Dissertation Type: Research

DEPARTMENT OF COMPUTER SCIENCE

Dictionary Matching with Fingerprints
An Empirical Analysis

Dominic Joseph Moylett

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 17th April, 2015

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Dominic Joseph Moylett, Friday 17th April, 2015

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Executive Summary

A compulsory section, of at most 1 page

This section should précis the project context, aims and objectives, and main contributions and achievements; the same section may be called an abstract elsewhere. The goal is to ensure the reader is clear about what the topic is, what you have done within this topic, *and* what your view of the outcome is.

The former aspects should be guided by your specification: essentially this section is a (very) short version of what is typically the first chapter. The latter aspects should be presented as a concise, factual bullet point list. The points will of course differ for each project, but an example is as follows:

- I spent 120 hours collecting material on and learning about the Java garbage-collection sub-system.
- I wrote a total of 5000 lines of source code, comprising a Linux device driver for a robot (in C) and a GUI (in Java) that is used to control it.
- I designed a new algorithm for computing the non-linear mapping from A-space to B-space using a genetic algorithm, see page 17.
- I implemented a version of the algorithm proposed by Jones and Smith in [6], see page 12, corrected a mistake in it, and compared the results with several alternatives.

Supporting Technologies

- I used the GNU Multiple Precision Arithmetic Library (GMP) to support my implementation of Karp-Rabin fingerprints.
- I used the C Minimum Perfect Hashing Library (CMPH) for static perfect hashing.
- I used an open-source implementation of Red-Black Trees from http://en.literateprograms.org/Red-black_tree_(C)?oldid=19567, with some minor adaptations.

Notation and Acronyms

CMPH	:	C Minimum Perfect Hashing Library
GMP	:	GNU Multiple Precision Arithmetic Library
VO	:	A Viable Occurrence, a portion of the text which might match a pattern
KMP	:	The Knuth-Morris-Pratt single pattern matching algorithm
BST	:	Binary Search Tree
RBT	:	Red-Black Tree, a specific instance of a binary search tree
T	:	A text string of n characters
t_i	:	The i-th character in T
$\mathcal{P}$	:	A list of k patterns
P_i	:	The i-th pattern in $\mathcal{P}$, a text string of m_i characters
M	:	A list of the length of each pattern in $\mathcal{P}$.
$p_{i,j}$	:	The j-th character in P_i
$\|S\|$	:	The length of a string S
$\phi(S)$	:	The Karp-Rabin fingerprint of a string S
ρ_S	:	The period of a string S

Acknowledgements

First and foremost, I would like to thank my supervisors: Dr. Raphaël Clifford and Dr. Benjamin Sach. This project would have been impossible without their work and advice. Alongside them, I would like to mention Dr. Markus Jalsenius for his assistance during the summer project that led to this work and Dr. Allyx Fontaine, who contributed to the paper on which my project is based and advised me alongside Benjamin every week.

Everyone on my course has had an impact on me over the past four years. In particular, I would like to mention William Coaluca, Stephen de Mora, Nicholas Phillips, James Savage and Ashley Whetter. I have put countless hours into many projects with one or more of them.

I would like to acknowledge David Beddows, Derek Bekoe, Timothy Lewis and Jonathan Walsh for remaining a stable household for the past three years - four in the case of David and Timothy.

Last, but most certainly not least, I would like to thank my family and friends for the infinite support, happiness and love they have given me my entire life.

Chapter 1

Contextual Background

A compulsory chapter, of roughly 10 **pages**

This chapter should describe the project context, and motivate each of the proposed aims and objectives. Ideally, it is written at a fairly high-level, and easily understood by a reader who is technically competent but not an expert in the topic itself.

In short, the goal is to answer three questions for the reader. First, what is the project topic, or problem being investigated? Second, why is the topic important, or rather why should the reader care about it? For example, why there is a need for this project (e.g., lack of similar software or deficiency in existing software), who will benefit from the project and in what way (e.g., end-users, or software developers) what work does the project build on and why is the selected approach either important and/or interesting (e.g., fills a gap in literature, applies results from another field to a new problem). Finally, what are the central challenges involved and why are they significant?

The chapter should conclude with a concise bullet point list that summarises the aims and objectives. For example:

> The high-level objective of this project is to reduce the performance gap between hardware and software implementations of modular arithmetic. More specifically, the concrete aims are:
>
> 1. Research and survey literature on public-key cryptography and identify the state of the art in exponentiation algorithms.
> 2. Improve the state of the art algorithm so that it can be used in an effective and flexible way on constrained devices.
> 3. Implement a framework for describing exponentiation algorithms and populate it with suitable examples from the literature on an ARM7 platform.
> 4. Use the framework to perform a study of algorithm performance in terms of time and space, and show the proposed improvements are worthwhile.

Chapter 2

Technical Background

2.1 Pattern Matching: Formal Definitions

Pattern matching with a single pattern is a simple problem to describe intuitively: We have a text and a pattern, and we want to output any indexes where the pattern occurs in the text.

More formally, we refer to the text by T, and define it as a string of n characters $t_0...t_{n-1}$. Likewise, the pattern is referred to as P, and is a string of m characters $p_0...p_{m-1}$. The aim of the text indexing problem is to output indexes $i \in \{m-1, ..., n-1\}$ such that $t_{i-m+1}...t_i = P$.

