an easy way to compare their creation with all-ready published algorithms. A better description of each corpus can be found in Appendix C.

### 3.1.3. Algorithm Implementations

All of the implementations used in these tests have been found on the internet as open source. The full list of implementations can be found in Appendix B.
3.2. Results

The results will be presented for the alphabets of 11, 64 and 256 characters in the smallest interval because those are the most interesting for the scope of this thesis. Full test results can be found in Appendix D.

In Figure 3.1 we can see the compression ratio for files of an alphabet of 11 symbols in the range of 100-300 bytes. Figure 3.2 shows us the compression ratio for an alphabet of 64 symbols. Finally, Figure 3.3 shows us the results for an alphabet of 256 symbols. The compression ratio is measured in symbols per byte.

![Figure 3.1: Compression ratio of random files with alphabet of 11 symbols, 100-300 bytes.](image)
Figure 3.2.: Compression ratio of random files with alphabet of 64 symbols, 100-300 bytes.
Figure 3.3.: **Compression ratio** of random files with alphabet of 256 symbols, 100-300 bytes.
3.3. Decision

As we can see in the figures there is a big difference between how well the algorithms work on different alphabet sizes. Using the full ASCII table as an alphabet (Figure 3.3) gives us no compression at all but instead an increase in file size which in the end tells us that sending highly random data of large alphabets is a bad idea if we want to compress it.

When it comes to an alphabet size of 64 (Figure 3.2) the compression is still rather bad but there is one algorithm that stands out. The adaptive arithmetic coding gives us some kind of compression over the whole range of file sizes. It’s not a very large compression, at 140 bytes the compression lies at 1.05 symbols per byte, which gives us the ability to put another 7 symbols into our SMS message \(140 \times 1.05 = 147\).

Looking at an alphabet of 11 symbols (Figure 3.1) we see a significantly better result where only one algorithm doesn’t give us any kind of compression at all, the LZ77 algorithm. While taking a closer look at compressing files of size 280 bytes (two SMS messages) there is one algorithm that can compress that amount of data down to only one message and that is Algorithm \(\Lambda\) (stated Lambda in the figures). The other Huffman derivates and the Arithmetic codings all have the ability to compress about 200-250 bytes down to one message.

Based on these tests I have chosen to take a closer look on Algorithm \(\Lambda\) and the Adaptive Arithmetic coding. Why I am choosing Algorithm \(\Lambda\) should be quite clear, it is the only algorithm that is able to compress two messages down to one single message. Choosing the arithmetic coding on the other hand may be harder to understand. This choice is based solely on that I prefer to have two algorithms from different families and with more distinguished capabilities. The arithmetic coding family is the one with the most efficient compression of the ones remaining. Picking the adaptive arithmetic coding instead of the sometimes better static is done because I wanted the two algorithms to have the same ability to be used on data with unknown statistics.
Chapter 4.

Design and Implementation

What works good are better than what looks good, because what works good lasts.
- Ray Eames

Both Algorithm $\Lambda$ and the Adaptive Arithmetic Coding is two very interesting algorithms. How to implement them can be a little hard to understand at a first glance. This chapter will try to help you in understanding the algorithms and my implementations better.

4.1. Algorithmic Foundations

The two algorithms are very different. In this section I will give you a quick description of how they work together with pseudo-code for them.

4.1.1. Algorithm $\Lambda$

Algorithm $\Lambda$ [6] is an Adaptive Huffman coding. What separates it from a traditional Huffman coding is that it starts out with a nearly empty Huffman Tree and by each symbol compressed it updates the tree.

The tree nodes can be of three different types, a branch, a leaf, or a NYT. NYT stands for Not Yet Transmitted and is used for symbols which haven’t been added to the tree yet. When such a symbol is encountered the Huffman Code for the NYT node is outputted followed by the uncompressed symbol. The symbol is then added to the tree for later use.

For all this to work the tree starts out with only one node, a NYT node. When adding a new symbol to the tree the NYT node is converted to a branch node with its left child as a new NYT node and the right child as a leaf corresponding to the new symbol. If
the symbol already exists in the tree it slides past all nodes of equal weight, but not its parent, and the weight is then updated.

The full update procedure is described in Algorithm 1 and 2. In the algorithms the tree is assumed to be stored in a consecutive list of nodes of decreasing weight. Branch nodes are always stored before leaf nodes. A more in-depth description of Algorithm Λ is given by J.S. Vitter [6].

Algorithm 1 Λ Update Procedure
1: \( b \leftarrow \) symbol to update
2: \( Tree \leftarrow \) Huffman Tree to update
3: if \( b \) is not part of \( Tree \) then
4: \( Tree.NYT \leftarrow \) branch node
5: \( Tree.NYT.RChild \leftarrow \) leaf \( b \) with weight 1
6: \( Tree.NYT.LChild \leftarrow \) new NYT
7: \( node \leftarrow \) node of old NYT
8: else
9: \( node \leftarrow \) node of \( b \)
10: end if
11: while \( node \) is part of \( Tree \) do
12: \( node \leftarrow \) slideAndIncrement(\( node \), \( Tree \))
13: end while

Algorithm 2 Λ SlideAndIncrement Procedure
1: \( node \leftarrow \) node to update
2: \( Tree \leftarrow \) Huffman Tree
3: \( wt \leftarrow node.weight \)
4: if \( node \) is a branch then
5: \( wt \leftarrow wt + 1 \)
6: end if
7: slide \( node \) past all nodes of weight \( wt \), stop at its parent
8: \( node.weight \leftarrow node.weight + 1 \)
9: if \( node \) is a leaf then
10: return new parent of \( node \)
11: else
12: return old parent of \( node \)
13: end if

4.1.2. Adaptive Arithmetic Coding

Arithmetic Coding is somewhat harder to implement. This is because the theory behind it is based on infinite precision arithmetic. By dividing an interval of \([0, 1)\) we will sooner
or later come to the conclusion that we no longer can distinguish the lower bound from
the higher bound.

There are a lot of different ways to take care of this problem and make Arithmetic
Coding work. I have decided to follow the algorithm described by Witten et. al. [5]
because it is a well-known and established algorithm. This particular algorithm uses
fixed point calculations instead of floating point calculations. By outputting every bit
that is known at the moment and then expanding the interval it takes care of the infinite
precision problem. The algorithm for encoding is described in Algorithm 3 and 4. Full
description is given by Witten et. al. [5].

### Algorithm 3: Arithmetic Update Procedure

1. $symbol \leftarrow$ symbol to update
2. $fullFrequency \leftarrow$ the full cumulative frequency
3. if $fullFrequency = maxFrequency$ then
4.   halve the cumulative frequency of all symbols and change $fullFrequency$ accord-
   ingly
5. end if
6. move $symbol$ past all symbols of equal or lesser count
7. $symbol.count \leftarrow symbol.count + 1$
8. $fullFrequency \leftarrow fullFrequency + 1$
9. update cumulative frequency on symbols of higher count

### 4.2. Implementation Design

These two algorithms have been implemented in the Java language. My choice of the
Java language is based solely on the fact that it is the most common on mobile phones
today. Both cheaper phones and more advanced ones using the Android platform have
the capability of running Java applications. Since Java is an object-oriented language I
had to give the algorithms a somewhat object-oriented approach. In this section I will
describe the choices I’ve taken to achieve this and how my implementations look like.

My hopes by implementing these algorithms are to see whether I can gain anything in
comparison to the open source variants and to be able to see how they perform in a
mobile Java environment.

For bit-I/O both classes use my class `BitHandler.java`. This class can both handle byte-
array to bit and bit to byte-array operations. The source code of `BitHandler.java` can
be found in Section A.2.
Algorithm 4 Arithmetic Encoding Procedure

1: whole ← 65535
2: quarter ← whole/4 + 1
3: half ← quarter · 2
4: threequarter ← quarter · 3
5: b ← symbol to encode
6: low ← low value of interval
7: high ← high value of interval
8: fullFrequency ← the full cumulative frequency
9: range ← high − low + 1
10: high ← low + (range * (b.cumulativeFreq + b.count))/fullFrequency − 1
11: low ← low + (range * b.cumulativeFreq)/fullFrequency
12: followCount ← 0
13: while true do
14: if high < half then \{Interval resides in lower half\}
15: output bit as 0
16: while followCount ≠ 0 do
17: output bit as 1
18: followCount ← followCount − 1
19: end while
20: else if low ≥ half then \{Interval resides in higher half\}
21: output bit as 1
22: while followCount ≠ 0 do
23: output bit as 0
24: followCount ← followCount − 1
25: end while
26: low ← low − half
27: high ← high − half
28: else if low ≥ quarter and high < threequarter then \{Interval resides in the middle half\}
29: followCount ← followCount + 1
30: low ← low − quarter
31: high ← high − quarter
32: else
33: break
34: end if
35: update(c)
36: end while
4.2.1. Algorithm Λ

My implementation has two classes, the outer Lambda class which symbolises the whole Algorithm Λ and its private class Node. The Node class carries all information needed to distinguish a particular node, its type (branch, leaf, or NYT), weight, symbol, parent, and child. Since the whole tree is stored in an array the parent and child variables contains only the numerical representation of where it is stored. Although only one child variable exists the branch nodes always have two children. The child variable points to its left child and to find its right child, 1 is added to its number.

The tree is stored in an array of nodes sorted in decreasing order based on its weight. Branch nodes is stored before leaf nodes of the same weight. This gives us that at position 0 of the array the root node is always stored and the NYT node is always at the very end. When splitting the NYT node there is no need to move any nodes (as long as there are room left at the end), just appending the two new nodes. To find the position of a leaf node in the array I have implemented a list that translates the symbol to its position without the need of searching through the array.

Full source-code of Algorithm Λ can be found in Section A.5.

4.2.2. Adaptive Arithmetic Coding

My Arithmetic Coding implementation is also divided into two classes, the Arithmetic class and its private class symbol. The symbol class symbolizes a symbol and its probability. All those symbols are stored in an array of 257 symbols (the full ASCII table and one End-Of-File symbol), sorted in decreasing probability. Since we don’t know any of the probabilities beforehand they are all given the equal probability at the beginning of the algorithm.

The probabilities is stored as number of occurrences and the cumulative number of occurrences of every symbol to the right in the array. To quickly find the position of a certain symbol I have implemented a list that translates a symbol to its position without the need of searching through the array.

Full source-code of my Adaptive Arithmetic Coding implementation can be found in Section A.1.
Chapter 5.

Testing and Results

Program testing can be used to show the presence of bugs, but never to show their absence!
- Edsger Dijkstra

To be able to compare the two chosen algorithms I have run tests to collect data of them. In this chapter these tests are both described and their results presented.

5.1. Specification of testing methods

Testing of my own implementations have been done in much the same way as the tests described in Section 3.1 combined with some new tests to measure the execution time and correctness of the algorithms. In this section I will describe these new tests.

5.1.1. Difference Tests

The difference tests have been designed to show any compression errors. This is done by checking whether the decompressed output differs from the original input. To achieve this I have used the `diff` command in unix systems and ran it on the random files described earlier. Full code listing of these tests can be found in Section A.4.

5.1.2. Time Tests

Time tests are used to measure the amount of time for the algorithm to execute. I have chosen to use Java’s built-in time methods to compare the system time before and after running the compress method. This gives me an output in nanoseconds which is taken care of in a bash script.

By putting the measurement inside the java class file I get rid of any overhead from file handling or the java runtime environment. It is still no exact measurement since the
computer systems I’m using are multitasking and I don’t have any way to give a process full priority in the system, i.e. I can’t be certain that my measured process aren’t put on hold waiting for another process to finish. To minimize the risk of this happening I have shut down as many processes as possible before running these tests. Therefore, while not being 100% accurate they can at least give me a hint of how the algorithms perform compared to each other.

Full code listings can be found in Section A.7.

5.2. Results

Results from these tests are shown in the following subsections. In the figures presented my implementations of algorithms are followed by $M$. Only the most relevant results will be shown, a selection of the rest can be found in Appendix E. The difference tests were all successful.

5.2.1. Compression Ratio

The compression ratio for my implementations compared to the earlier tested implementations are shown for alphabet 11, 64, and 256 in Figure 5.1, 5.2, and 5.3. My implementation of the adaptive arithmetic coding has the same ratio as the earlier tested version and are therefore hard to see in the graphs, as are shown in the tables in Appendix E.

5.2.2. Time Tests

The time measured in nanoseconds for my implementations are shown for alphabet 11, 64, and 256 in Figure 5.4, 5.5, and 5.6.

5.2.3. Space Complexity

Memory usage of my implementations are as follows. Algorithm Λ’s memory usage can be calculated by \( m = 1032 + c \cdot 60\text{byte} \) where \( c \) is the number of symbols in the tree. The arithmetic coding on the other hand have a constant memory usage of 3604\text{byte}. In these calculations the memory usage of the BitHandler class has been ignored because it is the same for both algorithms.
Figure 5.1.: Compression ratio of random files with alphabet of 11 symbols, 100-300 bytes, including my algorithm implementations.
Figure 5.2.: Compression ratio of random files with alphabet of 64 symbols, 100-300 bytes, including my algorithm implementations.
Chapter 5. Testing and Results

Figure 5.3.: Compression ratio of random files with alphabet of 256 symbols, 100-300 bytes, including my algorithm implementations.
Figure 5.4.: **Time (ns)** for compression of random files with alphabet of 11 symbols, 100-300 bytes.
Figure 5.5.: **Time (ns)** for compression of random files with alphabet of 64 symbols, 100-300 bytes.
Figure 5.6.: Time (ns) for compression of random files with alphabet of 256 symbols, 100-300 bytes.
Chapter 5. Testing and Results

5.2.4. Big-O notation

The Big-O notations are calculated with the source code of my implementations as a base (Appendix A) using \( b \) as the file size in bytes and \( s \) as the number of symbols in the tree or model. In Table 5.1 my calculations for the methods of Algorithm \( \Lambda \) are presented and in Table 5.2 the same is done for the Adaptive Arithmetic coding. A comparison of the \textit{compress} and \textit{decompress} methods of those implementations can be found in Table 5.3.

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitsToSymbol</td>
<td>( O(\log_2(s)) )</td>
</tr>
<tr>
<td>nodeNumToBits</td>
<td>( O(\log_2(s)) )</td>
</tr>
<tr>
<td>slideAndIncrement</td>
<td>( O(s) )</td>
</tr>
<tr>
<td>update</td>
<td>( O(s \cdot \log_2(s)) )</td>
</tr>
<tr>
<td>decode</td>
<td>( O(s \cdot \log_2(s)) )</td>
</tr>
<tr>
<td>encode</td>
<td>( O(s \cdot \log_2(s)) )</td>
</tr>
<tr>
<td>decompress</td>
<td>( O(s \cdot b \cdot \log_2(s)) )</td>
</tr>
<tr>
<td>compress</td>
<td>( O(s \cdot b \cdot \log_2(s)) )</td>
</tr>
</tbody>
</table>

Table 5.1.: Big-O calculations for my implementation of Algorithm \( \Lambda \)

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>( O(s) )</td>
</tr>
<tr>
<td>printFollowCount</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>decode</td>
<td>( O(s) )</td>
</tr>
<tr>
<td>encode</td>
<td>( O(s) )</td>
</tr>
<tr>
<td>decompress</td>
<td>( O(s \cdot b) )</td>
</tr>
<tr>
<td>compress</td>
<td>( O(s \cdot b) )</td>
</tr>
</tbody>
</table>

Table 5.2.: Big-O calculations for my implementation of Adaptive Arithmetic encoding

<table>
<thead>
<tr>
<th></th>
<th>Algorithm ( \Lambda )</th>
<th>Ad-Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>( O(s \cdot b \cdot \log_2(s)) )</td>
<td>( O(s \cdot b) )</td>
</tr>
<tr>
<td>Decompress</td>
<td>( O(s \cdot b \cdot \log_2(s)) )</td>
<td>( O(s \cdot b) )</td>
</tr>
</tbody>
</table>

Table 5.3.: Comparison of Big-O notation for my implementations.
Chapter 6.

