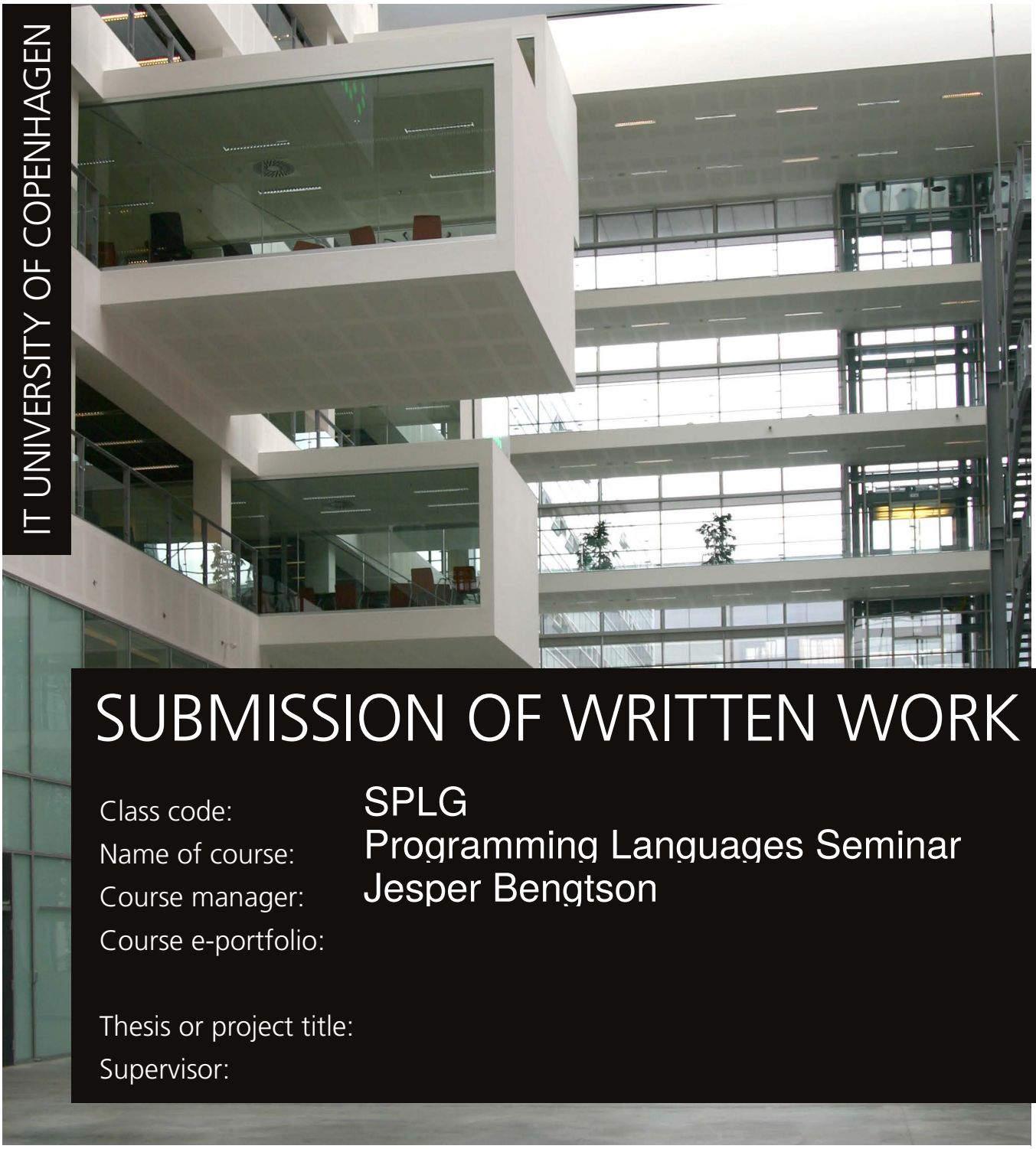

SUBMISSION OF WRITTEN WORK

Class code: SPLG
Name of course: Programming Languages Seminar
Course manager: Jesper Bengtson
Course e-portfolio:

Thesis or project title:
Supervisor:

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Ankush Jindal	01/01-1995	anji @itu.dk
2.	Kasper Stig Henningsen	14/09-1993	ksti @itu.dk
3.	Jonas Kastberg Hinrichsen	09/07-1992	jkas @itu.dk
4.	Simon Oliver Malone	12/03-1991	soma @itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Todo list

Proving the Correctness of String Search Algorithms in Coq

Simon Malone
soma@itu.dk

Kasper Stig Henningsen
ksti@itu.dk

Ankush Jindal
anji@itu.dk

Jonas Kastberg Hinrichsen
jkas@itu.dk

25th May 2016

Contents

List of Figures

List of Terms

KMP Knuth-Morris-Pratt. 1, 5, 8–13, 15–17

time complexity The amount of a specific given action the algorithm has to take to solve the problem. In this report, the time complexity will most often relate to comparisons between characters.. 1–5, 11, 13, 15, 16, 19

1 | Introduction

In this report we will investigate the mathematical proofs of various properties of some sting search algorithms, using the interactive proof assistant Coq.

The report will elaborate on the choices of theorems for each algorithm, as well as how each theorem was approached. Each approach will be compared to the actual proof, along with a discussion on the differences.

The string search algorithms in particular are a naive string search and Knuth-Morris-Pratt (KMP)[1].

1.1 Problem Statement

The goal of this project is to prove the correctness and time complexities of various string search algorithms, while gaining an understanding in how the proof of algorithms differ, in regards to their underlying implementation.

We will investigate the following algorithms:

- A naive string search algorithm
- The Knuth-Morris-Pratt (KMP) string search algorithm

The proofs are to be done in the interactive proof assistant Coq and we will utilise libraries and tools provided by Coq. It is assumed that these are correct and proven so.

Proving the algorithms follow these four steps:

- Implementing the algorithms in Coq.
- Defining which theorems of each algorithm that should be proven.
  - These must collectively prove the correctness and time complexity of the algorithm.
- Defining and proving the theorems Coq.
- Documenting the planned approach and practical proof of the theorems.

The deliverables of the project will be the Coq source-code, in which the proofs have been conducted, and this report, describing the proofs and any possible findings.

[1] `http://www.cs.utexas.edu/~eberlein/cs337/patternMatching.pdf` (visited on 2016-05-24)

2 | Problem Analysis

This chapter will elaborate on the choice of theorems that should been proven, and the reasoning behind these choices.

First is a description of the generally defined theorems, which we wish to prove for all algorithms. Then follows an overview of the necessary variations from the general definitions, that was introduced for either algorithm. This includes helper-functions (sub-algorithms), changes to the theorem definition and more.

2.1 Choice of Theorems

This section will elaborate on the chosen theorems, that should be proven for all algorithms. As mentioned in 1.1, these must collectively prove the **correctness** and **time complexity** of the algorithm.

To prove either of these, they have been split into separate properties which, in combination, proves the concept in general. The following will account for these properties and the theorems that will prove them.

2.1.1 Correctness

The general definition of a string search algorithm, is that it should search a given string `s`, for a substring `t`, and return the first index `n`, where the substring `t` is found in `s`.

The most commonly accepted error-case, is to return the index 0, or the error signal -1. We take from this that the algorithm must have a clearly defined behaviour in case a substring is not found.

From this, we have defined the three theorems, as described in the following subsections

Correctness hit

This theorem verifies that the algorithm works as intended, in the case that a substring is found. The general definition of the theorem has been defined as follows:

$$\forall\ s\ t\ n.\ \texttt{<algorithm>}\ s\ t\ = n \rightarrow \texttt{substring}\ n\ |t|\ s = t$$

That is, if the algorithm successfully finds an index, that index is indeed the start of a substring that matches the target. This proves that the algorithm correctly identifies the index of a correct substring.