It is worth noting that there are many other ways of defining this problem. The most notable differences in this paper are that the text and pattern are indexed at zero instead of one, and that the index at the end of the pattern's occurrence is returned instead of the index at the start. Both of these are done to be intentionally to be consistent with the code implemented: The zero-indexing is because the implementations are written in C, which also uses zero indexing, and reporting the index at the end of the occurrence is to cater for a limitation on the algorithm by Clifford et al. detailed in Section 2.8.2.

2.1.1 Dictionary Matching: Formal Definitions

Like pattern matching, dictionary matching is also simple to describe intuitively: We have one text as before, but now we have multiple patterns, and we want to output any indexes where a pattern occurs in the text.

Formally, this is defined as follows: We have a text n characters long $T = t_0...t_{n-1}$, and a set of k patterns $\mathcal{P} = \{P_0, ..., P_k\}$ of respective lengths $M = \{m_0, ..., m_k\}$. Hence a given pattern P_i is a string of m_i characters $p_{i,0}...pi, m_i - 1$. We output an index $j \in \{\min(M), ..., n_1\}$ if $\exists i \in \{0, ..., k-1\}$ such that $t_{j-m_i+1}...t_j = P_i$.

Note that for this work, we do not care about what patterns have occured in the text, only that a pattern has occured. This is due to a limitation with the algorithm by Clifford et al., which will be discussed in Section 2.8.2.

```
rotate buf by one
append t_i to buf for i = 0 upto m − 1 do
    if buf_i ≠ p_i then
        return -1
    end
end
return j
```

Algorithm 2.1: A naïve solution to single pattern matching.

2.2 The Streaming Model

Data streaming is a way of reducing space consumption for certain problems. Under this model, required space is reduced by not processing the entire problem input at once. Instead, the input is provided to the algorithm in portions, delivered via a stream of data. The algorithm processes one portion of the input at a time, and it is required that the algorithm is not allowed to store the entire input.

Under this model, we measure performance by two properties:

- **Space:** The size of the data structure
- **Time:** The time taken to process each portion in the stream

It is easy to see how pattern matching and in turn dictionary matching can be performed in this model. We can process the text by individual characters. During preprocessing we store the pattern and initialise a circular buffer buf which is m characters long. At index j when we receive character t_j we perform the algorithm described in Algorithm 2.1. A dictionary matching variant can be done by storing a circular buffer which is $\max(M)$ characters long and repeating Algorithm 2.1 k times. These algorithms use $O(m)$ and $O(\sum_{i=0}^{k-1} m_i)$ respectively, both in terms of space and time per character.

Of course, these are poor solutions to both pattern and dictionary matching. We can do much better in terms of both time and space complexity.

2.3 The Aho-Corasick Algorithm for Dictionary Matching

The Aho-Corasick Algorithm for Efficient String Matching[1] – known hereafter as Aho-Corasick – is a deterministic algorithm for dictionary matching. Published in 1975, the algorithm works as a generalisation of Knuth-Morris-Pratt (KMP)[12], extending the state machine from single patterns in KMP to multiple patterns.

Preprocessing consists of three algorithms. The first, Algorithm 2.2, produces the goto function, which determines what to do if the next character in the stream is a match. This in essence works by building a suffix tree: We traverse the tree until we either reach the end of the pattern or we hit a leaf, and then append the rest of the pattern to the leaf. Note that Σ refers to the alphabet of the patterns and $fail$ is a default fail state for if the goto function cannot find a character for that state.

The second, Algorithm 2.3 constructs the failure function for when the next character cannot be found and the output function for whether or not there is a match. This is similar to how the failure table is computed in Knuth-Morris-Pratt, by using previously computed failure tables to find the longest prefix that is also a suffix of that point in the pattern.

From these two algorithms alone it is possible to perform dictionary matching, using a computation method again similar to Knuth-Morris-Pratt: For each character t_j in the text when we are in state s,

$newstate \leftarrow -1$
for $i = 0$ **upto** $k - 1$ **do**
  $state \leftarrow -1$
  $j \leftarrow 0$
  while $goto(state, p_{i,j}) \neq fail$ **do**
    $state \leftarrow \text{goto}(state, p_{i,j})$
    $j \leftarrow j + 1$
  end
  while $j < m_i$ **do**
    $newstate \leftarrow newstate + 1$
    $\text{goto}(state, p_{i,j}) \leftarrow newstate$
    $state \leftarrow newstate$
    $j \leftarrow j + 1$
  end
  $\text{output}(state) = \{P_i\}$
end
forall the $a \in \Sigma$ *such that* $goto(-1, a) = fail$ **do**
  $\text{goto}(-1, a) = -1$
end

Algorithm 2.2: Constructing the goto function for Aho-Corasick.

we check if $\text{goto}(s, t_j) = fail$. If that is the case, we call $s \leftarrow \text{failure}(s)$ repeatedly until the previous check no longer holds. We then update our state $s \leftarrow \text{goto}(s, t_j)$, and if $\text{output}(s) \neq empty$ then we return j, otherwise we return -1. This runs in amortised $O(|\Sigma|)$ time per character, and worst case $O(|\Sigma| \max(M))$ time per character, as can be seen via Knuth-Morris-Pratt arguments.