Discussion and Conclusion

To study and not think is a waste.
To think and not study is dangerous.
- Confucius

Out of all the results already presented a conclusion has to be made. In the following sections I will discuss these results and what they mean and in the end draw a conclusion that answers my question of which algorithm is best suited for compression of data to be used in an Short Messaging System message.

6.1. Discussion

I have taken an overview of the most common types of compression algorithms, all having their own benefits and disadvantages. As I have already stated, Huffman and Arithmetic coding are all better than any dictionary method in the scope of this thesis. But one should not rule out the dictionary methods purely on these results because when studying the results from the corpus compression (Section D.3) one can see a significantly better compression ratio for these methods than any of the other alternatives. This is mainly because the files found in these corpuses are all in some way far more repetitive than my random tests. Dictionary methods are built around the fact that sooner or later all patterns comes back. In my random data files it is very hard to find any pattern because they are both random and very small, i.e. the number of combinations of symbols are greater than can fit inside the file.

Huffman and Arithmetic coding both compress symbol by symbol and therefore don’t rely on any repetitive pattern between symbols but instead the number of occurences of each symbols. When using a small alphabet each symbol will get a lot of occurences even in small files, this is what makes the compression work although randomness is used. Expanding the alphabet gives less occurences per symbol which in the algorithms gives a larger huffman tree and smaller probabilities. This makes the code for each symbol longer and when the alphabet gets large enough there will not be any compression at all, and in some cases even an increase in file size.
It’s interesting to compare the compression ratio of my implementations with the publicly available ones. When it comes to the Adaptive Arithmetic coding there exists absolutely no difference between the two implementations. My implementation of Algorithm Λ on the other hand gives a better compression ratio than the other implementation in some cases. Although the gain is not large this gives me a good lesson that implementation of an algorithm can be hard and although it may seem right there still may be some things that differ. My implementation of Algorithm Λ follows the paper by J.S. Vitter [6] and the test cases given by him all gives the correct result. This strengthens my belief that my implementation are correct.

When it comes to time testing one might wonder if the results presented in Section 5.2.2 are correct since there are some unknown random peaks. As I’ve explained earlier these peaks probably comes from the fact that I have no way to give my compression process full access to the processor when running. Since the curves overall seems to be linear and the peaks are sparse I can still draw the conclusion that my implementation of Arithmetic coding is far better than my implementation of Algorithm Λ when it comes to execution time. At first this confuses me as I’ve got the impression beforehand that arithmetic coding gives a good compression ratio but it costs in terms of efficiency. But it is not very hard to see that it is correct, all one needs to do is look at the Big-O notations which fully supports this difference.

6.2. Conclusion

In conclusion, which algorithm suits the scope of this thesis best can be hard to say. Both algorithms are good at different things. Algorithm Λ is clearly the winner when it comes to compression ratio while the Adaptive Arithmetic coding executes more efficiently. In my opinion compression ratio is more important because this scope have a very limited message size, therefore I would recommend Algorithm Λ. But it should be noted that a new decision has to be made for each application since the benefits of each algorithm differ.

6.3. Future work

While carrying out this thesis some ideas have been encountered that could be interesting to give a closer study in another report:

- The adaptive method of the arithmetic coding in my implementation is relatively basic. Can we gain anything in terms of compression ratio by using another method?

- If we know the alphabet used in advance can we then design the statistical model or the Huffman tree for better compression but still retain that it is adaptive?
Chapter 6. Discussion and Conclusion

- What are the advantages and disadvantages of using larger symbols (e.g. two or more characters per symbol)? Will we gain anything by using that?
Bibliography


Appendix A.

Code Listings

This thesis has resulted in a lot of source code. This Appendix lists everything created.

A.1. Arithmetic.java

My Java implementation of Arithmetic Coding.

```java
/*
 * Arithmetic.java – Adaptive Arithmetic Coding
 * 20100420 – Måns Andersson <mdan06@student.bth.se>
 */
/*
 * Interval is built around
 * 0] = Low point
 * [1 = High point
 * * Symbols are 0–255 = ASCII, 256 = EOF-symbol
 * symbols table is sorted by the descending frequency
 * * Use the symToIndex-table to quickly find the symNum of your symbol
 */
import java.lang.String;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;
public class Arithmetic {

    // Class to symbolize symbols in array
    private class symbol {
        public short sym;
        public int count;
        public int cumFreq;
    }

    private static final int numBits = 16; // 16
    private static final int whole = 65535; // 2^16 − 1
    private static final int quarter = whole / 4 + 1;
    private static final int half = quarter * 2;
    private static final int threequarter = quarter * 3;
    private static final int max_frequency = 16383; // 2^14−1

    private BitHandler bithndl;
    private int[] interval; // Interval 0 = low, 1 = high
}
```
private int followCount; // Counter for follow bits
private int value; // Value read by decompress function
private symbol[] symbols; // Array of symbols
private short[] symToIndex; // symbol->index jump list
private int fullFrequency; // the full cumulative frequency

class Arithmetic() {
    private BitHandler bithndl = new BitHandler(null);
    private int followCount = 0;
    private int[] interval = new int[2];
    private int interval[0] = 0;
    private int interval[1] = whole;

    // Initialize model
    symToIndex = new short[257];
    symbols = new symbol[257];
    for (int i=0;i<257;i++) {
        symbols[i] = new symbol();
        symbols[i].sym = (short)i;
        symbols[i].count = 1;
        symbols[i].cumFreq = 256-i;
        symToIndex[i] = (short)i;
    }
    fullFrequency = 257;
}

// Compress whole byte-array
public byte[] compress(byte[] b) {
    interval[0] = 0;
    interval[1] = whole;
    followCount = 0;
    // Encode file
    for (int i=0;i<b.length;i++) {
        encode((short)(b[i] & 0xFF));
    }
    // Encode EOF-character
    encode((short)256);
    // Finish and return
    followCount++;
    if (interval[0]<quarter) {
        bithndl.bit_addBit(0);
        printFollowCount(1);
    } else {
        bithndl.bit_addBit(1);
        printFollowCount(0);
    }
    bithndl.bit_finish();
    return bithndl.bit_getBytes();
}

// Decompress whole byte-array
public byte[] decompress(byte[] b) {
    interval[0] = 0;
    interval[1] = whole;
    followCount = 0;
    // Read first numBits bits
    int nextBit;
    bithndl.byte_setBytes(b);
    for (int i=0;i<numBits;i++) {
        if ((nextBit = bithndl.byte_getNextBit()) == 2) {
            value = 2*value;
        } else {
            value = 2*value + nextBit;
        }
    }
    // Loop through each character until EOF is encountered
while (true) {
    short symbol = decode();
    if (symbols[symbol].sym == 256) break; // EOF encountered
    bitndl.bit_addByte((char)symbol);
    update((short)symbol);
}  
return bitndl.bit_getBytes();
  
// Return number of bytes in returned array
public int getNumOfBytes() {
    return bitndl.bit_size();
}

// Encode one character
private void encode(short c) {
    short symNum = symToIndex[c];
    int range = interval[1] − interval[0] + 1;
    interval[1] = interval[0] + (range * (symbols[symNum].cumFreq + symbols[symNum−
. count]))/fullFrequency −1;
    interval[0] = interval[0] + (range * symbols[symNum].cumFreq)/fullFrequency;
    while (true) {
      if (interval[1] < half) { // [0:0.5)
        bithndl.bit_addBit(0);
        printFollowCount(1);
      } else if (interval[0] >= half) { // [0.5:1)
        bithndl.bit_addBit(1);
        printFollowCount(0);
        interval[0] = half;
        interval[1] = half;
      } else if (interval[0] >= quarter && interval[1] < threequarter) { // ←
        [0.25:0.75)
        followCount++;
        interval[0] = quarter;
        interval[1] = quarter;
      } else break;

        interval[0] = 2*interval[0];
    }
    update(c);
}

// Decode one character
private short decode() {
    int nextBit;
    int symbol;
    int range = interval[1]−interval[0]+1;
    int cumFreq = ((value−interval[0]+1) * fullFrequency − 1)/range;

    // Search symbol
    for (symbol = 0; symbols[symbol].cumFreq>cumFreq; symbol++) if (symbol == 256) ←
    break;

    // Count new intervals
    interval[1] = interval[0] + (range*symbols[symbol].cumFreq+symbols[symbol].←
    count))/fullFrequency−1;
    interval[0] = interval[0] + (range*symbols[symbol].cumFreq)/fullFrequency;
    while (true) {
      if (interval[1] < half) { // do nothing
        } else if (interval[0] >= half) {
          value = half;
          interval[0] = half;
          interval[1] = half;
        } else if (interval[0] >= quarter && interval[1] < threequarter) {
          value = quarter;
          interval[0] = quarter;
        }
interval[1] = quarter;
} else break;
interval[0] = 2*interval[0];
nextBit = bithndl.byte_getNextBit();
if (nextBit == 2) value = 2*value;
else value = 2*value + nextBit;
}
return symbols[symbol].sym;

// Print follow bits
private void printFollowCount(int bit) {
while (followCount != 0) {
  bithndl.bit_addBit(bit);
  followCount--;
}
}

// Update statistical model
private void update(short c) {
  // If max frequency, halve all counts
  if (fullFrequency == max_frequency) {
    int cumFreq = 0;
    for (int i=symbols.length-1;i>=0;i--) {
      symbols[i].count = (symbols[i].count+1)/2;
      symbols[i].cumFreq = cumFreq;
      cumFreq += symbols[i].count;
    }
    fullFrequency = cumFreq;
  }
  // Find new position of symbol
  short symIndex = symToIndex[c];
  short newIndex = symIndex;
  // If not symIndex 0, try moving it
  if (symIndex > 0) {
    while (symbols[newIndex].count >= symbols[newIndex-1].count) {
      newIndex--;
      if (newIndex == 0) break;
    }
    // Switch symbols
    if (symIndex != newIndex) {
      symbol symTmp = symbols[symIndex];
      symbol newTmp = symbols[newIndex];
      symbols[symIndex] = newTmp;
      symbols[newIndex] = symTmp;
      int intTmp = symTmp.cumFreq;
      symTmp.cumFreq = newTmp.cumFreq;
      newTmp.cumFreq = intTmp;
      symToIndex[symTmp.sym] = newIndex;
      symToIndex[newTmp.sym] = symIndex;
    }
  }
  symbols[newIndex].count++;
  // Add cumFreq
  while (newIndex != 0) {
    newIndex--;
    symbols[newIndex].cumFreq++;
  }
  fullFrequency++;
}

// Just some file handling, beginning, ending.
public static void main(String[] args) {
try {
    Arithmetic a = new Arithmetic();
    File ofile = new File(args[2]);
    File ifile = new File(args[1]);
    FileInputStream fis = new FileInputStream(ifile);
    FileOutputStream fos = new FileOutputStream(ofile);
    byte[] b = new byte[fis.available()];
    fis.read(b);
    if (args[0].equalsIgnoreCase("d")) { // Decompress
        b = a.decompress(b);
    } else { // Compress
        long startTime = System.nanoTime();
        b = a.compress(b);
        if (args[0].equalsIgnoreCase("t")) {
            System.out.println(""+System.nanoTime()-startTime));
        }
    }
    fos.write(b, 0, a.getNumOfBytes());
    fis.close();
} catch (java.lang.ArrayIndexOutOfBoundsException ex) {
    error("not enough args");
} catch (java.io.FileNotFoundException ex) {
    error("file not found");
} catch (java.io.IOException ex) {
    error("i/o error");
}

public static void error(String m) {
    System.out.println("Usage: java Arithmetic [c/d] <inputfile> <outputfile>");
    System.out.println("Error: "+m);
}

A.2. BitHandler.java

My Java implementation to handle bit-I/O.

/*
 * BitHandler.java – A class for handling bytes bit by bit
 * 20100414 – Måns Andersson <mdan06@student.bth.se>
 */

/* This class is divided into two parts which are completely separated
 * 1) Input byte-array, read bit by bit (methods beginning with byte)
 * 2) Input bits, return byte-array (methods beginning with bit)
 */
public class BitHandler {
    private byte[] bytes;
    private int byteC; // bytecount
    private short bitC; // bitcount
    private byte[] inputBytes;
    private int inputByteC; // bytecount
    private char inputTmp;
    private char inputTmpC;

    public BitHandler(byte[] b) {
        bytes = b;
        inputBytes = new byte[140];
    }
// This handles the byte to bit part (byte-array, bytes)
public int byte_getNextBit() {
    int retval = 0;
    if (byteC < bytes.length) {
        byte b = bytes[byteC];
        if (((b << bitC) >> 7) & 1) != 0) retval = 1;
        bitC++;
        if (bitC > 7) {
            bitC = 0;
            byteC++;
        } else retval = 2;
        return retval;
    }
    else return retval;
}

// Set byte-array
public void byte_setBytes(byte[] b) {
    bytes = b;
}

// This handles the bit to byte part (bit-array, inputBytes)

// Add one bit to array (add to front)
public void bit_addBit(int b) {
    inputTmp = (char)(inputTmp << 1);
    if (b == 1) {
        // Add a 1 (0 is added automatically)
        inputTmp += 1;
    }
    inputTmpC++;
    if (inputTmpC > 7) {
        // Add byte
        if (inputByteC == inputBytes.length) {
            byte[] b2 = new byte[inputByteC+140];
            System.arraycopy(inputBytes, 0, b2, 0, inputByteC);
            inputBytes = b2;
        }
        inputBytes[inputByteC] = (byte)inputTmp;
        inputByteC++;
        inputTmpC = 0;
        inputTmp = 0;
    }
}

// Add a whole byte to bit-array
public void bit_addByte(char b) {
    int counter = 7;
    for (;counter > -1;counter--) {
        if ((b >> counter & 1) != 0) bit_addBit(1);
        else bit_addBit(0);
    }
}

// Finish array by filling last byte with 0
public void bit_finish() {
    while(inputTmpC != 0) bit_addBit(0);
}

// Return bit-array
public byte[] bit_getBytes() {
    return inputBytes;
}

public int bit_size() {
    return inputByteC;
}
}
A.3. compresstest.sh

File for running tests on compression ratio.

```bash
#!/bin/bash

# MINMAX NAMES (NUM CHARs IN ALPHABET)
MM=(8 11 16 64 69 72 256)

# NUMBER OF BYTES PER FILE
NB=(100 110 120 140 150 160 170 200 220 230 240 250 260 270 280 290 300)
NBLOG=(32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144)

# CORPUSES AVAILABLE
CORPUSES='artificial calgary cantrbry large'

# Functions for different algorithms
# Args: 1 - Files
# Or 1 - Alphabet size
# 2 - Files
function compress-lz77 {    for FILE in $(ls $1); do
    A=./lz77c $FILE $FILE.LZ77
    done
    COMPRESSED=$(wc -c $1.LZ77 | awk '{print $1}')
    echo $COMPRESSED
    rm $1.LZ77
}

function compress-lzw {
    compress -f $1
    COMPRESSED=$(wc -c $1.Z | awk '{print $1}')
    echo $COMPRESSED
    uncompress -f $1.Z
}

function compress-vitter {
    for FILE in $(ls $2); do
        ./vitter c$1 $FILE $FILE.V
    done
    COMPRESSED=$(wc -c $2.V | awk '{print $1}')
    echo $COMPRESSED
    rm $2.V
}

function compress-huffman {
    for FILE in $(ls $1); do
        ./huffcode -i$FILE -o$FILE.H -c
    done
    COMPRESSED=$(wc -c $1.H | awk '{print $1}')
    echo $COMPRESSED
    rm $1.H
}

function compress-fgk {
    for FILE in $(ls $1); do
        A=./fgkc $FILE $FILE.FGK
    done
    COMPRESSED=$(wc -c $1.FGK | awk '{print $1}')
    echo $COMPRESSED
    rm $1.FGK
}

function compress-arithmetic {
    for FILE in $(ls $1); do
        A=./arithmetic -c -i $FILE -o $FILE.A
    done
    COMPRESSED=$(wc -c $1.A | awk '{print $1}')
    echo $COMPRESSED