Correctness miss

This theorem verifies that the algorithm works as intended, in the case that a substring is not found. The general definition of the theorem has been defined as follows:

$$\forall\ s\ t.\ \texttt{<algorithm>}\ s\ t\ = \texttt{<error signal>} \rightarrow \forall\ i.\ \texttt{substring}\ i\ |t|\ s \neq t$$

In other words, if the algorithm fails, it does so because the target string does not exist in the source string.

Correctness first

This theorem verifies that if the algorithm finds some substring, it is the first one. The general definition of the theorem has been defined as follows:

$$\forall\ s\ t.\ \texttt{<algorithm>}\ s\ t\ = n \rightarrow \forall\ i.\ i < n \rightarrow \texttt{substring}\ i\ |t|\ s \neq t$$

That is, no matter where in the source string we look, as long as it is before the found index, we cannot find a substring that matches the target. Therefore, when we know that if the found index is indeed the start of a correct substring, it must also be the first correct substring.

2.1.2 Time complexity

Each of the algorithms has a defined time complexity. Specifically, each algorithm has a concrete upper bound and lower bound, such that for any input, the amount of iterations needed to terminate, is some amount within the bounds.

To determine if an algorithm terminates in time, an alternation of each original algorithm has been defined, such that they take an additional input, defining the allowed number of iterations. The algorithm then returns whether or not it terminates within the giving amount of iterations.

In the following, this variation is referred to as `algorithm_term`.

The time complexity bounds are thus proven using four theorems, as outlined in the following subsection.

Above upper bound

This theorem verifies that the algorithm always terminates if allowed a number of iterations, that exceeds the upper bound.

$$\forall\ s\ t\ c.\ c > \texttt{<upper bound>} \rightarrow \texttt{<algorithm\_term>}\ s\ t\ c$$

At upper bound

This theorem verifies that there exists an input for which the algorithm does not terminate, if only allowed a number of iterations that is equal to the upper bound.

$$\forall\ c.\ \exists\ s\ t.\ c = \texttt{<upper bound>} \rightarrow \neg\ \texttt{<algorithm\_term>}\ s\ t\ c$$

Below lower bound

This theorem verifies that the algorithm never terminates if given less than the lower bound amount of iterations.

$$\forall\ s\ t\ c.\ c < \texttt{<lower bound>} \rightarrow \neg\ \texttt{<algorithm\_term>}\ s\ t\ c$$

At lower bound

This theorem verifies that the algorithm terminates for some input, if allowed a number of iterations equal to the lower bound.

$$\forall\ c.\ \exists\ s\ t.\ c = \texttt{<lower bound>} \rightarrow \texttt{<algorithm\_term>}\ s\ t\ c$$

2.2 Naive Considerations

This section elaborates on the additional considerations that was made, in regards to applying the theorems to the naive string search implementation.

The naive string search algorithm can be considered as an outer-loop, over each index of the source string, and an inner-loop over each index of the target string.

To perform the latter, we introduced an additional sub-algorithm `string_match` that has to be proven as well, since the implementation is reliant on its correctness.

The structure of the `naive` algorithm theorems, as well as the time complexity theorems of string match, are identical to the general outline presented in section 2.1, and have thus not been elaborated further.

Note that the proof of time complexity of naive will be determined in terms of iterations of `naive`, and will not decrement during `string_match`. In combination, the theorems proving the time complexity bounds of `naive` and `string_match`, proves the overall time complexity of `naive`, as it can at most call `string_match` once, during each iteration[2].

2.2.1 String matching

String match matches two given strings (`s` and `t`), character-wise, until the end of the second string. It will thus still return true when `t` is a proper prefix of `s`[3].

To prove the correctness of `string_match` we had to prove **hit** and **miss**.