To improve on this running time, Algorithm 2.4 is used, which combines the goto and failure functions to produce a next function, which given any state and character returns the next state. Computation now simply becomes as each character t_j comes in when we are in state s, call $s \leftarrow \text{next}(s, t_j)$ and return j if $\text{output}(s) \neq empty$. This runs in worst case $O(|\Sigma|)$ time per character, where the bottleneck is finding the value associated with character t_j in the next function. In both the case with the goto and failure functions and the case with only the next function, space complexity is $O(\sum_{i=0}^{k-1} m_i)$.

2.3.1 An Alternative: The Commentz-Walter Algorithm

Much like how Aho-Corasick is an algorithm for dictionary matching based on Knuth-Morris-Pratt, Commentz-Walter[8] is an algorithm based on Boyer-Moore algorithm[4], using similar techniques to Aho-Corasick to convert the algorithm from single pattern to multiple patterns. While it is interesting to note as an alternative, particularly because of its time improvement on average cases and the fact that a variant of it is used in the GNU command `grep`,[1] it is not implemented in this project. This is because, like Boyer-Moore, the Commentz-Walter algorithm skips indexes in the text, which is not possible in the streaming model.

2.4 Minimal Perfect Hashing

For a universe U, a hash function h is static if it can perform lookups for a pre-defined set of keys $S \subseteq U$ to a set of integers $\mathbb{Z}_m$. Said hash function is a static *perfect* hash function if $\forall x \in S, \text{h}(x)$ is collision-free, and thus takes constant time to look up. Finally, a hash function is a *minimal* perfect hash function if $m = |S|$. In other words, a minimal perfect hash function maps a set of m keys to $\mathbb{Z}_m$ without any

[1] See http://git.savannah.gnu.org/cgit/grep.git/tree/src/kwset.c

```
queue ← empty
foreach a ∈ Σ such that goto(−1, a) = s ≠ −1 do
    queue ← queue ∪ {s}
    failure(s) ← −1
end
while queue ≠ empty do
    r ← pop(queue)
    foreach a ∈ Σ such that goto(r, a) = s ≠ fail do
        queue ← queue ∪ s
        state ← failure(r)
        while goto(state, a) = fail do
            state ← failure(state)
        end
        failure(s) ← goto(state, a)
        output(s) ← output(s) ∪ output(failure(s))
    end
end
```

Algorithm 2.3: Constructing the failure and output functions for Aho-Corasick.

```
queue ← empty
foreach a ∈ Σ do
    next(−1, a) = goto(−1, a)
    if goto(−1, a) ≠ −1 then
        queue ← queue ∪ {goto(−1, a)}
    end
end
while queue ≠ empty do
    r ← pop(queue)
    foreach a ∈ Σ do
        if goto(r, a) = s ≠ fail then
            queue ← queue ∪ s
            next(r, a) = s
        else
            next(r, a) = next(failure(r), a)
        end
    end
end
```

Algorithm 2.4: Constructing the next function for Aho-Corasick.

collisions.

The implementation of minimal perfect hash functions will not be detailed here, as they are used merely as a library and are thus not part of implementation. For further information, I direct the reader to the C Minimal Perfect Hashing Library (CMPH) website: http://cmph.sourceforge.net/ Of particular interest is the paper on the Compress, Hash and Digest algorithm by Belazzougui, Botelho and Dietzfelbinger[3], as the algorithm from CMPH used throughout this work.

2.5 Karp-Rabin Fingerprints

Karp-Rabin fingerprints[11] are a function $\phi : \Sigma* \to \mathbb{Z}_p$ for some prime number p. For a text T of length n characters, the Karp-Rabin fingerprint is defined as:

$$\phi(T) = \sum_{i=0}^{n-1} r^i t_i \mod p$$

Where p is a prime number, and r is a random number such that $1 < r < p$. Alongside the fingerprint $\phi(T)$, we store $r^n \mod p$ and $r^{-n} \mod p$ in a tuple. Using these three properties, we can manipulate the fingerprints to affect the underlying strings in three ways[13]. Note that all equations listed below are modulo p.

- **Concatenate:** If we have a fingerprint $\{\phi(u), r^{n_1}, r^{-n_1}\}$ for a string u of length n_1 and another fingerprint $\{\phi(v), r^{n_2}, r^{-n_2}\}$ for a string v of length n_2, the concatenation of these two strings is $\{\phi(u) + \phi(v) * r^{n_1}, r^{n_1} * r^{n_2}, r^{-n_1} * r^{-n_2}\}$
- **Prefix:** If we have a fingerprint $\{\phi(uv), r^{n_1}, r^{-n_1}\}$ for a string uv of length n_1 and a fingerprint $\{\phi(v), r^{n_2}, r^{-n_2}\}$ for the n_2 suffix of uv, then we can work out the fingerprint of the $n_1 - n_2$ prefix of uv as $\{\phi(uv) - \phi(v) * r^{n_1}, r^{n_1} * r^{-n_2}, r^{-n_1} * r^{n_2}\}$
- **Suffix:** If we have a fingerprint $\{\phi(uv), r^{n_1}, r^{-n_1}\}$ for a string uv of length n_1 and a fingerprint $\{\phi(u), r^{n_2}, r^{-n_2}\}$ for the n_2 prefix of uv, then we can work out the fingerprint of the $n_1 - n_2$ suffix of uv as $\{(\phi(uv) - \phi(u)) * r^{-n_2}, r^{n_1} * r^{-n_2}, r^{-n_1} * r^{n_2}\}$

All of these operations can be completed in constant time.