```
rm $1''.A

}  

function compress-ad-arithmetic {
    for FILE in $( ls $1 ); do
        A="/arithmetic$c -i $FILE.AA $a -A
        COMPRESSED=`wc -c $1''.AA | tail -n 1 | awk '{print $1}''
        echo $COMPRESSED
        rm $1''.AA
    done
}

function compress-lambda-m {
    for FILE in $( ls $1 ); do
        java -classpath ./MyAlgs/ Lambda $FILE $FILE.L
        COMPRESSED=`wc -c $1''.L | tail -n 1 | awk '{print $1}''
        echo $COMPRESSED
        rm $1''.L
    done
}

function compress-arithmetic-m {
    for FILE in $( ls $1 ); do
        java -classpath ./MyAlgs/ Arithmetic $FILE $FILE.A
        COMPRESSED=`wc -c $1''.A | tail -n 1 | awk '{print $1}''
        echo $COMPRESSED
        rm $1''.A
    done
}

# Run tests on algorithms
# Args: 1 - Alphabet Size  
# 2 - Name of line in file  
# 3 - File name
function runtests {
    UNCOMPRESSED=`wc -c $3 | tail -n 1 | awk '{print $1}''

    # RUN TESTS
    COMPRESSED="compress-lz77 "$3"
    LZ77="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-lzw "$3"
    LZW="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-vitter $1 "$3"
    VITTER="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-huffman "$3"
    HUFFMAN="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-fgk "$3"
    FGK="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-arithmetic "$3"
    ARITHMETIC="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-ad-arithmetic "$3"
    ADARITHMETIC="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-lambda-m "$3"
    LAMBDA="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"
    COMPRESSED="compress-arithmetic-m "$3"
    ARITHMETIC="echo "$UNCOMPRESSED" "$COMPRESSED" | bc -l"

    # PRINT
    echo "$2;HUFFMAN;$VITTER;$FGK;ARITHMETIC;ADARITHMETIC;LZW;LZ77;LAMBDA;" >> SARITHMETIC"
}

# RANDOM FILES
for MINMAX in $MM: do
    echo "-;Huffman;Lambda;FGK;Arithmetic;Ad-Arithmetic;LZW;LZ77;Lambda [M];Arithmetic <-> [M]" > result-$MINMAX'.csv
    for NUMBYTES in $NB: do
        FILESTRING="/Slumpfile\test -$MINMAX-$NUMBYTES-4"
        echo 'runtests "$MINMAX" "$NUMBYTES" "$FILESTRING"' >> result-$MINMAX'.csv
    done
done
Appendix A. Code Listings

```
A.4. difftest.sh

File for checking that the decompressed file is the same as the uncompressed.

#!/bin/bash

# MINMAX NAMES (NUM CHARs IN ALPHABET)
MM='8 11 16 64 69 72 256'
# NUMBER OF BYTES PER FILE
NB='100 110 120 130 140 150 160 170 190 200 210 220 230 240 250 260 270 280 290 300'
NBLOG='32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144'
# CORPUSES AVAILABLE
CORPUSES='artificial calgary cantrbry large'

# Functions for different algorithms
# Args: 1 - Files
function compress-lambda {
  for FILE in $1; do
    java -classpath ./MyAlgs/Lambda c $FILE $FILE.L
    java -classpath ./MyAlgs/Lambda d $FILE.L $FILE.L2
    diff $FILE $FILE.L2
  done
  rm $1".L
  rm $1".L2
}

function compress-arithmetic {
  for FILE in $1; do
    java -classpath ./MyAlgs/Arithmetic c $FILE $FILE.A
    java -classpath ./MyAlgs/Arithmetic d $FILE.A $FILE.A2
    diff $FILE $FILE.A2
  done
  rm $1".A
  rm $1".A2
}

# Run tests on algorithms
# Args: 1 - File Name
```
Appendix A. Code Listings

```java
function runtests {
    # RUN TESTS
    compress-lambda "$1"
    compress-arithmetic "$1"
}

# RANDOM FILES
for MINMAX in $MM; do
    for NUMBYTES in $NB; do
        FILESTRING="/slumpfile/"test-$MINMAX-$NUMBYTES.s"
        runtests "$FILESTRING"
    done
done
echo 'This should be the only output'
```

A.5. Lambda.java

My Java implementation of Algorithm Λ

```java
/*
* Lambda.java - Algorithm Lambda, an adaptive huffman algorithm by J.S. Vitter
* 20100414 - Måns Andersson <mdan06@student.bth.se>
*/

/*
* All nodes are stored in consecutive decreasing order based on their weight.
* Leafs are stored after internal nodes of the same weight.
* Root are always nodeNum 0
* NYT is always stored last in the ArrayList
* Therefore [...] [NYT] => [...] [oldNYT] [Symbol] [NYT] when adding
* Use the nodeList to quickly find the node of a symbol
*/
import java.lang.String;
import java.lang.Integer;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;
import java.util.Stack;

public class Lambda {
    private static final int t_none = 0, t_branch = 0, t_symbol = 1, t_nyt = 2;

    private class Node {
        // Child is always the right child. Left child is found by (right child + 1)
        public int weight;
        public int type;
        public int symbol = 256;
        public int parent = -1;
        public int child = -1;

        public String toString() {
            return "Type: " + type + " - W: " + weight + " - Char: " + (char)symbol + " - P:C: " + parent + ":" + child;
        }
    }

    private BitHandler bithndl;
    private Node[] tree; // All nodes
}
```

41
private int treeSize;
private int[] nodeList; // nodeList[c] = node of c
private int nyt; // Pointer to NYT node

public Lambda(int alphabetSize) {
    bithndl = new BitHandler(null);
    treeSize = alphabetSize*3;
    tree = new Node[treeSize];
    nodeList = new int[257];
    for (int i=0;i<257;i++) {
        nodeList[i] = -1;
    }
    tree[0] = new Node();
    tree[0].type = t_nyt;
    tree[0].child = -1;
    tree[0].parent = -1;
    tree[0].weight = 0;
    nyt = 0;
}

// Compress / Decompress functions for whole byte-arrays
public byte[] compress(byte[] b) {
    for (int i=0;i<b.length;i++) {
        encode((char)(b[i] & 0xFF));
    }
    nodeNumToBits(nyt);
    bithndl.bit_finish();
    return bithndl.bit_getBytes();
}

public byte[] decompress(byte[] b) {
    String bits = "";
    int lastBit = 2;
    int c;
    bithndl.byte_setBytes(b);
    lastBit = bithndl.byte_getNextBit();
    while (lastBit < 2) {
        if (lastBit == 1) bits = bits + "1";
        else bits = bits + "0";
        c = decode(bits);
        lastBit = bithndl.byte_getNextBit();
        if (c != -1) bits = "";
    }
    bithndl.bit_finish();
    return bithndl.bit_getBytes();
}

public int getNumOfBytes() {
    return bithndl.bit_size();
}

// Encode a character, adding and updating the tree
private void encode(char c) {
    // Check if it exists
    int nodeNum = nodeList[c];
    if (nodeNum == -1) { // Node doesn't exist, use NYT and add
        nodeNum = this.nyt;
        nodeNumToBits(nodeNum);
        bithndl.bit_addByte(c);
    } else { // Node exists, calculate hCode
        nodeNumToBits(nodeNum);
    }
    update(c);
}

// Decode a character, adding and updating the tree
private int decode(String s) {
```java
int c = bitsToSymbol(s);
if (c != -1) {
    update((char)c);
    bithndl.bit_addByte((char)c);
}
return c;
}

// Add symbol to tree and update it
private void update(char c) {
    Node tmpNode;
    // Check if it exists
    int nodeNum = nodeList[c];
    if (nodeNum < 0) { // Node doesn't exist, use NYT and add
        if (this.nyts > = tree.length-3) { // tree is full, allocate more space
            Node[] newTree = new Node[(int)(tree.length*1.5)];
            System.arraycopy(tree,0,newTree,0,tree.length);
            tree = newTree;
        }
        nodeNum = this.nyts;
    }
    // Change old NYT
    tmpNode = tree[nodeNum];
    tmpNode.type = t_branch;
    tmpNode.child = nodeNum+1;

    // Create symbol node
    tmpNode = new Node();
    tmpNode.type = t_symbol;
    tmpNode.symbol = c;
    tmpNode.parent = nodeNum;
    tmpNode.weight = 1;
    tree[nodeNum+1] = tmpNode;
    nodeList[c] = nodeNum + 1;

    // Create new NYT
    tmpNode = new Node();
    tmpNode.type = t_nyt;
    tmpNode.parent = nodeNum;
    tree[nodeNum+2] = tmpNode;
    this.nyts = nodeNum + 2;
}
}

// Slide nodeNum to the front of its weight block
private int slideAndIncrement(int nodeNum) {
    Node nodeTmp = tree[nodeNum];
    int parent = nodeTmp.parent; // in the end this will carry the return value
    if (nodeNum > 0) {
        Node moveTmp = null;
        int weight = nodeTmp.weight;
        int moveNum = nodeNum;

        if (nodeTmp.type == t_branch) weight++;
        else parent = -2;

        while (weight >= tree[moveNum-1].weight && parent != moveNum-1 && nodeTmp.parent != moveNum-1) {
            moveNum--;
            // Shift Nodes
            moveTmp = tree[moveNum];
            tree[moveNum] = nodeTmp;
            tree[nodeNum] = moveTmp;
            int tmp = nodeTmp.parent;
            nodeTmp.parent = moveTmp.parent;
        }
    }
}
```
moveTmp.parent = tmp;
nodelist[nodeTmp.symbol] = moveNum;
nodelist[moveTmp.symbol] = nodeNum;
if (nodeTmp.type == t_branch) {
    tree[nodeTmp.child].parent = moveNum;
    tree[nodeTmp.child+1].parent = moveNum;
}  
if (moveTmp.type == t_branch) {
    tree[moveTmp.child].parent = nodeNum;
    tree[moveTmp.child+1].parent = nodeNum;
}  
nodeNum--;  
if (moveNum == 0) break;
}
if (nodeTmp.type != t_branch) {
    parent = nodeTmp.parent;
}
nodeTmp.weight++;
return parent;
}
// Convert a node Number to bits
private void nodeNumToBits(int nodeNum) {
    Stack<Integer> stack = new Stack<Integer>();
    int parent = tree[nodeNum].parent;
    while (parent != -1) {
        if (tree[parent].child == nodeNum) { // this is a right child (1)
            stack.push(new Integer(1));
        } else { // this is a left child (0)
            stack.push(new Integer(0));
        }
        nodeNum = parent;
        parent = tree[nodeNum].parent;
    }
    while (!stack.empty()) {
        bithndl.bit_addBit(stack.pop().intValue());
    }
}
// Convert a number of bits to a symbol
private int bitsToSymbol(String bits) {
    Node curNode = tree[0];
    int index;
    for (index=0;index<bits.length() && curNode.type == t_branch;index++) {
        if (bits.charAt(index) == '1') curNode = tree[curNode.child];
        else curNode = tree[curNode.child+1];
    }
    switch (curNode.type) {
        case t_branch: // Branch  
            return -1;
        case t_symbol: // Symbol  
            return curNode.symbol;
        case t_nyt: // NYT  
            if (bits.length() - index == 8) { // NYT + Symbol
                char c = 0;
                for (;index<bits.length();index++) {
                    c = (char)(c << 1);
                }
                return c;
            } else return -1;
    }
    return -1;
}
// Debug printout of tree
private void debugPrint() {
    int i = 0;
    do {

Appendix A. Code Listings

```java
System.out.println(tree[i]);
i++;
} while (tree[i] != null);

// Just some file handling, beginning, ending.
public static void main(String[] args) {
    try {
        Lambda l = new Lambda(256);
        File ofile = new File(args[2]);
        File ifile = new File(args[1]);
        FileInputStream fis = new FileInputStream(ifile);
        FileOutputStream fos = new FileOutputStream(ofile);
        byte[] b = new byte[fis.available()];
        fis.read(b);
        if (args[0].equalsIgnoreCase("d")) { // Decompress
            b = l.decompress(b);
        } else { // Compress
            long startTime = System.nanoTime();
            b = l.compress(b);
            if (args[0].equalsIgnoreCase("t")) {
                System.out.println("" + (System.nanoTime() - startTime));
            }
        }
        fos.write(b, 0, l.getNumOfBytes());
        fis.close();
    } catch (java.lang.ArrayIndexOutOfBoundsException ex) {
        error("not enough args");
    } catch (java.io.FileNotFoundException ex) {
        error("file not found");
    } catch (java.io.IOException ex) {
        error("i/o error");
    }
} public static void error(String m) {
    System.out.println("Usage: java Lambda [c/d] <inputfile> <outputfile>");
    System.out.println("Error: "+m);
}
```

A.6. mktest.c

File for creating random data files.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#define NUM_OF_EACH_FILE 10 // Number of files in each combination
#define NUM_OF_MIN_MAX 7 // Number of Min/Max combinations
#define NUM_OF_SIZES_LOG 14 // Number of sizes in Log interval
#define NUM_OF_SIZES 21 // Number of sizes in Small interval

// MIN-MAX = {all characters},
// {numbers + separator},
// {whole english alphabet and some other chars}
// third number is name of minmax
int minMax[][3] = {0.255.256}, {0.7.8}, {0.15.16}, {48, 58, 11}, {0.63, 64}, {0.71, 72}, {58, 126, 69};
```
Appendix A. Code Listings

```c
// SIZES = number of bytes in each file

int sizesLOG[2] = {32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144};

unsigned char randChar(unsigned char min, unsigned char max) {
    int diff = (int)((max + 1) - min);
    if (diff < 1) diff = 1;
    int r = rand() % diff + min;
    if (r > 255) r = 255;
    return (unsigned char)r;
}

int main() {
    FILE* f;
    char fileName[1024];
    int mx=0;
    int size=0;
    int fNum = 0;
    int cNum = 0;
    srand(time(NULL));

    for (mx=0; mx<NUM_OF_MIN_MAX; mx++) { // MIN MAX
        for (size=0; size<NUM_OF_SIZES; size++) { // SIZES
            for (fNum=0; fNum<NUM_OF_EACH_FILE; fNum++) { // FILE NUMBER
                sprintf(fileName, "test−%i−%i", minMax[mx][2], sizes[size], fNum);
                f = fopen(fileName, "w");
                for (cNum=0; cNum<sizes[size]; cNum++) {
                    if (putc(randChar(minMax[mx][0], minMax[mx][1]), f) == EOF) {
                        printf("Error when writing char %i to %s", cNum, fileName);
                        break;
                    }
                }
                fclose(f);
            }
        }
        for (size=0; size<NUM_OF_SIZES_LOG; size++) { // SIZES
            for (fNum=0; fNum<NUM_OF_EACH_FILE; fNum++) { // FILE NUMBER
                sprintf(fileName, "test−log−%i−%i", minMax[mx][2], sizesLOG[size], fNum);
                f = fopen(fileName, "w");
                for (cNum=0; cNum<sizesLOG[size]; cNum++) {
                    if (putc(randChar(minMax[mx][0], minMax[mx][1]), f) == EOF) {
                        printf("Error when writing char %i to %s", cNum, fileName);
                        break;
                    }
                }
                fclose(f);
            }
        }
    }
    return 0;
}
```

A.7. timetest.sh

File for running time tests on algorithms.
Appendix A. Code Listings

```bash
#!/bin/bash

# MINMAX NAMES (NUM CHARs IN ALPHABET)
MINMAX='8 11 16 64 69 72 256
# NUMBERS OF BYTES PER FILE
NB='100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300'

NBLOG='32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144'