String Match hit

This theorem verifies that string match behaves correctly in the case where the target string matches the beginning of the source string.

[2]The total time complexity of `naive` is then dependent on t, as the time complexity of `string_match` is so (see A.2.2)

[3]We discovered that this implementation is logically equivalent to the `prefix` function in the Coq string library. At this time, however, correctness of `string_match` had already been proven, so we stuck with it as it eases the transition to the time complexity proofs, where we could not use the aforementioned `prefix` function.

$$\forall\ s\ t.\ \texttt{string\_match}\ s\ t\ = true \rightarrow \texttt{substring}\ 0\ |t|\ s\ = t$$

String Match miss

This theorem verifies that string match behaves correctly in the case in which the strings do not match.

$$\forall\ s\ t.\ \texttt{string\_match}\ s\ t\ = false \rightarrow \texttt{substring}\ 0\ |t|\ s\ \neq t$$

2.3 Knuth-Morris-Pratt Considerations

This section elaborates on the additional considerations that were made, in regards to applying the theorems to the KMP string search implementation.

The KMP algorithm iterates over the source string, while cleverly skipping sequences that has already been checked in a previously failed match, by using a prefix table. This table is constructed before the algorithm is run, and given as a part of the input.

Each of the KMP theorems have then been extended with an assumption that the given table is correct.

This used to say: (...) cannot match, as given by a table

$$\forall\ t\ pi.\ \texttt{table\_correct}\ t\ pi\ = true \rightarrow \texttt{<theorem>}$$

The structure of the KMP algorithm theorems are identical to the general outline presented in section 2.1. During implementation we found some restrictions in regards to the proofs, that will be elaborated on later.

The complete running time analysis for the KMP algorithm is a net sum of the time complexity for the pre-processing part and the algorithm itself - that is creation of table for string `t` and the string search done thereafter.

2.3.1 KMP table

To use the table, it is necessary to prove its correctness in regards to the target string `t`. The correctness of the table is binary, as in it can only either be correct or incorrect. There is then only one theorem to be proven for correctness, verifying that a table is correct in regards to some string `t`.

We also have to prove the time complexity of the creation of the prefix table. This is done with the general approach described in 2.1.

KMP table correctness

The correctness of the KMP table is defined as the following predicate, which we denote `pi_correct`.

$$t\ pi.\ \forall\ k\ m.\ 0 < k \rightarrow k < |t| \rightarrow \texttt{look\_up}\ k\ pi = m \\ \rightarrow \texttt{prefix\_is\_suffix}\ k\ m\ t \wedge \neg\ \texttt{prefix\_is\_suffix}\ k\ (m+1)\ t$$

`prefix_is_suffix` is a predicate defined as follows.

$$k\ m\ t.\ \texttt{substring}\ 0\ m\ t = \texttt{substring}\ (k-(m-1))\ m\ t$$

If we denote the algorithm that creates a table based on some target string `t` as `make_table`, we define the theorem to be proven for correctness as:

$$\forall\ t.\ \texttt{pi\_correct}\ t\ (\texttt{make\_table}\ t)$$

3 | Implementation

This chapter will elaborate on the concrete implementation and proof of each of the string search algorithms. Each algorithm is presented, such that the Coq implementation of it, and any sub-algorithms presented in chapter 2, is elaborated upon[4].

As described in chapter 2, the algorithms should have some defined behaviour in the error-case. The implementations handle this by wrapping the result in an option, where `None` is the error-case

Maybe touch upon when the error case is, and account for the mismatch in communication that caused the KMP hit trouble?

3.1 General Variants

To more easily define our theorems, we made some general variants of each algorithm, which each encapsulates some property. The variants in question are the following[5]:

- `algorithm_prop`
- `algorithm_candy`
- `algorithm_candy_prop`

The candy variants are our terminology for the `<algorithm_term>` variants, introduced in chapter 2, where candy is an input denoting the allowed number of iterations. The result is wrapped in another option, where `None` determines that the algorithm did not finish with the given number of iterations.