It is interesting to note that a variant of the Karp-Rabin algorithm can be used for a subset of dictionary matching, where all the patterns are the same length m[6]. This can be done by storing a fingerprint of the last m characters read from the text, and using static perfect hashing as described in section 2.4 to check if the fingerprint of the text matches any fingerprints of the patterns. Using suffix and concatenation tectniques above and storing a circular buffer of the last m characters, we can accomplish this with $O(k + m)$ space and $O(1)$ time per character. However, due to the limitation that all the patterns have to be the same length, this method has not been implemented for this project.

The last point to mention is the probability of a collision. Breslauer and Galil[5] provide a theorem that if u and v are two different strings of length $l \leq n$, $p \in \theta(n^{2+\alpha})$ for some level of accuracy $\alpha \geq 0$ and $r \in \mathbb{Z}_p$ is randomly chosen, then the probability that $\phi(u) = \phi(v)$ is smaller than $\frac{1}{n^{1+\alpha}}$. We will however see later why this does not necessarily hold for the dictionary matching algorithm devised by Clifford et al. [7].

2.6 Porat and Porat: Single Pattern Matching in Sublinear Space

In 2009, Porat and Porat[13] provided the first solution to a pattern matching problem in sublinear space to the size of the pattern. Utilising Karp-Rabin fingerprints as described in the previous section, their randomised algorithm for single pattern matching in the streaming model had $O(\log m)$ complexity both in terms of space and time per character.

Detailed below is not Porat and Porat's algorithm itself, but a variant of it developed by Breslauer and Galil in 2014[5]. The two algorithms can be seen as computationally equivalent.

Instead of storing the entire pattern in a single fingerprint, the pattern is broken up into $\lfloor \log_2 m \rfloor$ fingerprints, each a power of two prefix of the patter. These fingerprints denoted ϕ_i, are computed as follows:

Level number	1	2	3
VO locations stored	5	3,4	-1,0,1,2

Figure 2.1: Example state of VO lists after 7 characters, where $T = aaaaaaa$ and $P = aaaaaaaa$

$$\phi_i = \phi(p_0...p_{2^i-1})$$

If the pattern is not a power of two in length, the remaining characters can be stored either in the fingerprint of the final prefix $\phi_{\lfloor \log_2 m \rfloor}$ or in a new final level, $\phi_{\lceil \log_2 m \rceil}$.

These fingerprints can be created in a streaming fashion, so each character of the pattern only needs to be read once. This can be done via dynamic programming, concatenating the current row with the fingerprint of the already computed previous row:

$$\phi_i = \begin{cases} \phi(p_0), & \text{if } i = 0 \\ \textbf{Concatenate}(\phi_{i-1}, \phi(p_{2^{i-1}}...p_{2^i-1})), & \text{otherwise} \end{cases}$$

With this structure, we can now look at what we compute as each character of the text enters our stream. When t_j enters the stream, we first compute the fingerprint $\phi(t_j)$, update our fingerprint of the text read so far $\phi(t_0...t_j)$ and check if $\phi(t_j) = \phi_0$. If this case is true, we have what is referred to as a viable occurrence (VO) for level 1. When we have a VO at level 1 after character $\phi(t_j)$ has entered the stream, we store two properties: $j-1$ and $\phi(t_0...t_{j-1})$[2] in a list of viable occurrences for level 1.

After performing the above, we retrieve the oldest VO we have stored at level 1, which has properties j' and $\phi(t_0...t_{j'-1})$. If $j - j' = 2$, we now know that enough characters have passed for us to be able to check if this viable occurrence requires promotion. We remove this occurrence from our list of VOs for level 1 and use the fingerprint suffix operation on our fingerprint of the text and $\phi(t_0...t_{j'-1})$ to retrieve $\phi(t_{j'}...t_j)$. We then check if $\phi(t_{j'}...t_j) = \phi_1$ and if this is the case, we promote this occurrence by storing j' and $\phi(t_0...t_{j'-1})$ in a list of viable occurrences for level 2. Otherwise, we discard the occurrence.

We repeat the above process $\log_2 m$ times per character. At the i-th level, we check if the oldest VO occured 2^i characters back and if so, we then check if the fingerprint of the last 2^i characters matches the fingerprint of the 2^i prefix of the pattern. If they match, we promote this occurrence to the $i+1$-th level. At the final level, we check if the oldest VO for this level occured m characters ago. If so, we check if the fingerprint of the last m characters of the text matches the fingerprint of the whole pattern. If they do match, then a match is reported at index j, where t_j was the last character read.

This algorithm gives us $O(\log m)$ time per character, but the space complexity is still linear. This can be easily seen if the text and pattern are both strings of the letter a. After 6 characters, the list of viable occurrences for each level would look like similar to the example given in Figure **??**. Note that level 0 is not included in the aforementioned figure as there are no VOs stored for that level.

At level i, we have to store up to 2^{i-1} viable occurrences. The final row has to store at most $\frac{m}{2}$ viable occurrences. Storing these VOs naïvely in a list will result in $1 + 2 + ... + 2^{i-1} + ... + \frac{m}{2} \in O(m)$ space being used overall, so that is not an option. But there is a way of compressing these VOs.