# CORPUSES AVAILABLE
CORPUSES='artificial calgary cantrbry large'

# Functions for different algorithms
# Args: 1 - Files
function compress-lambda {
    TIME=0
    for FILE in $(ls $1); do
        TIMETMP=/grave.ts1
        java -classpath ./MyAlgs/ Lambda t $FILE $FILE.L
        TIME=echo "$TIME+$TIMETMP" | bc
        done
    rm $1.L
    echo $TIME
}

function compress-arithmetic {
    TIME=0
    for FILE in $(ls $1); do
        TIMETMP=/grave.ts1
        java -classpath ./MyAlgs/ Arithmetic t $FILE $FILE.A
        TIME=echo "$TIME+$TIMETMP" | bc
        done
    rm $1.A
    echo $TIME
}

# Run tests on algorithms
# Args: 1 - File Name
# 2 - Number of bytes in file
function runtests {
    FILECOUNT=`ls $1 | wc -w`
    # RUN TESTS
    TIME=compress-lambda "$1"
    LAMBDM=echo "$TIME/$FILECOUNT" | bc
    TIME=compress-arithmetic "$1"
    ARITHMETIC=echo "$TIME/$FILECOUNT" | bc
    echo "$2:$LAMBDM:$ARITHMETIC"
}

# RANDOM FILES
for MINMAX in $MINMAX; do
    # REGULAR INTERVAL
    for NUMBYTES in $NB; do
        FILESTRING=./Slumpifier/test-$MINMAX-$NUMBYTES-s
        runtests "$FILESTRING" $NUMBYTES >> result-time-$MINMAX.csv
    done
    echo "result-time-$MINMAX.csv done"
    # LOG INTERVAL
    for NUMBYTES in $NBLOG; do
        FILESTRING=./Slumpifier/test-log-$MINMAX-$NUMBYTES-s
        runtests "$FILESTRING" $NUMBYTES >> result-time-log-$MINMAX.csv
    done
    echo "result-time-log-$MINMAX.csv done"
    # CORPUSES
    for CORPUS in $CORPUSES; do
        FILESTRING=./Slumpifier/test-$CORPUS-$MINMAX-$NUMBYTES-s
        runtests "$FILESTRING" $NUMBYTES >> result-time-$CORPUS-$MINMAX.csv
    done
    echo "result-time-$CORPUS-$MINMAX.csv done"
    done
```
Appendix A. Code Listings