The `algorithm_prop` and `algorithm_candy_prop` versions of the algorithms are variants where the output is mapped to a property (hence the name). That is `Some` maps to $True$ and `None` maps to $False$[6].

To ensure the correctness of the candy variants, a new theorem, Candy Correspondence, has been introduced for each algorithm, stating that the result of `algorithm_candy` is equal to that of `algorithm`, in any case, given that there is enough candy to terminate.

3.2 Naive Substring Search

This section elaborates on the implementation and proof of the naive substring search algorithm.

Note that the implementation and proof of string match is outlined in appendix A.2.

[4]Note that in this chapter, we will use `typewriter` font in our mathematical proofs to denote a Coq specified function or algorithm

[5]Note that we planned to always use these variation where applicable, however during implementation, we incidentally used the base algorithm in some cases, instead of the designated variant.

[6]It cannot be assumed that `algorithm_candy_prop -> algorithm_prop` as the former property is related to the amount of candy whilst the latter is related to the result of the search

3.2.1 Naive implementation

The implementation of the naive algorithm in Coq is defined as seen in figure 3.1. It is defined as a fixpoint, with the source string `s` as the decreasing element. For each recursive iteration, the source string `s` is decremented with the head character, while increasing the current index `n`.

```
Fixpoint naive (s t:string) (n:nat) : option nat :=
  match s with
    | EmptyString ⇒ match t with
      | EmptyString ⇒ Some n
      | _ ⇒ None
      end
    | String c s' ⇒  match string_match s t with
      | false ⇒ naive s' t (n+1)
      | true ⇒ Some n
      end
    end.
```

Figure 3.1: Naive implementation

In most of the proofs, `naive` is called with value `n` being some value `m` instead of 0. This was done to make the theorems more general. The reader can interpret a call `naive s t m` as some recursive call to naive, at which the first `m` characters of `s` has already been compared and removed.

3.2.2 Naive theorems

In the following, we elaborate on the proofs outlined in 2.1 and 3.1. The theorems and their proofs as defined in Coq, can be found in appendix B.13.

The following is an overview of the proven theorem as they were defined in Coq, and a short description on how they were approached.

Naive correctness hit

$$\forall\ s\ t\ m\ n.\ \texttt{naive}\ s\ t\ m\ = \texttt{Some}\ n \rightarrow \texttt{substring}\ (n-m)\ |t|\ s = t$$

The theorem was proven through induction on `s` because it is the decreasing element of the fixpoint.

Naive correctness miss

$$\forall\ s\ t\ m.\ \texttt{naive}\ s\ t\ m = \texttt{None} \rightarrow \forall\ i.\ i < |s| - |t| \rightarrow \texttt{substring}\ i\ |t|\ s \neq t$$

This theorem was proven by just using two intricate lemmas we made; `naive_none_implies_string_match_false` and `string_match_implies_substring`[7].

[7]Note that we ought to have used `naive_prop`, as the result of naive is redundant.

Naive correctness first

$$\forall\ s\ t\ m\ n.\ \texttt{naive}\ s\ t\ m = \texttt{Some}\ n \rightarrow \forall\ i.\ i < n - m \rightarrow \texttt{substring}\ i\ |t|\ s \neq t$$

As earlier, this theorem was proven by induction on `s` for the same reason. In each subsequent case it is then proven that something *False* is assumed.

Naive time complexity above upper bound

The time complexity upper bound is defined as the length of `s`.

$$\forall\ s\ t\ m\ c.\ c > |s| \rightarrow \texttt{naive\_candy\_prop}\ s\ t\ m\ c$$

Again, we prove the theorem by induction on `s`.