Consider what has happened when level i receives a promotion from level $i-1$ at index j. This means that the fingerprint $\phi(t_{j-2^{i-1}}...t_j)$ matched ϕ_{i-1}. Now consider if level i receives a promotion at index $j+1$. Now both the fingerprints $\phi(t_{j-2^{i-1}}...t_j)$ and $\phi(t_{j-2^{i-1}+1}...t_{j+1})$ matched ϕ_{i-1}. Assuming that a collision did not occur in the Karp-Rabin fingerprinting – an assumption that holds with at least probability $1 - \frac{1}{n^{1+\alpha}}$ since the associated strings are the same length and fingerprinting parameters p and

[2] If $j = 0$ then -1 and the fingerprint of the empty string will be stored as a VO.

j	0	1	2	3	4	5	6
t_j	a	a	a	a	a	a	a
VO for level 3 starting at -1	a	a	a	a			
VO for level 3 starting at 0		a	a	a	a		
VO for level 3 starting at 1			a	a	a	a	
VO for level 3 starting at 2				a	a	a	a

Figure 2.2: Example state of VO list for level 3 after 7 characters, where $T = aaaaaaa$ and $P = aaaaaaaa$

r have been picked correctly – it must hold that $t_{j-2^{i-1}}...t_j = t_{j-2^{i-1}+1}...t_{j+1}$. In order for this to be the case, it is necessary that the prefix $p_0...p_{2^{i-1}-1}$ repeats itself.

We can see this in the example where the text and pattern are just strings of the letter a. If we consider a more detailed look at where the viable occurrences are promoted to level 3, as shown in Figure 2.2, we can see that the only reason we need to store $2^{3-1} = 4$ VOs is because the 4 character prefix of the pattern is so repetitive.

It is at this point that we shall describe the period of a string. For any string T of length n, the period ρ_T is the shortest prefix of T which we can repeat $\frac{n}{|\rho_T|}$ times in order to re-create T. For the situation shown in Figure 2.2, the period of the pattern prefix $\rho_{p_0...p_3} = a$.

More generally, if level i needs to store more than one VO at a given point, the prefix $p_0...p_{2^{i-1}-1}$ must be periodic. We can now store the VOs for a given level not as a list, but as an arithmetic progression, with the following properties:

- The location and fingerprint of the oldest VO we need to store
- The location and fingerprint of the newest VO currently stored
- The fingerprint of the period
- The length of the period
- The number of VOs currently stored

The fingerprint and length of the period can both be computed when we need to store two VOs at a given level: The length by taking the second VO location and subtracting the first VO location, and the fingerprint by working out the suffix of the second VO fingerprint and the first VO fingerprint. Both of these are constant time operations.

When we want to remove a VO from a row, we update the oldest location by adding on the length of the period, update the oldest fingerprint by concatenating it with the fingerprint of the period, and decrement our counter. Again, this is a constant time operation.

There is however, a caution about this method. It must be remembered that we are not comparing the strings directly; we are merely comparing fingerprints of them. Thus if there is a collision in the fingerprints, we might have a case where the strings are not periodic.

We can check for this when we insert a new VO into an arithmetic progression. If there are two or more VOs already stored, we compare the difference between the location of the newest VO currently stored and the location of this new VO, and also check the suffix of the new VO's fingerprint with the fingerprint of the newest VO currently stored. If these two values are equal to the length and fingerprint of the period, then we store this new VO by incrementing the number of VOs currently stored and continue as usual.

If the above condition does not hold and these two conditions do not match, there is no clear consensus on how to handle this case of a non-periodic VO. Porat and Porat themselves ignore this case, and simply accept that there is a possibility of both false positives and false negatives. Breslauer and Galil[5] recommend not inserting the occurrence into the pattern, yet reporting the index as a match against the whole pattern anyway to accept some chance of false positives yet still finding all instances of the pattern.

Independent of whether or not the condition holds, inserting and removing VOs can be performed in constant time and the VOs for a given row can be stored compactly in $O(1)$ space. Because there are $\lfloor \log_2 m \rfloor$ levels, the overall algorithm now uses $O(\log m)$ in both space and time per character.

2.7 Binary Search Trees

A Binary Search Tree (BST)[9] is a tree where each node has at most two children and for every node in the tree, all the descendants to the left of the tree have a smaller value than the given node, and those to the right have a larger value. The height of a BST is determined by the longest distance from any leaf to the root of the tree, and a BST is self-balancing if its height is kept small regardless of what items are inserted or removed. Because a lot of BST operations run in time dependent on the height of the tree, keeping this factor small is important.

Of particular note are Red-Black Trees (RBT)[10], which are the binary search trees used in this project. Their time complexity when containing n items is $O(logn)$ for insert, search and delete and $O(n)$ Because this is used as a library function and not implemented by myself, we will not go into detail on how RBTs work. For more information on Red-Black Trees, I encourage the reader to consult the CLRS chapter cited above, the original paper by Bayer[2], and the website for the implementation used in this project: http://en.literateprograms.org/Red-black_tree_%28C%29

2.8 Clifford, Fontaine, Porat and Sach: Dictionary Matching in Sublinear Space

Clifford et al.[7] provided a solution to dictionary matching under the streaming model in less space than it takes to store the pattern. Their solution uses $O(k \log m)$ space and $O(\log m)$ time per character, where $m = \max(M)$. It is worth noting that this description is based off of a version of this paper that was not accept, and the accepted version of the paper will likely have some differences to what is described here. Also note that these are how the algorithm is described in the paper; any changes to how the algorithm works in later chapters are my own work to correct the algorithm.