```bash
echo "Lambda [M]; Arithmetic [M]" > result-time-$CORPUS'.csv
for FILE in `ls ./$CORPUS/*/`; do
    echo 'runtests $FILE $FILE' >> result-time-$CORPUS'.csv
    done
    echo "result-time-$CORPUS.csv done"
    done
```
Appendix B.

Algorithm Implementations

The algorithms used in testing have been found at the following web addresses. All algorithms were available 2010-05-14.

**Huffman** [http://sourceforge.net/projects/huffman/files/huffman/1.1/huffman-1.1.tar.gz/download](http://sourceforge.net/projects/huffman/files/huffman/1.1/huffman-1.1.tar.gz/download)

**Lambda** [http://compression-code.googlecode.com/files/vitter.c](http://compression-code.googlecode.com/files/vitter.c)

**FGK** [http://sites.google.com/site/datacompressionguide/fgk/FGK.zip](http://sites.google.com/site/datacompressionguide/fgk/FGK.zip)


**LZW** The compress utility found in BSD Unix (Mac OS X 10.6.3)

**LZ77** [http://sites.google.com/site/datacompressionguide/lz77/LZSS.zip](http://sites.google.com/site/datacompressionguide/lz77/LZSS.zip)
Appendix C.

Description of publicly available corpuses

Some publicly available corpuses are used when testing the different implementations. In this appendix those are described.

C.1. Artificial Corpus

The Artificial Corpus contains files that stands out by being either non-repetitive or very repetitive. Compressing this corpus may therefore show the algorithm's worst-case behaviour.

- a.txt The single letter 'a'
- aaa.txt The letter 'a' repeated 100'000 times.
- alphabet.txt The english alphabet repeated until it fills 100’000 characters.
- random.txt 100’000 random characters from an alphabet of size 64.

C.2. Calgary Corpus

The Calgary Corpus is rather old in comparison to the others. Developed in the late 1980’s it contains files related to what was commonly available at that time. Nowadays it may seem a bit outdated but it is still "reasonably reliable as a performance indicator." [2]

- bib Bibliography file (reference format)
- book1 Fiction book
- book2 Non-fiction book in troff format
Appendix C. Description of publicly available corpuses

geo  Geophysical data
news  USENET batch file
obj1  Object code for VAX
obj2  Object code for Apple Mac
paper1  Technical paper
paper2  Technical paper
  pic  Black and white fax picture
progc  Source code in C
progl  Source code in LISP
progp  Source code in PASCAL
trans  Transcript of terminal session

C.3. Canterbury Corpus

The Canterbury Corpus is quite like the calgary corpus but is more modern.

alice29.txt  Alice’s Adventures in Wonderland by Lewis Carroll
asuryoulik.txt  As You Like It by Shakespeare
  cp.html  HTML source
fielsg.c  Source code in C
grammar.lsp  Source code in LISP
kennedy.xls  Excel spreadsheet
lcet10.txt  Technical writing
plrabn12.txt  Poetry, Paradise Lost by Joe Milton
  ptt5  CCIT test set (fax)
sum  SPARC executable
xargs.1  GNU manual page
Appendix C. Description of publicly available corpuses

C.4. Large Corpus

The Large Corpus consists of files with relatively larger size than the other corpuses. This is because some algorithms require larger sets of data to give a satisfactory compression. Also, it may be easier to measure time differences on larger files.

- **E.coli** The complete genome of the E.Coli bacterium
- **bible.txt** The King James version of the bible
- **world192.txt** The CIA world fact book
Appendix D.

Results from pre-testing

D.1. Small Interval

<table>
<thead>
<tr>
<th>11 Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.14</td>
<td>1.67</td>
<td>1.37</td>
<td>1.3</td>
<td>1.28</td>
<td>1.08</td>
<td>0.81</td>
</tr>
<tr>
<td>110</td>
<td>1.19</td>
<td>1.72</td>
<td>1.42</td>
<td>1.35</td>
<td>1.3</td>
<td>1.09</td>
<td>0.82</td>
</tr>
<tr>
<td>120</td>
<td>1.23</td>
<td>1.74</td>
<td>1.47</td>
<td>1.4</td>
<td>1.32</td>
<td>1.1</td>
<td>0.84</td>
</tr>
<tr>
<td>130</td>
<td>1.29</td>
<td>1.78</td>
<td>1.51</td>
<td>1.45</td>
<td>1.34</td>
<td>1.14</td>
<td>0.86</td>
</tr>
<tr>
<td>140</td>
<td>1.33</td>
<td>1.8</td>
<td>1.54</td>
<td>1.49</td>
<td>1.35</td>
<td>1.15</td>
<td>0.87</td>
</tr>
<tr>
<td>150</td>
<td>1.36</td>
<td>1.82</td>
<td>1.57</td>
<td>1.52</td>
<td>1.37</td>
<td>1.16</td>
<td>0.88</td>
</tr>
<tr>
<td>160</td>
<td>1.4</td>
<td>1.85</td>
<td>1.6</td>
<td>1.55</td>
<td>1.38</td>
<td>1.18</td>
<td>0.89</td>
</tr>
<tr>
<td>170</td>
<td>1.43</td>
<td>1.87</td>
<td>1.62</td>
<td>1.58</td>
<td>1.4</td>
<td>1.18</td>
<td>0.9</td>
</tr>
<tr>
<td>180</td>
<td>1.46</td>
<td>1.89</td>
<td>1.65</td>
<td>1.61</td>
<td>1.41</td>
<td>1.21</td>
<td>0.92</td>
</tr>
<tr>
<td>190</td>
<td>1.49</td>
<td>1.9</td>
<td>1.67</td>
<td>1.64</td>
<td>1.43</td>
<td>1.22</td>
<td>0.92</td>
</tr>
<tr>
<td>200</td>
<td>1.52</td>
<td>1.92</td>
<td>1.7</td>
<td>1.67</td>
<td>1.44</td>
<td>1.23</td>
<td>0.94</td>
</tr>
<tr>
<td>210</td>
<td>1.54</td>
<td>1.94</td>
<td>1.72</td>
<td>1.69</td>
<td>1.45</td>
<td>1.24</td>
<td>0.94</td>
</tr>
<tr>
<td>220</td>
<td>1.56</td>
<td>1.94</td>
<td>1.73</td>
<td>1.7</td>
<td>1.46</td>
<td>1.26</td>
<td>0.95</td>
</tr>
<tr>
<td>230</td>
<td>1.58</td>
<td>1.96</td>
<td>1.75</td>
<td>1.73</td>
<td>1.47</td>
<td>1.26</td>
<td>0.96</td>
</tr>
<tr>
<td>240</td>
<td>1.61</td>
<td>1.96</td>
<td>1.76</td>
<td>1.74</td>
<td>1.49</td>
<td>1.27</td>
<td>0.96</td>
</tr>
<tr>
<td>250</td>
<td>1.63</td>
<td>1.98</td>
<td>1.78</td>
<td>1.76</td>
<td>1.49</td>
<td>1.29</td>
<td>0.97</td>
</tr>
<tr>
<td>260</td>
<td>1.64</td>
<td>1.99</td>
<td>1.79</td>
<td>1.78</td>
<td>1.51</td>
<td>1.3</td>
<td>0.97</td>
</tr>
<tr>
<td>270</td>
<td>1.66</td>
<td>1.99</td>
<td>1.8</td>
<td>1.79</td>
<td>1.51</td>
<td>1.3</td>
<td>0.99</td>
</tr>
<tr>
<td>280</td>
<td>1.68</td>
<td>2.01</td>
<td>1.81</td>
<td>1.81</td>
<td>1.52</td>
<td>1.32</td>
<td>0.99</td>
</tr>
<tr>
<td>290</td>
<td>1.69</td>
<td>2.01</td>
<td>1.82</td>
<td>1.82</td>
<td>1.53</td>
<td>1.33</td>
<td>1</td>
</tr>
<tr>
<td>300</td>
<td>1.71</td>
<td>2.02</td>
<td>1.84</td>
<td>1.83</td>
<td>1.54</td>
<td>1.33</td>
<td>1</td>
</tr>
</tbody>
</table>
## Appendix D. Results from pre-testing

<table>
<thead>
<tr>
<th>64 Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.43</td>
<td>0.83</td>
<td>0.73</td>
<td>0.47</td>
<td>1.03</td>
<td>0.87</td>
<td>0.71</td>
</tr>
<tr>
<td>110</td>
<td>0.45</td>
<td>0.84</td>
<td>0.74</td>
<td>0.49</td>
<td>1.04</td>
<td>0.88</td>
<td>0.72</td>
</tr>
<tr>
<td>120</td>
<td>0.47</td>
<td>0.86</td>
<td>0.76</td>
<td>0.51</td>
<td>1.05</td>
<td>0.88</td>
<td>0.73</td>
</tr>
<tr>
<td>130</td>
<td>0.48</td>
<td>0.87</td>
<td>0.78</td>
<td>0.52</td>
<td>1.05</td>
<td>0.88</td>
<td>0.74</td>
</tr>
<tr>
<td>140</td>
<td>0.5</td>
<td>0.88</td>
<td>0.8</td>
<td>0.54</td>
<td>1.05</td>
<td>0.88</td>
<td>0.74</td>
</tr>
<tr>
<td>150</td>
<td>0.52</td>
<td>0.9</td>
<td>0.82</td>
<td>0.56</td>
<td>1.06</td>
<td>0.88</td>
<td>0.74</td>
</tr>
<tr>
<td>160</td>
<td>0.53</td>
<td>0.92</td>
<td>0.83</td>
<td>0.58</td>
<td>1.07</td>
<td>0.89</td>
<td>0.75</td>
</tr>
<tr>
<td>170</td>
<td>0.54</td>
<td>0.92</td>
<td>0.84</td>
<td>0.59</td>
<td>1.06</td>
<td>0.89</td>
<td>0.75</td>
</tr>
<tr>
<td>180</td>
<td>0.55</td>
<td>0.93</td>
<td>0.85</td>
<td>0.6</td>
<td>1.07</td>
<td>0.89</td>
<td>0.75</td>
</tr>
<tr>
<td>190</td>
<td>0.57</td>
<td>0.95</td>
<td>0.87</td>
<td>0.62</td>
<td>1.07</td>
<td>0.89</td>
<td>0.76</td>
</tr>
<tr>
<td>200</td>
<td>0.58</td>
<td>0.95</td>
<td>0.87</td>
<td>0.63</td>
<td>1.08</td>
<td>0.9</td>
<td>0.76</td>
</tr>
<tr>
<td>210</td>
<td>0.61</td>
<td>0.97</td>
<td>0.89</td>
<td>0.65</td>
<td>1.08</td>
<td>0.9</td>
<td>0.76</td>
</tr>
<tr>
<td>220</td>
<td>0.61</td>
<td>0.97</td>
<td>0.9</td>
<td>0.66</td>
<td>1.09</td>
<td>0.9</td>
<td>0.76</td>
</tr>
<tr>
<td>230</td>
<td>0.63</td>
<td>0.98</td>
<td>0.91</td>
<td>0.67</td>
<td>1.09</td>
<td>0.9</td>
<td>0.77</td>
</tr>
<tr>
<td>240</td>
<td>0.64</td>
<td>0.99</td>
<td>0.92</td>
<td>0.68</td>
<td>1.09</td>
<td>0.9</td>
<td>0.77</td>
</tr>
<tr>
<td>250</td>
<td>0.66</td>
<td>1</td>
<td>0.93</td>
<td>0.7</td>
<td>1.09</td>
<td>0.9</td>
<td>0.77</td>
</tr>
<tr>
<td>260</td>
<td>0.66</td>
<td>1.01</td>
<td>0.94</td>
<td>0.71</td>
<td>1.1</td>
<td>0.9</td>
<td>0.77</td>
</tr>
<tr>
<td>270</td>
<td>0.67</td>
<td>1.01</td>
<td>0.95</td>
<td>0.72</td>
<td>1.1</td>
<td>0.9</td>
<td>0.78</td>
</tr>
<tr>
<td>280</td>
<td>0.69</td>
<td>1.02</td>
<td>0.96</td>
<td>0.73</td>
<td>1.1</td>
<td>0.89</td>
<td>0.77</td>
</tr>
<tr>
<td>290</td>
<td>0.7</td>
<td>1.04</td>
<td>0.97</td>
<td>0.75</td>
<td>1.11</td>
<td>0.9</td>
<td>0.78</td>
</tr>
<tr>
<td>300</td>
<td>0.71</td>
<td>1.04</td>
<td>0.97</td>
<td>0.75</td>
<td>1.11</td>
<td>0.9</td>
<td>0.78</td>
</tr>
</tbody>
</table>
### Appendix D. Results from pre-testing

<table>
<thead>
<tr>
<th>256 Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.29</td>
<td>0.63</td>
<td>0.56</td>
<td>0.32</td>
<td>0.97</td>
<td>0.86</td>
<td>0.71</td>
</tr>
<tr>
<td>110</td>
<td>0.3</td>
<td>0.64</td>
<td>0.57</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.71</td>
</tr>
<tr>
<td>120</td>
<td>0.3</td>
<td>0.64</td>
<td>0.57</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.72</td>
</tr>
<tr>
<td>130</td>
<td>0.3</td>
<td>0.63</td>
<td>0.57</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.73</td>
</tr>
<tr>
<td>140</td>
<td>0.31</td>
<td>0.64</td>
<td>0.58</td>
<td>0.34</td>
<td>0.98</td>
<td>0.87</td>
<td>0.73</td>
</tr>
<tr>
<td>150</td>
<td>0.31</td>
<td>0.64</td>
<td>0.58</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.74</td>
</tr>
<tr>
<td>160</td>
<td>0.32</td>
<td>0.64</td>
<td>0.59</td>
<td>0.34</td>
<td>0.98</td>
<td>0.87</td>
<td>0.74</td>
</tr>
<tr>
<td>170</td>
<td>0.32</td>
<td>0.65</td>
<td>0.59</td>
<td>0.35</td>
<td>0.98</td>
<td>0.87</td>
<td>0.74</td>
</tr>
<tr>
<td>180</td>
<td>0.32</td>
<td>0.65</td>
<td>0.59</td>
<td>0.35</td>
<td>0.98</td>
<td>0.88</td>
<td>0.75</td>
</tr>
<tr>
<td>190</td>
<td>0.33</td>
<td>0.65</td>
<td>0.6</td>
<td>0.35</td>
<td>0.98</td>
<td>0.88</td>
<td>0.75</td>
</tr>
<tr>
<td>200</td>
<td>0.33</td>
<td>0.66</td>
<td>0.6</td>
<td>0.36</td>
<td>0.98</td>
<td>0.88</td>
<td>0.75</td>
</tr>
<tr>
<td>210</td>
<td>0.33</td>
<td>0.66</td>
<td>0.6</td>
<td>0.36</td>
<td>0.98</td>
<td>0.88</td>
<td>0.75</td>
</tr>
<tr>
<td>220</td>
<td>0.33</td>
<td>0.65</td>
<td>0.6</td>
<td>0.36</td>
<td>0.97</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>230</td>
<td>0.34</td>
<td>0.66</td>
<td>0.6</td>
<td>0.36</td>
<td>0.98</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>240</td>
<td>0.35</td>
<td>0.67</td>
<td>0.61</td>
<td>0.37</td>
<td>0.98</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>250</td>
<td>0.35</td>
<td>0.67</td>
<td>0.62</td>
<td>0.37</td>
<td>0.98</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>260</td>
<td>0.35</td>
<td>0.68</td>
<td>0.62</td>
<td>0.38</td>
<td>0.98</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>270</td>
<td>0.36</td>
<td>0.68</td>
<td>0.62</td>
<td>0.38</td>
<td>0.98</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>280</td>
<td>0.36</td>
<td>0.68</td>
<td>0.63</td>
<td>0.39</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
</tr>
<tr>
<td>290</td>
<td>0.36</td>
<td>0.69</td>
<td>0.63</td>
<td>0.39</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
</tr>
<tr>
<td>300</td>
<td>0.36</td>
<td>0.69</td>
<td>0.63</td>
<td>0.39</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
</tr>
</tbody>
</table>
D.2. Logarithmic Interval

<table>
<thead>
<tr>
<th>11 Characters</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>0.61</td>
<td>1.17</td>
<td>0.85</td>
<td>0.75</td>
<td>1.09</td>
<td>0.91</td>
<td>0.62</td>
</tr>
<tr>
<td>64</td>
<td>0.88</td>
<td>1.45</td>
<td>1.15</td>
<td>1.04</td>
<td>1.19</td>
<td>0.98</td>
<td>0.72</td>
</tr>
<tr>
<td>128</td>
<td>1.27</td>
<td>1.76</td>
<td>1.49</td>
<td>1.43</td>
<td>1.33</td>
<td>1.12</td>
<td>0.85</td>
</tr>
<tr>
<td>256</td>
<td>1.64</td>
<td>1.99</td>
<td>1.79</td>
<td>1.77</td>
<td>1.5</td>
<td>1.29</td>
<td>0.98</td>
</tr>
<tr>
<td>512</td>
<td>1.91</td>
<td>2.12</td>
<td>1.98</td>
<td>2.01</td>
<td>1.68</td>
<td>1.46</td>
<td>1.09</td>
</tr>
<tr>
<td>1024</td>
<td>2.07</td>
<td>2.18</td>
<td>2.08</td>
<td>2.15</td>
<td>1.85</td>
<td>1.58</td>
<td>1.18</td>
</tr>
<tr>
<td>2048</td>
<td>2.17</td>
<td>2.22</td>
<td>2.14</td>
<td>2.23</td>
<td>2</td>
<td>1.69</td>
<td>1.28</td>
</tr>
<tr>
<td>4096</td>
<td>2.21</td>
<td>2.24</td>
<td>2.17</td>
<td>2.27</td>
<td>2.11</td>
<td>1.8</td>
<td>1.38</td>
</tr>
<tr>
<td>8192</td>
<td>2.24</td>
<td>2.25</td>
<td>2.19</td>
<td>2.29</td>
<td>2.19</td>
<td>1.86</td>
<td>1.45</td>
</tr>
<tr>
<td>16384</td>
<td>2.25</td>
<td>2.25</td>
<td>2.19</td>
<td>2.3</td>
<td>2.24</td>
<td>1.92</td>
<td>1.49</td>
</tr>
<tr>
<td>32768</td>
<td>2.25</td>
<td>2.25</td>
<td>2.2</td>
<td>2.31</td>
<td>2.27</td>
<td>1.95</td>
<td>1.51</td>
</tr>
<tr>
<td>65536</td>
<td>2.26</td>
<td>2.25</td>
<td>2.2</td>
<td>2.31</td>
<td>2.28</td>
<td>1.98</td>
<td>1.52</td>
</tr>
<tr>
<td>131072</td>
<td>2.26</td>
<td>2.26</td>
<td>2.2</td>
<td>2.31</td>
<td>2.29</td>
<td>2.02</td>
<td>1.52</td>
</tr>
<tr>
<td>262144</td>
<td>2.26</td>
<td>2.26</td>
<td>2.2</td>
<td>2.31</td>
<td>2.29</td>
<td>2.05</td>
<td>1.53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64 Characters</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>0.3</td>
<td>0.69</td>
<td>0.55</td>
<td>0.35</td>
<td>0.98</td>
<td>0.82</td>
<td>0.59</td>
</tr>
<tr>
<td>64</td>
<td>0.36</td>
<td>0.76</td>
<td>0.64</td>
<td>0.41</td>
<td>1.01</td>
<td>0.86</td>
<td>0.67</td>
</tr>
<tr>
<td>128</td>
<td>0.47</td>
<td>0.86</td>
<td>0.76</td>
<td>0.51</td>
<td>1.04</td>
<td>0.88</td>
<td>0.73</td>
</tr>
<tr>
<td>256</td>
<td>0.66</td>
<td>1.01</td>
<td>0.94</td>
<td>0.7</td>
<td>1.1</td>
<td>0.91</td>
<td>0.78</td>
</tr>
<tr>
<td>512</td>
<td>0.88</td>
<td>1.13</td>
<td>1.08</td>
<td>0.92</td>
<td>1.15</td>
<td>0.89</td>
<td>0.81</td>
</tr>
<tr>
<td>1024</td>
<td>1.06</td>
<td>1.22</td>
<td>1.19</td>
<td>1.09</td>
<td>1.2</td>
<td>0.89</td>
<td>0.84</td>
</tr>
<tr>
<td>2048</td>
<td>1.18</td>
<td>1.27</td>
<td>1.25</td>
<td>1.2</td>
<td>1.24</td>
<td>0.91</td>
<td>0.88</td>
</tr>
<tr>
<td>4096</td>
<td>1.25</td>
<td>1.3</td>
<td>1.29</td>
<td>1.26</td>
<td>1.28</td>
<td>0.93</td>
<td>0.94</td>
</tr>
<tr>
<td>8192</td>
<td>1.29</td>
<td>1.32</td>
<td>1.31</td>
<td>1.3</td>
<td>1.3</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>16384</td>
<td>1.31</td>
<td>1.32</td>
<td>1.32</td>
<td>1.31</td>
<td>1.31</td>
<td>1.04</td>
<td>1.02</td>
</tr>
<tr>
<td>32768</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
<td>1.08</td>
<td>1.03</td>
</tr>
<tr>
<td>65536</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.32</td>
<td>1.09</td>
<td>1.04</td>
</tr>
<tr>
<td>131072</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.09</td>
<td>1.04</td>
</tr>
<tr>
<td>262144</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.09</td>
<td>1.05</td>
</tr>
</tbody>
</table>
### Appendix D. Results from pre-testing

<table>
<thead>
<tr>
<th>256 Characters</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>0.26</td>
<td>0.61</td>
<td>0.49</td>
<td>0.3</td>
<td>0.95</td>
<td>0.82</td>
<td>0.59</td>
</tr>
<tr>
<td>64</td>
<td>0.28</td>
<td>0.63</td>
<td>0.54</td>
<td>0.31</td>
<td>0.97</td>
<td>0.85</td>
<td>0.67</td>
</tr>
<tr>
<td>128</td>
<td>0.3</td>
<td>0.64</td>
<td>0.57</td>
<td>0.33</td>
<td>0.97</td>
<td>0.87</td>
<td>0.73</td>
</tr>
<tr>
<td>256</td>
<td>0.35</td>
<td>0.67</td>
<td>0.62</td>
<td>0.37</td>
<td>0.98</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>512</td>
<td>0.41</td>
<td>0.75</td>
<td>0.69</td>
<td>0.47</td>
<td>0.98</td>
<td>0.84</td>
<td>0.78</td>
</tr>
<tr>
<td>1024</td>
<td>0.56</td>
<td>0.83</td>
<td>0.79</td>
<td>0.6</td>
<td>0.98</td>
<td>0.8</td>
<td>0.79</td>
</tr>
<tr>
<td>2048</td>
<td>0.72</td>
<td>0.9</td>
<td>0.87</td>
<td>0.75</td>
<td>0.98</td>
<td>0.77</td>
<td>0.8</td>
</tr>
<tr>
<td>4096</td>
<td>0.84</td>
<td>0.94</td>
<td>0.93</td>
<td>0.86</td>
<td>0.99</td>
<td>0.73</td>
<td>0.81</td>
</tr>
<tr>
<td>8192</td>
<td>0.91</td>
<td>0.97</td>
<td>0.96</td>
<td>0.92</td>
<td>0.99</td>
<td>0.7</td>
<td>0.82</td>
</tr>
<tr>
<td>16384</td>
<td>0.95</td>
<td>0.98</td>
<td>0.98</td>
<td>0.96</td>
<td>1</td>
<td>0.68</td>
<td>0.82</td>
</tr>
<tr>
<td>32768</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
<td>0.98</td>
<td>1</td>
<td>0.68</td>
<td>0.83</td>
</tr>
<tr>
<td>65536</td>
<td>0.99</td>
<td>1</td>
<td>0.99</td>
<td>0.99</td>
<td>1</td>
<td>0.71</td>
<td>0.83</td>
</tr>
<tr>
<td>131072</td>
<td>0.99</td>
<td>1</td>
<td>1</td>
<td>0.99</td>
<td>1</td>
<td>0.75</td>
<td>0.83</td>
</tr>
<tr>
<td>262144</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.75</td>
<td>0.83</td>
</tr>
</tbody>
</table>
Appendix D. Results from pre-testing

D.3. Corporuses

<table>
<thead>
<tr>
<th>Artifical</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.txt</td>
<td>0.1</td>
<td>0.17</td>
<td>0.11</td>
<td>0.17</td>
<td>0.33</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>aaa.txt</td>
<td>0</td>
<td>8</td>
<td>7.99</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.46</td>
</tr>
<tr>
<td>alphabet.txt</td>
<td>1.68</td>
<td>1.59</td>
<td>1.66</td>
<td>1.7</td>
<td>1.69</td>
<td>0</td>
<td>8.44</td>
</tr>
<tr>
<td>random.txt</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.08</td>
<td>1.04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calgary</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>1.52</td>
<td>1.53</td>
<td>1.53</td>
<td>1.53</td>
<td>1.53</td>
<td>2.39</td>
<td>2.16</td>
</tr>
<tr>
<td>book1</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.77</td>
<td>1.76</td>
<td>2.32</td>
<td>1.84</td>
</tr>
<tr>
<td>book2</td>
<td>1.66</td>
<td>1.66</td>
<td>1.66</td>
<td>1.67</td>
<td>1.67</td>
<td>2.44</td>
<td>2.16</td>
</tr>
<tr>
<td>geo</td>
<td>1.39</td>
<td>1.41</td>
<td>1.4</td>
<td>1.4</td>
<td>1.41</td>
<td>1.32</td>
<td>1.21</td>
</tr>
<tr>
<td>news</td>
<td>1.53</td>
<td>1.53</td>
<td>1.53</td>
<td>1.54</td>
<td>1.54</td>
<td>2.07</td>
<td>1.98</td>
</tr>
<tr>
<td>obj1</td>
<td>1.26</td>
<td>1.32</td>
<td>1.31</td>
<td>1.29</td>
<td>1.34</td>
<td>1.53</td>
<td>1.75</td>
</tr>
<tr>
<td>obj2</td>
<td>1.27</td>
<td>1.27</td>
<td>1.27</td>
<td>1.27</td>
<td>1.32</td>
<td>1.92</td>
<td>2.39</td>
</tr>
<tr>
<td>paper1</td>
<td>1.58</td>
<td>1.59</td>
<td>1.59</td>
<td>1.59</td>
<td>1.6</td>
<td>2.12</td>
<td>2.19</td>
</tr>
<tr>
<td>paper2</td>
<td>1.71</td>
<td>1.72</td>
<td>1.72</td>
<td>1.73</td>
<td>1.73</td>
<td>2.27</td>
<td>2.1</td>
</tr>
<tr>
<td>paper3</td>
<td>1.69</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>2.1</td>
<td>2.01</td>
</tr>
<tr>
<td>paper4</td>
<td>1.63</td>
<td>1.67</td>
<td>1.66</td>
<td>1.65</td>
<td>1.66</td>
<td>1.91</td>
<td>1.98</td>
</tr>
<tr>
<td>paper5</td>
<td>1.54</td>
<td>1.58</td>
<td>1.58</td>
<td>1.57</td>
<td>1.58</td>
<td>1.82</td>
<td>1.98</td>
</tr>
<tr>
<td>paper6</td>
<td>1.56</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.6</td>
<td>2.04</td>
<td>2.19</td>
</tr>
<tr>
<td>pic</td>
<td>4.79</td>
<td>4.81</td>
<td>4.81</td>
<td>6.55</td>
<td>6.86</td>
<td>8.25</td>
<td>4.91</td>
</tr>
<tr>
<td>progc</td>
<td>1.51</td>
<td>1.52</td>
<td>1.52</td>
<td>1.52</td>
<td>1.53</td>
<td>2.07</td>
<td>2.28</td>
</tr>
<tr>
<td>progl</td>
<td>1.65</td>
<td>1.66</td>
<td>1.66</td>
<td>1.67</td>
<td>1.68</td>
<td>2.64</td>
<td>3.21</td>
</tr>
<tr>
<td>progp</td>
<td>1.62</td>
<td>1.63</td>
<td>1.63</td>
<td>1.63</td>
<td>1.63</td>
<td>2.57</td>
<td>3.22</td>
</tr>
<tr>
<td>trans</td>
<td>1.43</td>
<td>1.43</td>
<td>1.43</td>
<td>1.44</td>
<td>1.46</td>
<td>2.45</td>
<td>2.83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Canterbury</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>1.73</td>
<td>1.73</td>
<td>1.73</td>
<td>1.75</td>
<td>1.74</td>
<td>2.44</td>
<td>2.11</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>1.65</td>
<td>1.65</td>
<td>1.65</td>
<td>1.66</td>
<td>1.65</td>
<td>2.28</td>
<td>1.93</td>
</tr>
<tr>
<td>cp.html</td>
<td>1.49</td>
<td>1.51</td>
<td>1.51</td>
<td>1.51</td>
<td>1.51</td>
<td>2.17</td>
<td>2.3</td>
</tr>
<tr>
<td>fields.c</td>
<td>1.52</td>
<td>1.56</td>
<td>1.56</td>
<td>1.54</td>
<td>1.56</td>
<td>2.25</td>
<td>2.91</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>1.52</td>
<td>1.65</td>
<td>1.64</td>
<td>1.57</td>
<td>1.62</td>
<td>2.05</td>
<td>2.41</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.23</td>
<td>2.37</td>
<td>3.32</td>
<td>3.15</td>
</tr>
<tr>
<td>lceil10.txt</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.71</td>
<td>1.72</td>
<td>2.62</td>
<td>2.16</td>
</tr>
<tr>
<td>plrabin12.txt</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.76</td>
<td>1.76</td>
<td>2.37</td>
<td>1.85</td>
</tr>
<tr>
<td>ptt5</td>
<td>4.79</td>
<td>4.81</td>
<td>4.81</td>
<td>6.55</td>
<td>6.86</td>
<td>8.25</td>
<td>4.91</td>
</tr>
<tr>
<td>sum</td>
<td>1.44</td>
<td>1.47</td>
<td>1.47</td>
<td>1.46</td>
<td>1.55</td>
<td>1.9</td>
<td>2.14</td>
</tr>
<tr>
<td>xargs.1</td>
<td>1.47</td>
<td>1.57</td>
<td>1.56</td>
<td>1.51</td>
<td>1.54</td>
<td>1.81</td>
<td>2.02</td>
</tr>
</tbody>
</table>
### Appendix D. Results from pre-testing

<table>
<thead>
<tr>
<th>Large</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.coli</td>
<td>4</td>
<td>3.56</td>
<td>3.56</td>
<td>4</td>
<td>3.94</td>
<td>3.69</td>
<td>2.88</td>
</tr>
<tr>
<td>bible.txt</td>
<td>1.82</td>
<td>1.82</td>
<td>1.82</td>
<td>1.84</td>
<td>1.84</td>
<td>2.89</td>
<td>2.5</td>
</tr>
<tr>
<td>world192.txt</td>
<td>1.59</td>
<td>1.59</td>
<td>1.59</td>
<td>1.6</td>
<td>1.6</td>
<td>2.51</td>
<td>1.93</td>
</tr>
</tbody>
</table>
Appendix E.

Results from post-testing

E.1. Compression Ratio

E.1.1. Small Interval

<table>
<thead>
<tr>
<th>11 Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.19</td>
<td>1.67</td>
<td>1.54</td>
<td>1.3</td>
<td>1.28</td>
<td>1.08</td>
<td>0.86</td>
<td>1.77</td>
<td>1.28</td>
</tr>
<tr>
<td>110</td>
<td>1.24</td>
<td>1.72</td>
<td>1.59</td>
<td>1.35</td>
<td>1.3</td>
<td>1.09</td>
<td>0.87</td>
<td>1.8</td>
<td>1.3</td>
</tr>
<tr>
<td>120</td>
<td>1.29</td>
<td>1.74</td>
<td>1.63</td>
<td>1.4</td>
<td>1.32</td>
<td>1.1</td>
<td>0.89</td>
<td>1.83</td>
<td>1.32</td>
</tr>
<tr>
<td>130</td>
<td>1.34</td>
<td>1.78</td>
<td>1.66</td>
<td>1.45</td>
<td>1.34</td>
<td>1.14</td>
<td>0.91</td>
<td>1.85</td>
<td>1.34</td>
</tr>
<tr>
<td>140</td>
<td>1.38</td>
<td>1.8</td>
<td>1.69</td>
<td>1.49</td>
<td>1.35</td>
<td>1.15</td>
<td>0.92</td>
<td>1.88</td>
<td>1.35</td>
</tr>
<tr>
<td>150</td>
<td>1.41</td>
<td>1.82</td>
<td>1.71</td>
<td>1.52</td>
<td>1.37</td>
<td>1.16</td>
<td>0.92</td>
<td>1.9</td>
<td>1.37</td>
</tr>
<tr>
<td>160</td>
<td>1.45</td>
<td>1.85</td>
<td>1.74</td>
<td>1.55</td>
<td>1.38</td>
<td>1.18</td>
<td>0.93</td>
<td>1.91</td>
<td>1.38</td>
</tr>
<tr>
<td>170</td>
<td>1.48</td>
<td>1.87</td>
<td>1.76</td>
<td>1.58</td>
<td>1.4</td>
<td>1.18</td>
<td>0.94</td>
<td>1.92</td>
<td>1.4</td>
</tr>
<tr>
<td>180</td>
<td>1.51</td>
<td>1.89</td>
<td>1.78</td>
<td>1.61</td>
<td>1.41</td>
<td>1.21</td>
<td>0.96</td>
<td>1.94</td>
<td>1.41</td>
</tr>
<tr>
<td>190</td>
<td>1.54</td>
<td>1.9</td>
<td>1.8</td>
<td>1.64</td>
<td>1.43</td>
<td>1.22</td>
<td>0.96</td>
<td>1.95</td>
<td>1.43</td>
</tr>
<tr>
<td>200</td>
<td>1.57</td>
<td>1.92</td>
<td>1.82</td>
<td>1.67</td>
<td>1.44</td>
<td>1.23</td>
<td>0.98</td>
<td>1.96</td>
<td>1.44</td>
</tr>
<tr>
<td>210</td>
<td>1.59</td>
<td>1.94</td>
<td>1.85</td>
<td>1.69</td>
<td>1.45</td>
<td>1.24</td>
<td>0.98</td>
<td>1.98</td>
<td>1.45</td>
</tr>
<tr>
<td>220</td>
<td>1.61</td>
<td>1.94</td>
<td>1.84</td>
<td>1.7</td>
<td>1.46</td>
<td>1.26</td>
<td>0.98</td>
<td>1.98</td>
<td>1.46</td>
</tr>
<tr>
<td>230</td>
<td>1.63</td>
<td>1.96</td>
<td>1.86</td>
<td>1.73</td>
<td>1.47</td>
<td>1.26</td>
<td>0.99</td>
<td>1.99</td>
<td>1.47</td>
</tr>
<tr>
<td>240</td>
<td>1.65</td>
<td>1.96</td>
<td>1.87</td>
<td>1.74</td>
<td>1.49</td>
<td>1.27</td>
<td>0.99</td>
<td>2.01</td>
<td>1.48</td>
</tr>
<tr>
<td>250</td>
<td>1.67</td>
<td>1.98</td>
<td>1.89</td>
<td>1.76</td>
<td>1.49</td>
<td>1.29</td>
<td>1</td>
<td>2.01</td>
<td>1.49</td>
</tr>
<tr>
<td>260</td>
<td>1.69</td>
<td>1.99</td>
<td>1.9</td>
<td>1.78</td>
<td>1.51</td>
<td>1.3</td>
<td>1</td>
<td>2.01</td>
<td>1.51</td>
</tr>
<tr>
<td>270</td>
<td>1.7</td>
<td>1.99</td>
<td>1.9</td>
<td>1.79</td>
<td>1.51</td>
<td>1.3</td>
<td>1.02</td>
<td>2.01</td>
<td>1.51</td>
</tr>
<tr>
<td>280</td>
<td>1.72</td>
<td>2.01</td>
<td>1.91</td>
<td>1.81</td>
<td>1.52</td>
<td>1.32</td>
<td>1.02</td>
<td>2.02</td>
<td>1.52</td>
</tr>
<tr>
<td>290</td>
<td>1.73</td>
<td>2.01</td>
<td>1.92</td>
<td>1.82</td>
<td>1.53</td>
<td>1.33</td>
<td>1.02</td>
<td>2.03</td>
<td>1.53</td>
</tr>
<tr>
<td>300</td>
<td>1.75</td>
<td>2.02</td>
<td>1.93</td>
<td>1.83</td>
<td>1.54</td>
<td>1.33</td>
<td>1.03</td>
<td>2.04</td>
<td>1.54</td>
</tr>
</tbody>
</table>
## Appendix E. Results from post-testing

<table>
<thead>
<tr>
<th>64 Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.44</td>
<td>0.83</td>
<td>0.77</td>
<td>0.47</td>
<td>1.03</td>
<td>0.87</td>
<td>0.75</td>
<td>0.84</td>
<td>1.03</td>
</tr>
<tr>
<td>110</td>
<td>0.45</td>
<td>0.84</td>
<td>0.79</td>
<td>0.49</td>
<td>1.04</td>
<td>0.88</td>
<td>0.76</td>
<td>0.85</td>
<td>1.04</td>
</tr>
<tr>
<td>120</td>
<td>0.47</td>
<td>0.86</td>
<td>0.8</td>
<td>0.51</td>
<td>1.05</td>
<td>0.88</td>
<td>0.76</td>
<td>0.87</td>
<td>1.05</td>
</tr>
<tr>
<td>130</td>
<td>0.49</td>
<td>0.87</td>
<td>0.82</td>
<td>0.52</td>
<td>1.05</td>
<td>0.88</td>
<td>0.77</td>
<td>0.88</td>
<td>1.05</td>
</tr>
<tr>
<td>140</td>
<td>0.5</td>
<td>0.88</td>
<td>0.83</td>
<td>0.54</td>
<td>1.05</td>
<td>0.88</td>
<td>0.77</td>
<td>0.89</td>
<td>1.05</td>
</tr>
<tr>
<td>150</td>
<td>0.52</td>
<td>0.9</td>
<td>0.85</td>
<td>0.56</td>
<td>1.06</td>
<td>0.88</td>
<td>0.77</td>
<td>0.91</td>
<td>1.06</td>
</tr>
<tr>
<td>160</td>
<td>0.54</td>
<td>0.92</td>
<td>0.87</td>
<td>0.58</td>
<td>1.07</td>
<td>0.89</td>
<td>0.78</td>
<td>0.93</td>
<td>1.07</td>
</tr>
<tr>
<td>170</td>
<td>0.55</td>
<td>0.92</td>
<td>0.87</td>
<td>0.59</td>
<td>1.06</td>
<td>0.89</td>
<td>0.78</td>
<td>0.93</td>
<td>1.06</td>
</tr>
<tr>
<td>180</td>
<td>0.56</td>
<td>0.93</td>
<td>0.88</td>
<td>0.6</td>
<td>1.07</td>
<td>0.89</td>
<td>0.78</td>
<td>0.93</td>
<td>1.07</td>
</tr>
<tr>
<td>190</td>
<td>0.58</td>
<td>0.95</td>
<td>0.9</td>
<td>0.62</td>
<td>1.07</td>
<td>0.89</td>
<td>0.78</td>
<td>0.95</td>
<td>1.07</td>
</tr>
<tr>
<td>200</td>
<td>0.59</td>
<td>0.95</td>
<td>0.91</td>
<td>0.63</td>
<td>1.08</td>
<td>0.9</td>
<td>0.79</td>