The first step to understanding the algorithm as described in the paper is to consider a subset of the dictionary matching problem, where all the patterns are a power of two in length. This algorithm is very similar to the one described in Section 2.6, where all the patterns are broken up into $\log m_i$ fingerprints, denoted $\phi_{i,j}$ and defined as follows:

$$\phi i, j = \phi(p_{i,0} ... p_{i,2^j - 1})$$

Each level of the algorithm now contains up to k of these prefix fingerprints, and stores all of them in a static perfect hash table. Along with this, each level stores up to k aritmetic progressions of viable occurrences. When the next character enters the stream, each level checks if one of the arithmetic progressions has a viable occurrence that requires processing. If there is, then the algorithm uses static perfect hashing to check if the last 2^j characters in the text match any of the prefixes at that level. If there is a match, that VO is promoted to the next level. Finally, a flag is also specified in the hash table

to indicate if the given fingerprint is actually the fingerprint of a whole pattern. If that flag is true, then a match at that index is reported.

In terms of complexity, each level requires $O(k)$ space and there are $\log m$ levels, so space usage is $O(k \log m)$ as required. Time complexity depends on how long it takes to figure out if an arithmetic progression needs processing and if so which one. Assuming this can be done in constant time, then each level takes constant time and thus an overall performance of $O(\log m)$ time per character is given.

In order to go from this to the general case of any length of pattern, the patterns are broken up into three cases, based on their length m_i and period $\rho_i = \rho_{P_i}$:

$$|\rho_i| \geq k; (m_i \geq k \text{ and } |\rho_i| < k); m_i < k$$

2.8.1 Patterns with Long Periods

We start with the case where for every pattern P_i in our dictionary $|\rho_i| \geq k$. We start by defining Q_i to be the $m_i - k$ prefix of the i-th pattern. For this algorithm to work, we continue under the assumption that $|\rho_{Q_i}| \geq k$. We will see a brief solution for when this assumption does not hold in the subsubsection at the end of this case.

The first part of this case works the same as in the power of two length case. We perform the above algorithm on $\log |Q|$ levels, where Q is the prefix of the longest pattern. If there is a match at a given level, we insert the viable occurrence that matched into a special row, which stores one arithmetic progression for each prefix. At each text index j, we process two prefixes. Let Q_i be one of those prefixes processed at j, and we perform the following:

1. First, if $j \geq l + |Q_i|$, where l is the location of the VO stored in the arithmetic progression related to Q_i, then we check if $\phi(t_{l+1}...t_{l+Q_i+1}) = \phi(Q_i)$. In other words, did the $|Q_i|$ characters following the VO location match the prefix?
2. If there is a match, then we insert all the fingerprints for the k length suffixes of all the patterns in the dictionary for which Q_i is a prefix into a binary search tree (BST). The binary search tree is set up so that it will only be queried when the stream reaches index $l + |Q_i| + k$.

Finally, the algorithm checks to see if any binary search trees need processing. If so, the algorithm takes the fingerprint of the last k characters seen in the stream, and searches the BST to see if a match is found. If a match is found, then an entire pattern has been matched, and the index j is returned.

Time complexity wise, processing the power of two length prefixes costs $O(\log |Q|)$ time per character by simply substituting $m = |Q|$ into the previous definitions. The first step of processing each prefix takes constant time, but may be delayed by up to $\frac{k}{2}$ characters. The second step is more complicated, and in the worst case – where all the patterns have the same $m_i - k$ length prefix – will take $O(k \log k)$ time per character if implemented naïvely. However, because of our assumption that $\forall i, |\rho_{Q_i}| \geq k$, each prefix can only occur once every k characters. This means that, amortised over k characters, our time complexity becomes $O(\log k)$ time per character. Furthermore, this can be deamortised by inserting two suffixes into the BST per index, bringing our worst case time complexity for this step down to $O(\log k)$ per character. Steps 1 and 2 are both delayed by at most $\frac{k}{2}$ indexes each, so the overall delay will be at most k indexes, within time for the BST to be processed. Finally, searching the BST takes $O(\log k)$ time. Putting all of this together gives us $O(\log |Q| + \log k)$ time per character, and because of our assumption that $\rho_Q \geq l$, this becomes $O(\log |Q|) \in O(\log m)$ time per character.

As for space usage, the power of two length prefixes uses $O(k \log |Q|)$ space. Storing the fingerprints of all the prefixes costs $O(k)$ space, as does storing lists of all the suffixes. To cater for both the $\frac{k}{2}$ delay

in the first step of the prefix processing and searching for the fingerprint of the last k characters in a BST, we store a circular buffer of the last k fingerprints, which takes up $O(k)$ space. Finally, we may need up to k binary search trees, and at any given time the total number of nodes across all BSTs is at most k, so this again is $O(k)$ space. This gives us an overall space usage of $O(k \log |Q|) \in O(k \log m)$ space.