<td>0.95</td>
<td>1.08</td>
</tr>
<tr>
<td>210</td>
<td>0.61</td>
<td>0.97</td>
<td>0.92</td>
<td>0.65</td>
<td>1.08</td>
<td>0.9</td>
<td>0.79</td>
<td>0.97</td>
<td>1.08</td>
</tr>
<tr>
<td>220</td>
<td>0.62</td>
<td>0.97</td>
<td>0.93</td>
<td>0.66</td>
<td>1.09</td>
<td>0.9</td>
<td>0.79</td>
<td>0.98</td>
<td>1.09</td>
</tr>
<tr>
<td>230</td>
<td>0.64</td>
<td>0.98</td>
<td>0.94</td>
<td>0.67</td>
<td>1.09</td>
<td>0.9</td>
<td>0.79</td>
<td>0.99</td>
<td>1.09</td>
</tr>
<tr>
<td>240</td>
<td>0.65</td>
<td>0.99</td>
<td>0.95</td>
<td>0.68</td>
<td>1.09</td>
<td>0.9</td>
<td>0.79</td>
<td>0.99</td>
<td>1.09</td>
</tr>
<tr>
<td>250</td>
<td>0.66</td>
<td>1</td>
<td>0.96</td>
<td>0.7</td>
<td>1.09</td>
<td>0.9</td>
<td>0.79</td>
<td>1</td>
<td>1.09</td>
</tr>
<tr>
<td>260</td>
<td>0.67</td>
<td>1.01</td>
<td>0.97</td>
<td>0.71</td>
<td>1.1</td>
<td>0.9</td>
<td>0.79</td>
<td>1.01</td>
<td>1.1</td>
</tr>
<tr>
<td>270</td>
<td>0.68</td>
<td>1.01</td>
<td>0.98</td>
<td>0.72</td>
<td>1.1</td>
<td>0.9</td>
<td>0.8</td>
<td>1.01</td>
<td>1.1</td>
</tr>
<tr>
<td>280</td>
<td>0.69</td>
<td>1.02</td>
<td>0.98</td>
<td>0.73</td>
<td>1.1</td>
<td>0.89</td>
<td>0.79</td>
<td>1.02</td>
<td>1.1</td>
</tr>
<tr>
<td>290</td>
<td>0.71</td>
<td>1.04</td>
<td>1</td>
<td>0.75</td>
<td>1.11</td>
<td>0.9</td>
<td>0.8</td>
<td>1.04</td>
<td>1.11</td>
</tr>
<tr>
<td>300</td>
<td>0.71</td>
<td>1.04</td>
<td>1</td>
<td>0.75</td>
<td>1.11</td>
<td>0.9</td>
<td>0.8</td>
<td>1.04</td>
<td>1.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>256 Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.3</td>
<td>0.63</td>
<td>0.59</td>
<td>0.32</td>
<td>0.97</td>
<td>0.86</td>
<td>0.75</td>
<td>0.63</td>
<td>0.97</td>
</tr>
<tr>
<td>110</td>
<td>0.3</td>
<td>0.64</td>
<td>0.59</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.75</td>
<td>0.63</td>
<td>0.98</td>
</tr>
<tr>
<td>120</td>
<td>0.31</td>
<td>0.64</td>
<td>0.59</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.76</td>
<td>0.63</td>
<td>0.98</td>
</tr>
<tr>
<td>130</td>
<td>0.31</td>
<td>0.63</td>
<td>0.59</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.76</td>
<td>0.63</td>
<td>0.98</td>
</tr>
<tr>
<td>140</td>
<td>0.31</td>
<td>0.64</td>
<td>0.6</td>
<td>0.34</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
<td>0.63</td>
<td>0.98</td>
</tr>
<tr>
<td>150</td>
<td>0.31</td>
<td>0.64</td>
<td>0.59</td>
<td>0.33</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
<td>0.63</td>
<td>0.98</td>
</tr>
<tr>
<td>160</td>
<td>0.32</td>
<td>0.64</td>
<td>0.6</td>
<td>0.34</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
<td>0.64</td>
<td>0.98</td>
</tr>
<tr>
<td>170</td>
<td>0.32</td>
<td>0.65</td>
<td>0.61</td>
<td>0.35</td>
<td>0.98</td>
<td>0.87</td>
<td>0.77</td>
<td>0.64</td>
<td>0.98</td>
</tr>
<tr>
<td>180</td>
<td>0.32</td>
<td>0.65</td>
<td>0.61</td>
<td>0.35</td>
<td>0.98</td>
<td>0.88</td>
<td>0.77</td>
<td>0.64</td>
<td>0.98</td>
</tr>
<tr>
<td>190</td>
<td>0.33</td>
<td>0.65</td>
<td>0.61</td>
<td>0.35</td>
<td>0.98</td>
<td>0.88</td>
<td>0.77</td>
<td>0.64</td>
<td>0.98</td>
</tr>
<tr>
<td>200</td>
<td>0.33</td>
<td>0.66</td>
<td>0.62</td>
<td>0.36</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.65</td>
<td>0.98</td>
</tr>
<tr>
<td>210</td>
<td>0.34</td>
<td>0.66</td>
<td>0.62</td>
<td>0.36</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.65</td>
<td>0.98</td>
</tr>
<tr>
<td>220</td>
<td>0.33</td>
<td>0.65</td>
<td>0.61</td>
<td>0.36</td>
<td>0.97</td>
<td>0.88</td>
<td>0.78</td>
<td>0.64</td>
<td>0.97</td>
</tr>
<tr>
<td>230</td>
<td>0.34</td>
<td>0.66</td>
<td>0.62</td>
<td>0.36</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.65</td>
<td>0.98</td>
</tr>
<tr>
<td>240</td>
<td>0.35</td>
<td>0.67</td>
<td>0.63</td>
<td>0.37</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>250</td>
<td>0.35</td>
<td>0.67</td>
<td>0.63</td>
<td>0.37</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>260</td>
<td>0.36</td>
<td>0.68</td>
<td>0.63</td>
<td>0.38</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>270</td>
<td>0.36</td>
<td>0.68</td>
<td>0.63</td>
<td>0.38</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>280</td>
<td>0.36</td>
<td>0.68</td>
<td>0.64</td>
<td>0.39</td>
<td>0.98</td>
<td>0.87</td>
<td>0.78</td>
<td>0.67</td>
<td>0.98</td>
</tr>
<tr>
<td>290</td>
<td>0.36</td>
<td>0.69</td>
<td>0.64</td>
<td>0.39</td>
<td>0.98</td>
<td>0.87</td>
<td>0.78</td>
<td>0.67</td>
<td>0.98</td>
</tr>
<tr>
<td>300</td>
<td>0.36</td>
<td>0.69</td>
<td>0.65</td>
<td>0.39</td>
<td>0.98</td>
<td>0.87</td>
<td>0.78</td>
<td>0.67</td>
<td>0.98</td>
</tr>
</tbody>
</table>
### E.1.2. Logarithmic Interval

<table>
<thead>
<tr>
<th>Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>0.65</td>
<td>1.17</td>
<td>1.04</td>
<td>0.75</td>
<td>1.09</td>
<td>0.91</td>
<td>0.71</td>
<td>1.35</td>
<td>1.09</td>
</tr>
<tr>
<td>64</td>
<td>0.94</td>
<td>1.45</td>
<td>1.34</td>
<td>1.04</td>
<td>1.19</td>
<td>0.98</td>
<td>0.79</td>
<td>1.58</td>
<td>1.19</td>
</tr>
<tr>
<td>128</td>
<td>1.32</td>
<td>1.76</td>
<td>1.65</td>
<td>1.43</td>
<td>1.33</td>
<td>1.12</td>
<td>0.9</td>
<td>1.84</td>
<td>1.33</td>
</tr>
<tr>
<td>256</td>
<td>1.68</td>
<td>1.99</td>
<td>1.89</td>
<td>1.77</td>
<td>1.5</td>
<td>1.29</td>
<td>1.01</td>
<td>2.01</td>
<td>1.5</td>
</tr>
<tr>
<td>512</td>
<td>1.93</td>
<td>2.12</td>
<td>2.04</td>
<td>2.01</td>
<td>1.68</td>
<td>1.46</td>
<td>1.1</td>
<td>2.1</td>
<td>1.68</td>
</tr>
<tr>
<td>1024</td>
<td>2.09</td>
<td>2.18</td>
<td>2.11</td>
<td>2.15</td>
<td>1.85</td>
<td>1.58</td>
<td>1.19</td>
<td>2.15</td>
<td>1.85</td>
</tr>
<tr>
<td>2048</td>
<td>2.18</td>
<td>2.22</td>
<td>2.16</td>
<td>2.23</td>
<td>2</td>
<td>1.69</td>
<td>1.28</td>
<td>2.18</td>
<td>2</td>
</tr>
<tr>
<td>4096</td>
<td>2.22</td>
<td>2.24</td>
<td>2.18</td>
<td>2.27</td>
<td>2.11</td>
<td>1.8</td>
<td>1.38</td>
<td>2.19</td>
<td>2.11</td>
</tr>
<tr>
<td>8192</td>
<td>2.24</td>
<td>2.25</td>
<td>2.19</td>
<td>2.29</td>
<td>2.19</td>
<td>1.86</td>
<td>1.45</td>
<td>2.2</td>
<td>2.19</td>
</tr>
<tr>
<td>16384</td>
<td>2.25</td>
<td>2.25</td>
<td>2.19</td>
<td>2.3</td>
<td>2.24</td>
<td>1.92</td>
<td>1.49</td>
<td>2.2</td>
<td>2.24</td>
</tr>
<tr>
<td>32768</td>
<td>2.25</td>
<td>2.25</td>
<td>2.2</td>
<td>2.31</td>
<td>2.27</td>
<td>1.95</td>
<td>1.51</td>
<td>2.2</td>
<td>2.27</td>
</tr>
<tr>
<td>65536</td>
<td>2.26</td>
<td>2.25</td>
<td>2.2</td>
<td>2.31</td>
<td>2.28</td>
<td>1.98</td>
<td>1.52</td>
<td>2.2</td>
<td>2.28</td>
</tr>
<tr>
<td>131072</td>
<td>2.26</td>
<td>2.26</td>
<td>2.2</td>
<td>2.31</td>
<td>2.29</td>
<td>2.02</td>
<td>1.52</td>
<td>2.2</td>
<td>2.29</td>
</tr>
<tr>
<td>262144</td>
<td>2.26</td>
<td>2.26</td>
<td>2.2</td>
<td>2.31</td>
<td>2.29</td>
<td>2.05</td>
<td>1.53</td>
<td>2.2</td>
<td>2.29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.32</td>
<td>0.69</td>
<td>0.62</td>
<td>0.35</td>
<td>0.98</td>
<td>0.82</td>
<td>0.68</td>
<td>0.74</td>
<td>0.98</td>
</tr>
<tr>
<td>128</td>
<td>0.37</td>
<td>0.76</td>
<td>0.7</td>
<td>0.41</td>
<td>1.01</td>
<td>0.86</td>
<td>0.73</td>
<td>0.79</td>
<td>1.01</td>
</tr>
<tr>
<td>256</td>
<td>0.47</td>
<td>0.86</td>
<td>0.8</td>
<td>0.51</td>
<td>1.04</td>
<td>0.88</td>
<td>0.76</td>
<td>0.86</td>
<td>1.04</td>
</tr>
<tr>
<td>512</td>
<td>0.67</td>
<td>1.01</td>
<td>0.96</td>
<td>0.7</td>
<td>1.1</td>
<td>0.91</td>
<td>0.79</td>
<td>1.01</td>
<td>1.1</td>
</tr>
<tr>
<td>1024</td>
<td>0.88</td>
<td>1.13</td>
<td>1.1</td>
<td>0.92</td>
<td>1.15</td>
<td>0.89</td>
<td>0.82</td>
<td>1.13</td>
<td>1.15</td>
</tr>
<tr>
<td>2048</td>
<td>1.06</td>
<td>1.22</td>
<td>1.2</td>
<td>1.09</td>
<td>1.2</td>
<td>0.89</td>
<td>0.85</td>
<td>1.21</td>
<td>1.2</td>
</tr>
<tr>
<td>4096</td>
<td>1.18</td>
<td>1.27</td>
<td>1.26</td>
<td>1.2</td>
<td>1.24</td>
<td>0.91</td>
<td>0.89</td>
<td>1.26</td>
<td>1.24</td>
</tr>
<tr>
<td>8192</td>
<td>1.25</td>
<td>1.3</td>
<td>1.29</td>
<td>1.26</td>
<td>1.28</td>
<td>0.93</td>
<td>0.94</td>
<td>1.29</td>
<td>1.28</td>
</tr>
<tr>
<td>16384</td>
<td>1.29</td>
<td>1.32</td>
<td>1.31</td>
<td>1.3</td>
<td>1.3</td>
<td>0.99</td>
<td>0.99</td>
<td>1.31</td>
<td>1.3</td>
</tr>
<tr>
<td>32768</td>
<td>1.31</td>
<td>1.32</td>
<td>1.32</td>
<td>1.31</td>
<td>1.31</td>
<td>1.04</td>
<td>1.02</td>
<td>1.32</td>
<td>1.31</td>
</tr>
<tr>
<td>65536</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
<td>1.08</td>
<td>1.03</td>
<td>1.33</td>
<td>1.32</td>
</tr>
<tr>
<td>131072</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.09</td>
<td>1.04</td>
<td>1.33</td>
<td>1.33</td>
</tr>
<tr>
<td>262144</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.09</td>
<td>1.05</td>
<td>1.33</td>
<td>1.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chars</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>0.27</td>
<td>0.61</td>
<td>0.55</td>
<td>0.3</td>
<td>0.95</td>
<td>0.82</td>
<td>0.68</td>
<td>0.65</td>
<td>0.95</td>
</tr>
<tr>
<td>64</td>
<td>0.28</td>
<td>0.63</td>
<td>0.57</td>
<td>0.31</td>
<td>0.97</td>
<td>0.85</td>
<td>0.73</td>
<td>0.64</td>
<td>0.97</td>
</tr>
<tr>
<td>128</td>
<td>0.31</td>
<td>0.64</td>
<td>0.59</td>
<td>0.33</td>
<td>0.97</td>
<td>0.87</td>
<td>0.76</td>
<td>0.63</td>
<td>0.97</td>
</tr>
<tr>
<td>256</td>
<td>0.35</td>
<td>0.67</td>
<td>0.63</td>
<td>0.37</td>
<td>0.98</td>
<td>0.88</td>
<td>0.78</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>512</td>
<td>0.42</td>
<td>0.75</td>
<td>0.7</td>
<td>0.47</td>
<td>0.98</td>
<td>0.84</td>
<td>0.79</td>
<td>0.72</td>
<td>0.98</td>
</tr>
<tr>
<td>1024</td>
<td>0.56</td>
<td>0.83</td>
<td>0.79</td>
<td>0.6</td>
<td>0.98</td>
<td>0.8</td>
<td>0.8</td>
<td>0.81</td>
<td>0.98</td>
</tr>
<tr>
<td>2048</td>
<td>0.72</td>
<td>0.9</td>
<td>0.87</td>
<td>0.75</td>
<td>0.98</td>
<td>0.77</td>
<td>0.8</td>
<td>0.88</td>
<td>0.98</td>
</tr>
<tr>
<td>4096</td>
<td>0.84</td>
<td>0.94</td>
<td>0.93</td>
<td>0.86</td>
<td>0.99</td>
<td>0.73</td>
<td>0.81</td>
<td>0.93</td>
<td>0.99</td>
</tr>
<tr>
<td>8192</td>
<td>0.91</td>
<td>0.97</td>
<td>0.96</td>
<td>0.92</td>
<td>0.99</td>
<td>0.7</td>
<td>0.82</td>
<td>0.96</td>
<td>0.99</td>
</tr>
<tr>
<td>16384</td>
<td>0.95</td>
<td>0.98</td>
<td>0.98</td>
<td>0.96</td>
<td>1</td>
<td>0.68</td>
<td>0.82</td>
<td>0.98</td>
<td>1</td>
</tr>
<tr>
<td>32768</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
<td>0.98</td>
<td>1</td>
<td>0.68</td>
<td>0.83</td>
<td>0.99</td>
<td>1</td>
</tr>
<tr>
<td>65536</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>1</td>
<td>0.71</td>
<td>0.83</td>
<td>0.99</td>
<td>1</td>
</tr>
<tr>
<td>131072</td>
<td>0.99</td>
<td>1</td>
<td>1</td>
<td>0.99</td>
<td>1</td>
<td>0.75</td>
<td>0.83</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>262144</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.75</td>
<td>0.83</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

62
### Appendix E. Results from post-testing

#### E.1.3. Corpuses

<table>
<thead>
<tr>
<th>Artificial</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.txt</td>
<td>0.1</td>
<td>0.17</td>
<td>0.11</td>
<td>0.17</td>
<td>0.33</td>
<td>0.2</td>
<td>0.1</td>
<td>0.5</td>
<td>0.33</td>
</tr>
<tr>
<td>aaa.txt</td>
<td>0</td>
<td>8</td>
<td>7.99</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.46</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>alphabet.txt</td>
<td>1.68</td>
<td>1.59</td>
<td>1.66</td>
<td>1.7</td>
<td>1.69</td>
<td>0</td>
<td>8.44</td>
<td>2.63</td>
<td>1.69</td>
</tr>
<tr>
<td>random.txt</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.33</td>
<td>1.08</td>
<td>1.04</td>
<td>1.33</td>
<td>1.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calgary</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>1.52</td>
<td>1.53</td>
<td>1.53</td>
<td>1.53</td>
<td>1.53</td>
<td>2.39</td>
<td>2.16</td>
<td>1.53</td>
<td>1.53</td>
</tr>
<tr>
<td>book1</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.76</td>
<td>1.76</td>
<td>2.32</td>
<td>1.84</td>
<td>1.75</td>
<td>1.76</td>
</tr>
<tr>
<td>book2</td>
<td>1.66</td>
<td>1.66</td>
<td>1.66</td>
<td>1.67</td>
<td>1.67</td>
<td>2.44</td>
<td>2.16</td>
<td>1.66</td>
<td>1.67</td>
</tr>
<tr>
<td>geo</td>
<td>1.39</td>
<td>1.41</td>
<td>1.4</td>
<td>1.41</td>
<td>1.32</td>
<td>1.32</td>
<td>1.21</td>
<td>1.4</td>
<td>1.41</td>
</tr>
<tr>
<td>news</td>
<td>1.53</td>
<td>1.53</td>
<td>1.53</td>
<td>1.54</td>
<td>1.54</td>
<td>2.07</td>
<td>1.98</td>
<td>1.53</td>
<td>1.54</td>
</tr>
<tr>
<td>obj1</td>
<td>1.26</td>
<td>1.32</td>
<td>1.31</td>
<td>1.34</td>
<td>1.34</td>
<td>1.92</td>
<td>2.39</td>
<td>1.37</td>
<td>1.34</td>
</tr>
<tr>
<td>obj2</td>
<td>1.27</td>
<td>1.27</td>
<td>1.27</td>
<td>1.32</td>
<td>1.32</td>
<td>2.12</td>
<td>2.19</td>
<td>1.59</td>
<td>1.61</td>
</tr>
<tr>
<td>paper1</td>
<td>1.58</td>
<td>1.59</td>
<td>1.59</td>
<td>1.6</td>
<td>1.6</td>
<td>2.27</td>
<td>2.1</td>
<td>1.72</td>
<td>1.73</td>
</tr>
<tr>
<td>paper2</td>
<td>1.71</td>
<td>1.72</td>
<td>1.72</td>
<td>1.73</td>
<td>1.73</td>
<td>2.27</td>
<td>2.1</td>
<td>1.72</td>
<td>1.73</td>
</tr>
<tr>
<td>paper3</td>
<td>1.69</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>2.1</td>
<td>2.01</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>paper4</td>
<td>1.63</td>
<td>1.67</td>
<td>1.66</td>
<td>1.66</td>
<td>1.66</td>
<td>1.91</td>
<td>1.98</td>
<td>1.67</td>
<td>1.66</td>