An Edge Case

As previously mentioned, there is an edge case in the above algorithm if $\exists i$ such that $|\rho_{Q_i}| < k$. Any patterns which fall under this case can be processed by using the algorithm for long patterns with short periods described in Section 2.8.2 to process their prefix Q_i. Any matches returned from this algorithm can be extended from $m_i - k$ to m_i by a combination of fingerprinting and static perfect hashing.

2.8.2 Long Patterns with Short Periods

The next case detailed is where the patterns in the dictionary are longer than k, but their periods are shorter. In this algorithm, we store a fingerprint of the k length prefix of each pattern, and call the result K_i. For each K_i, we store a fingerprint of the period of the pattern $\phi(\rho_i)$ and the period's length $|rho_i|$ along with a counter of the number of times it has occurred the index of the last time it occurred in the current arithmetic progression and the fingerprint of the text at the last occurrence.

When a new index comes in, we use a static perfect hash function to check if the fingerprint of the last k characters in the text stream matches some K_i. If so, we check if this index fits in with the rest of K_i's arithmetic progression by checking the current index is $|\rho_i|$ characters away from the last occurrence and the fingerprint suffix of the current stream with the fingerprint of the stream at the last occurrence matches the fingerprint of the period. If they do match then we increment the counter, otherwise we abandon the arithmetic progression by resetting the counter to 1 and setting the last occurrence to the current index and fingerprint.

The second step we perform is to check if the fingerprint of the last k characters in the text matches the last k characters in one of the patterns – referred to as the *tail* of each pattern. This can be done by storing the fingerprint of the tails of each pattern in another static perfect hash table.

At this point, we are going to assume that no patterns are a suffix of another pattern. If this is not the case, then it is possible to perform dictionary matching, but it comes at the cost of not knowing which patterns have matched. This is why for our choice of dictionary matching, we do not care about what pattern(s) have matched. It is also why we can only return the index at the end of the instance; the only way we could know the starting index of the would be if we knew what the pattern was, since then we could just simply take the current index and subtract the length of the pattern.

Anyway, either through no pattern being a suffix of any other pattern or through discarding the patterns which do have other patterns as suffixes, we end up in a situation where each pattern has a unique tail. If the last k characters in the stream match the tail of some pattern P_i, we check the arithmetic progression associated with that pattern. In order for a match to have occured, there need to have been at least $\lfloor \frac{m_i}{\rho_i} \rfloor$ occurrences of K_i in the progression, and the last occurrence must have happened $m_i \mod \rho_i$ characters ago. If both of these conditions hold, then a match is reported.

Complexity wise, the progressions can be stored in $O(k)$ space, as can the fingerprints of the tails. The fingerprint of the last k characters in the text stream can be stored by using a circular array of such fingerprints for the last k indexes. Thus space usage is $O(k)$. As for time, the static perfect hashing operations are constant time, as is inserting the occurrence into the arithmetic progressions and checking for a match, so time overall is $O(1)$.

if $y \geq x/2$ **then**
 | $\mathcal{H}_3$.insert($\phi(p_{y-x/2}...p_{m_i}), \mathcal{S}(p_{y-x/2}...p_{m_i})$) `PreProc`$(x/2, y - x/2)$
else
 | `PreProc`$(x/2, y)$
end

Algorithm 2.5: `PreProc`(x, y): Preprocessing of a single pattern

2.8.3 Short Patterns

The final case to consider is where all the patterns are shorter than k. The algorithm for this case is an adaptation of binary search, searching over suffixes of the stream of lengths from 1 to k to see if a pattern matches any of them. However, binary search cannot be applied naïvely, as we may need to search both parts of the search space. Instead, we use a hash table with a fingerprint as the key and a boolean as the value to find out which half of the space to search.

Algorithm 2.5 computes the hash table that is used. We call `PreProc`(k', m_i) on each pattern P_i, where k' is the nearest integer power of two no smaller than k. Note that $\mathcal{S}$ is a boolean function which, given a string, returns True if there is a pattern in the dictionary which is a suffix of that string, and False otherwise.

At each index j, we keep a cyclic buffer of the previous k' fingerprints of the whole stream. We start our binary search by seeing if $\phi(t_{j-k/2+1}...t_j)$ is within the hash table and its associated boolean value. If the boolean value is True, we know that there is an occurance of the pattern which ends at this index, so we report it. If the boolean value is false but the key exists in the table, then we know that this fingerprint matches the suffix of a pattern, but not necessarily all of it, so we check the longer suffixes of the text. If there is no key in the table, then we know that no pattern matches this suffix, but they might match shorter ones, so we check shorter suffixes instead. This continues until we either report an instance or run out of search space.

For space complexity, the only space used is the hash function which uses $O(k \log k)$ space to store the fingerprints and their associated booleans and the cyclic buffer, which uses $O(k') \in O(2k) \in O(k)$ space, so the overall space usage is $O(k \log k)$. The only step in the algorithm is the binary search, which takes $O(\log k)$ time per character.

2.9 Arbitrary-Precision Arithmetic

Arbitrary-Precision Arithmetic are libaries which allow for computation of numbers beyond what is capable of a typical machine. Common applications include Cryptography and linear algebra.

Again, little detail will be provided here due to the fact that this is not implemented and only used as an external library. For more information, feel free to visit the GNU Multiple Precision Arithmetic Library (GMP) website at https://gmplib.org/.