</tr>
<tr>
<td>paper5</td>
<td>1.54</td>
<td>1.58</td>
<td>1.58</td>
<td>1.57</td>
<td>1.58</td>
<td>1.82</td>
<td>1.98</td>
<td>1.58</td>
<td>1.58</td>
</tr>
<tr>
<td>paper6</td>
<td>1.56</td>
<td>1.58</td>
<td>1.58</td>
<td>1.58</td>
<td>1.6</td>
<td>2.04</td>
<td>2.19</td>
<td>1.58</td>
<td>1.6</td>
</tr>
<tr>
<td>pic</td>
<td>4.79</td>
<td>4.81</td>
<td>4.81</td>
<td>6.55</td>
<td>6.86</td>
<td>8.25</td>
<td>4.91</td>
<td>4.81</td>
<td>6.86</td>
</tr>
<tr>
<td>proge</td>
<td>1.51</td>
<td>1.52</td>
<td>1.52</td>
<td>1.52</td>
<td>1.53</td>
<td>2.07</td>
<td>2.28</td>
<td>1.52</td>
<td>1.53</td>
</tr>
<tr>
<td>progl</td>
<td>1.65</td>
<td>1.66</td>
<td>1.66</td>
<td>1.67</td>
<td>1.68</td>
<td>2.64</td>
<td>3.21</td>
<td>1.66</td>
<td>1.68</td>
</tr>
<tr>
<td>progp</td>
<td>1.62</td>
<td>1.63</td>
<td>1.63</td>
<td>1.63</td>
<td>1.63</td>
<td>2.57</td>
<td>3.22</td>
<td>1.63</td>
<td>1.63</td>
</tr>
<tr>
<td>trans</td>
<td>1.43</td>
<td>1.43</td>
<td>1.43</td>
<td>1.44</td>
<td>1.46</td>
<td>2.45</td>
<td>2.83</td>
<td>1.43</td>
<td>1.46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Canterbury</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>1.73</td>
<td>1.73</td>
<td>1.73</td>
<td>1.75</td>
<td>1.74</td>
<td>2.44</td>
<td>2.11</td>
<td>1.73</td>
<td>1.74</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>1.65</td>
<td>1.65</td>
<td>1.65</td>
<td>1.66</td>
<td>1.65</td>
<td>2.28</td>
<td>1.93</td>
<td>1.65</td>
<td>1.65</td>
</tr>
<tr>
<td>cp.html</td>
<td>1.49</td>
<td>1.51</td>
<td>1.51</td>
<td>1.51</td>
<td>1.51</td>
<td>2.17</td>
<td>2.3</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>fields.c</td>
<td>1.52</td>
<td>1.56</td>
<td>1.56</td>
<td>1.54</td>
<td>1.56</td>
<td>2.25</td>
<td>2.91</td>
<td>1.56</td>
<td>1.56</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>1.52</td>
<td>1.65</td>
<td>1.64</td>
<td>1.57</td>
<td>1.62</td>
<td>2.05</td>
<td>2.41</td>
<td>1.65</td>
<td>1.62</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.23</td>
<td>2.37</td>
<td>3.32</td>
<td>3.15</td>
<td>2.22</td>
<td>2.38</td>
</tr>
<tr>
<td>lect10.xls</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.71</td>
<td>1.72</td>
<td>2.62</td>
<td>2.16</td>
<td>1.7</td>
<td>1.72</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.76</td>
<td>1.76</td>
<td>2.37</td>
<td>1.85</td>
<td>1.75</td>
<td>1.76</td>
</tr>
<tr>
<td>ptt5</td>
<td>4.79</td>
<td>4.81</td>
<td>4.81</td>
<td>6.55</td>
<td>6.86</td>
<td>8.25</td>
<td>4.91</td>
<td>4.81</td>
<td>6.86</td>
</tr>
<tr>
<td>sum</td>
<td>1.44</td>
<td>1.47</td>
<td>1.47</td>
<td>1.46</td>
<td>1.55</td>
<td>1.9</td>
<td>2.14</td>
<td>1.47</td>
<td>1.55</td>
</tr>
<tr>
<td>xargs.1</td>
<td>1.47</td>
<td>1.57</td>
<td>1.56</td>
<td>1.51</td>
<td>1.54</td>
<td>1.81</td>
<td>2.02</td>
<td>1.56</td>
<td>1.54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Large</th>
<th>Huffman</th>
<th>Lambda</th>
<th>FGK</th>
<th>Arithmetic</th>
<th>Ad-Arithmetic</th>
<th>LZW</th>
<th>LZ77</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.coli</td>
<td>4</td>
<td>3.56</td>
<td>3.56</td>
<td>4</td>
<td>3.94</td>
<td>3.69</td>
<td>2.88</td>
<td>3.56</td>
<td>3.94</td>
</tr>
<tr>
<td>bible.txt</td>
<td>1.82</td>
<td>1.82</td>
<td>1.82</td>
<td>1.84</td>
<td>1.84</td>
<td>2.89</td>
<td>2.5</td>
<td>1.82</td>
<td>1.84</td>
</tr>
<tr>
<td>world192.txt</td>
<td>1.59</td>
<td>1.59</td>
<td>1.59</td>
<td>1.6</td>
<td>1.6</td>
<td>2.51</td>
<td>1.93</td>
<td>1.59</td>
<td>1.6</td>
</tr>
</tbody>
</table>
## Appendix E. Results from post-testing

### E.2. Time Tests

#### E.2.1. Small Interval

<table>
<thead>
<tr>
<th>11 Chars</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1793400</td>
<td>350800</td>
</tr>
<tr>
<td>110</td>
<td>1069000</td>
<td>369400</td>
</tr>
<tr>
<td>120</td>
<td>1173200</td>
<td>392200</td>
</tr>
<tr>
<td>130</td>
<td>1230800</td>
<td>414300</td>
</tr>
<tr>
<td>140</td>
<td>1304000</td>
<td>449900</td>
</tr>
<tr>
<td>150</td>
<td>1392900</td>
<td>467900</td>
</tr>
<tr>
<td>160</td>
<td>1469600</td>
<td>488700</td>
</tr>
<tr>
<td>170</td>
<td>1552000</td>
<td>508100</td>
</tr>
<tr>
<td>180</td>
<td>1648300</td>
<td>527900</td>
</tr>
<tr>
<td>190</td>
<td>1726700</td>
<td>551300</td>
</tr>
<tr>
<td>200</td>
<td>1803700</td>
<td>576100</td>
</tr>
<tr>
<td>210</td>
<td>1854100</td>
<td>605600</td>
</tr>
<tr>
<td>220</td>
<td>1949300</td>
<td>621200</td>
</tr>
<tr>
<td>230</td>
<td>3584800</td>
<td>677200</td>
</tr>
<tr>
<td>240</td>
<td>2096100</td>
<td>673300</td>
</tr>
<tr>
<td>250</td>
<td>2439900</td>
<td>693300</td>
</tr>
<tr>
<td>260</td>
<td>2368600</td>
<td>719900</td>
</tr>
<tr>
<td>270</td>
<td>2369900</td>
<td>735400</td>
</tr>
<tr>
<td>280</td>
<td>2953500</td>
<td>764700</td>
</tr>
<tr>
<td>290</td>
<td>2516200</td>
<td>777300</td>
</tr>
<tr>
<td>300</td>
<td>3898400</td>
<td>801200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64 Chars</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2041000</td>
<td>468400</td>
</tr>
<tr>
<td>110</td>
<td>2203800</td>
<td>494200</td>
</tr>
<tr>
<td>120</td>
<td>4239400</td>
<td>531000</td>
</tr>
<tr>
<td>130</td>
<td>3387800</td>
<td>573200</td>
</tr>
<tr>
<td>140</td>
<td>2931700</td>
<td>605500</td>
</tr>
<tr>
<td>150</td>
<td>2983100</td>
<td>655700</td>
</tr>
<tr>
<td>160</td>
<td>4645600</td>
<td>682900</td>
</tr>
<tr>
<td>170</td>
<td>3376200</td>
<td>723000</td>
</tr>
<tr>
<td>180</td>
<td>6233100</td>
<td>767600</td>
</tr>
<tr>
<td>190</td>
<td>3868900</td>
<td>837200</td>
</tr>
<tr>
<td>200</td>
<td>4284400</td>
<td>1328700</td>
</tr>
<tr>
<td>210</td>
<td>4178700</td>
<td>874700</td>
</tr>
<tr>
<td>220</td>
<td>4365000</td>
<td>921900</td>
</tr>
<tr>
<td>230</td>
<td>6217100</td>
<td>947500</td>
</tr>
<tr>
<td>240</td>
<td>4796400</td>
<td>986700</td>
</tr>
<tr>
<td>250</td>
<td>4952900</td>
<td>1021900</td>
</tr>
<tr>
<td>260</td>
<td>5114100</td>
<td>1062000</td>
</tr>
<tr>
<td>270</td>
<td>5243200</td>
<td>1095500</td>
</tr>
<tr>
<td>280</td>
<td>5446700</td>
<td>1138800</td>
</tr>
<tr>
<td>290</td>
<td>6528700</td>
<td>1168100</td>
</tr>
<tr>
<td>300</td>
<td>5673400</td>
<td>1267900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>256 Chars</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3048700</td>
<td>1098600</td>
</tr>
<tr>
<td>110</td>
<td>3164100</td>
<td>954800</td>
</tr>
<tr>
<td>120</td>
<td>3569100</td>
<td>1386400</td>
</tr>
<tr>
<td>130</td>
<td>4296600</td>
<td>1135900</td>
</tr>
<tr>
<td>140</td>
<td>4429300</td>
<td>1245800</td>
</tr>
<tr>
<td>150</td>
<td>5419500</td>
<td>2203000</td>
</tr>
<tr>
<td>160</td>
<td>6059300</td>
<td>1215500</td>
</tr>
<tr>
<td>170</td>
<td>6883900</td>
<td>1947800</td>
</tr>
<tr>
<td>180</td>
<td>7975700</td>
<td>1428100</td>
</tr>
<tr>
<td>190</td>
<td>7018000</td>
<td>1416800</td>
</tr>
<tr>
<td>200</td>
<td>8528400</td>
<td>1506400</td>
</tr>
<tr>
<td>210</td>
<td>9716100</td>
<td>2223200</td>
</tr>
<tr>
<td>220</td>
<td>15176800</td>
<td>2254300</td>
</tr>
<tr>
<td>230</td>
<td>9361400</td>
<td>1827500</td>
</tr>
<tr>
<td>240</td>
<td>13367700</td>
<td>3567200</td>
</tr>
<tr>
<td>250</td>
<td>17165200</td>
<td>1887600</td>
</tr>
<tr>
<td>260</td>
<td>15833700</td>
<td>3522100</td>
</tr>
<tr>
<td>270</td>
<td>11460400</td>
<td>2098800</td>
</tr>
<tr>
<td>280</td>
<td>12161400</td>
<td>3333400</td>
</tr>
<tr>
<td>290</td>
<td>14115600</td>
<td>3019200</td>
</tr>
<tr>
<td>300</td>
<td>13467200</td>
<td>2273600</td>
</tr>
</tbody>
</table>
### E.2.2. Logarithmic Interval

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>418500</td>
<td>185000</td>
<td>36</td>
<td>640400</td>
<td>205900</td>
</tr>
<tr>
<td>64</td>
<td>681500</td>
<td>257700</td>
<td>64</td>
<td>3269600</td>
<td>313900</td>
</tr>
<tr>
<td>128</td>
<td>1219800</td>
<td>415400</td>
<td>128</td>
<td>3077900</td>
<td>567300</td>
</tr>
<tr>
<td>256</td>
<td>2317100</td>
<td>704200</td>
<td>256</td>
<td>5032700</td>
<td>1042200</td>
</tr>
<tr>
<td>512</td>
<td>4332100</td>
<td>1278000</td>
<td>512</td>
<td>9271600</td>
<td>2038900</td>
</tr>
<tr>
<td>1024</td>
<td>11396800</td>
<td>2389700</td>
<td>1024</td>
<td>33551600</td>
<td>5601600</td>
</tr>
<tr>
<td>2048</td>
<td>22091000</td>
<td>9410700</td>
<td>2048</td>
<td>70664900</td>
<td>21228600</td>
</tr>
<tr>
<td>4096</td>
<td>77516800</td>
<td>23836300</td>
<td>4096</td>
<td>74611800</td>
<td>25497500</td>
</tr>
<tr>
<td>8192</td>
<td>92079100</td>
<td>34948900</td>
<td>8192</td>
<td>81374000</td>
<td>36886300</td>
</tr>
<tr>
<td>16384</td>
<td>105635700</td>
<td>41739000</td>
<td>16384</td>
<td>118703800</td>
<td>45626500</td>
</tr>
<tr>
<td>32768</td>
<td>123130100</td>
<td>47391700</td>
<td>32768</td>
<td>128028400</td>
<td>47504200</td>
</tr>
<tr>
<td>65536</td>
<td>134664000</td>
<td>55004900</td>
<td>65536</td>
<td>140341000</td>
<td>56583200</td>
</tr>
<tr>
<td>131072</td>
<td>161416300</td>
<td>66534800</td>
<td>131072</td>
<td>205851200</td>
<td>89507700</td>
</tr>
<tr>
<td>262144</td>
<td>231523400</td>
<td>10338600</td>
<td>262144</td>
<td>315402000</td>
<td>177841800</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>727000</td>
<td>378500</td>
<td>36</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>64</td>
<td>1532800</td>
<td>518100</td>
<td>64</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>128</td>
<td>6721200</td>
<td>1549000</td>
<td>128</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>256</td>
<td>11002200</td>
<td>1980700</td>
<td>256</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>512</td>
<td>26824100</td>
<td>4802300</td>
<td>512</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>1024</td>
<td>58542600</td>
<td>8476900</td>
<td>1024</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>2048</td>
<td>71344500</td>
<td>21355200</td>
<td>2048</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>4096</td>
<td>78644600</td>
<td>26971200</td>
<td>4096</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>8192</td>
<td>97622100</td>
<td>29310300</td>
<td>8192</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>16384</td>
<td>106844900</td>
<td>39459100</td>
<td>16384</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>32768</td>
<td>11502000</td>
<td>45588300</td>
<td>32768</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>65536</td>
<td>157359300</td>
<td>67974400</td>
<td>65536</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>131072</td>
<td>226669200</td>
<td>116834100</td>
<td>131072</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>262144</td>
<td>4685660700</td>
<td>268667000</td>
<td>262144</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
</tbody>
</table>

### E.2.3. Corpuses

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.txt</td>
<td>70000</td>
<td>74000</td>
<td>E.coli</td>
<td>461504000</td>
<td>3253898000</td>
</tr>
<tr>
<td>aaa.txt</td>
<td>106263000</td>
<td>27762000</td>
<td>bible.txt</td>
<td>14325115000</td>
<td>130257750000</td>
</tr>
<tr>
<td>alphabet.txt</td>
<td>140915000</td>
<td>643130000</td>
<td>world192.txt</td>
<td>6667441000</td>
<td>5559496000</td>
</tr>
<tr>
<td>random.txt</td>
<td>168861000</td>
<td>74408000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Appendix E. Results from post-testing

<table>
<thead>
<tr>
<th>Calgary</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>170241000</td>
<td>73577000</td>
</tr>
<tr>
<td>book1</td>
<td>792872000</td>
<td>522066000</td>
</tr>
<tr>
<td>book2</td>
<td>634593000</td>
<td>393641000</td>
</tr>
<tr>
<td>geo</td>
<td>171482000</td>
<td>74940000</td>
</tr>
<tr>
<td>news</td>
<td>382218000</td>
<td>224042000</td>
</tr>
<tr>
<td>obj1</td>
<td>116303000</td>
<td>41854000</td>
</tr>
<tr>
<td>obj2</td>
<td>315697000</td>
<td>161521000</td>
</tr>
<tr>
<td>paper1</td>
<td>129560000</td>
<td>50432000</td>
</tr>
<tr>
<td>paper2</td>
<td>145178000</td>
<td>58635000</td>
</tr>
<tr>
<td>paper3</td>
<td>127456000</td>
<td>49667000</td>
</tr>
<tr>
<td>paper4</td>
<td>113931000</td>
<td>38663000</td>
</tr>
<tr>
<td>paper5</td>
<td>103284000</td>
<td>38559000</td>
</tr>
<tr>
<td>paper6</td>
<td>124618000</td>
<td>48398000</td>
</tr>
<tr>
<td>pic</td>
<td>257336000</td>
<td>73896000</td>
</tr>
<tr>
<td>progc</td>
<td>124501000</td>
<td>53973000</td>
</tr>
<tr>
<td>progl</td>
<td>136615000</td>
<td>55232000</td>
</tr>
<tr>
<td>progp</td>
<td>125119000</td>
<td>50717000</td>
</tr>
<tr>
<td>trans</td>
<td>163032000</td>
<td>69047000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Canterbury</th>
<th>Lambda [M]</th>
<th>Arithmetic [M]</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>191030000</td>
<td>81238000</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>170545000</td>
<td>75187000</td>
</tr>
<tr>
<td>cp.html</td>
<td>999510000</td>
<td>44692000</td>
</tr>
<tr>
<td>fields.c</td>
<td>102517000</td>
<td>39044000</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>74139000</td>
<td>23927000</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>870092000</td>
<td>532082000</td>
</tr>
<tr>
<td>lct10.xls</td>
<td>410801000</td>
<td>225129000</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>429554000</td>
<td>258853000</td>
</tr>
<tr>
<td>ptt5</td>
<td>206428000</td>
<td>74198000</td>
</tr>
<tr>
<td>sum</td>
<td>130899000</td>
<td>45070000</td>
</tr>
<tr>
<td>xargs.1</td>
<td>825090000</td>
<td>25275000</td>
</tr>
</tbody>
</table>