Chapter 3

Project Execution

A topic-specific chapter, of roughly 20 pages

This chapter is intended to describe what you did: the goal is to explain the main activity or activities, of any type, which constituted your work during the project. The content is highly topic-specific, but for many projects it will make sense to split the chapter into two sections: one will discuss the design of something (e.g., some hardware or software, or an algorithm, or experiment), including any rationale or decisions made, and the other will discuss how this design was realised via some form of implementation.

This is, of course, far from ideal for *many* project topics. Some situations which clearly require a different approach include:

- In a project where asymptotic analysis of some algorithm is the goal, there is no real "design and implementation" in a traditional sense even though the activity of analysis is clearly within the remit of this chapter.
- In a project where analysis of some results is as major, or a more major goal than the implementation that produced them, it might be sensible to merge this chapter with the next one: the main activity is such that discussion of the results cannot be viewed separately.

Note that it is common to include evidence of "best practice" project management (e.g., use of version control, choice of programming language and so on). Rather than simply a rote list, make sure any such content is useful and/or informative in some way: for example, if there was a decision to be made then explain the trade-offs and implications involved.

3.1 Example Section

This is an example section; the following content is auto-generated dummy text. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi

dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

3.1.1 Example Sub-section

This is an example sub-section; the following content is auto-generated dummy text. Notice the examples in Figure 3.1, Table 3.1, Algorithm 3.1 and Listing 3.1. Lorem ipsum dolor sit amet, consectetuer

Figure 3.1: This is an example figure.

foo	bar	baz
0	0	0
1	1	1
⋮	⋮	⋮
9	9	9

Table 3.1: This is an example table.

adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed

for $i = 0$ **upto** n **do**
  $t_i \leftarrow 0$
end

Algorithm 3.1: This is an example algorithm.

```
for( i = 0; i < n; i++ ) {
  t[ i ] = 0;
}
```

Listing 3.1: This is an example listing.

ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Example Sub-sub-section

This is an example sub-sub-section; the following content is auto-generated dummy text. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Example paragraph. This is an example paragraph; note the trailing full-stop in the title, which is intended to ensure it does not run into the text.

Chapter 4

Critical Evaluation

A topic-specific chapter, of roughly 10 **pages**

This chapter is intended to evaluate what you did. The content is highly topic-specific, but for many projects will have flavours of the following:

1. functional testing, including analysis and explanation of failure cases,
2. behavioural testing, often including analysis of any results that draw some form of conclusion wrt. the aims and objectives, and
3. evaluation of options and decisions within the project, and/or a comparison with alternatives.

This chapter often acts to differentiate project quality: even if the work completed is of a high technical quality, critical yet objective evaluation and comparison of the outcomes is crucial. In essence, the reader wants to learn something, so the worst examples amount to simple statements of fact (e.g., "graph X shows the result is Y"); the best examples are analytical and exploratory (e.g., "graph X shows the result is Y, which means Z; this contradicts [1], which may be because I use a different assumption"). As such, both positive *and* negative outcomes are valid *if* presented in a suitable manner.

Chapter 5

Conclusion

A compulsory chapter, of roughly 2 pages

The concluding chapter of a dissertation is often underutilised because it is too often left too close to the deadline: it is important to allocation enough attention. Ideally, the chapter will consist of three parts:

1. (Re)summarise the main contributions and achievements, in essence summing up the content.
2. Clearly state the current project status (e.g., "X is working, Y is not") and evaluate what has been achieved with respect to the initial aims and objectives (e.g., "I completed aim X outlined previously, the evidence for this is within Chapter Y"). There is no problem including aims which were not completed, but it is important to evaluate and/or justify why this is the case.
3. Outline any open problems or future plans. Rather than treat this only as an exercise in what you *could* have done given more time, try to focus on any unexplored options or interesting outcomes (e.g., "my experiment for X gave counter-intuitive results, this could be because Y and would form an interesting area for further study" or "users found feature Z of my software difficult to use, which is obvious in hindsight but not during at design stage; to resolve this, I could clearly apply the technique of Smith [7]").

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[3] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer Berlin Heidelberg, 2009.

[4] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[5] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4):22:1–22:12, August 2014.

[6] K. Seluk Candan and Maria Luisa Sapino. *Data Management for Multimedia Retrieval*, pages 205–206. Cambridge University Press, May 2010.

[7] Raphaël Clifford, Allyx Fontaine, Ely Porat, and Benjamin Sach. Dictionary matching in a stream. Modifications to the algorithm are already in development. Version used for this project was the latest version available at the time that implementation began., February 2015.

[8] Beate Commentz-Walter. A string matching algorithm fast on the average. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, 1979.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 286–298. MIT Press, 2009.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 308–338. MIT Press, 2009.

[11] Richard M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[12] Donald E. Knuth, jr Morris, James H., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[13] B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS '09. 50th Annual IEEE Symposium on*, pages 315–323, Oct 2009.

Appendix A

An Example Appendix

Content which is not central to, but may enhance the dissertation can be included in one or more appendices; examples include, but are not limited to

- lengthy mathematical proofs, numerical or graphical results which are summarised in the main body,
- sample or example calculations, and
- results of user studies or questionnaires.

Note that in line with most research conferences, the marking panel is not obliged to read such appendices.