Naive time complexity at upper bound

$$\forall\ t\ m\ c.\ \exists\ s.\ c = |s| \rightarrow \neg\ \texttt{naive\_candy\_prop}\ s\ t\ m\ c$$

The theorem was proven by inspecting the case in which `s` is the empty string.

Naive time complexity below lower bound

The time complexity lower bound is defined as 1.

$$\forall\ s\ t\ m\ c.\ c < 1 \rightarrow \neg\ \texttt{naive\_candy\_prop}\ s\ t\ m\ c$$

The theorem is trivially true, as $c < 1$ implies that $c = 0$. The algorithm can never run with $c = 0$.

Naive time complexity at lower bound

$$\forall\ t\ m\ c.\ \exists\ s.\ c = 1 \rightarrow \texttt{naive\_candy\_prop}\ s\ t\ m\ c$$

The theorem was proven by inspecting the case where `s` is the empty string, and then simplifying the goal until it became trivially true.

Naive candy correspondence

$$\forall\ s\ t\ m\ c\ r.\ c > |s| \rightarrow \texttt{naive\_candy}\ s\ t\ m\ c = \texttt{Some}\ r \iff \texttt{naive}\ s\ t\ m = r$$

The theorem was proven through induction on s, for both implications.

3.3 KMP String Search

This section elaborates on the implementation and proof of the KMP string search algorithm.

3.3.1 KMP implementation

The implementation is as follows.

```
Fixpoint kmp_candy (s t : string) (m i candy : nat) (pi : list nat) :=
  match candy with
  | 0 ⇒  None
  | S candy' ⇒
    if (m+i) <? (length s)
      then
         if string_dec (substring i 1 t) (substring (m+i) 1 s)
         then
            if (i+1 =? length t)
            then Some(Some(m))
            else kmp_candy s t m (i+1) candy' pi
         else
            if i =? 0
            then kmp_candy s t (m+i+1) 0 candy' pi
            else kmp_candy s t (m+i-(look_up i pi)) (look_up i pi)
    ↪ candy' pi
      else Some(None)
  end.
```

Figure 3.2: KMP implementation

Due to the lack of a decreasing argument, the candy version of the algorithm was used for the base implementation, where candy is the decreasing argument.

Two arguments, `m` and `i`, have been introduced, which denotes the current index of `s` and `t` respectively. This was necessary, as KMP does not systematically advance the strings like Naive does.

Is this still the order (m for s and i for t)?

3.3.2 KMP theorems

In the following, we elaborate on the outline in 2.1 and 3.1.

We did not manage to provide complete proofs for all the theorems, however most of them were proven, provided the assumption of some lemmas. All unproven lemmas are listed in appendix A.3. We did not manage to prove the theorem Correctness First, even with some assumed unproven lemmas.

KMP correctness hit

$$\forall\ s\ t\ pi\ c\ n.\ c > (2 * |s|) \rightarrow \texttt{pi\_correct}\ t\ pi\ \rightarrow$$
$$\texttt{kmp\_candy}\ s\ t\ 0\ 0\ c\ pi = \texttt{Some}\ (\texttt{Some}\ n)\ \rightarrow \texttt{substring}\ n\ |t|\ s\ = t$$

The Theorem was proven correct, albeit assuming some unproven lemmas.

We approached the theorem through induction on `s`, even though it is not the decreasing element.

The used unproven lemmas are as follows:

$$\forall\ s\ t\ a\ c\ pi\ n.\ c > 2 * |(\texttt{String}\ a\ s)| \rightarrow n > 0 \rightarrow \texttt{pi\_correct}\ t\ pi\ \rightarrow$$
$$\texttt{kmp\_candy}\ (\texttt{String}\ a\ s)\ t\ 0\ 0\ c\ pi\ = \texttt{Some}\ (\texttt{Some}\ n) \rightarrow$$
$$\texttt{kmp\_candy}\ s\ t\ 0\ 0\ c\ pi\ = \texttt{Some}\ (\texttt{Some}\ (n-1)).$$

As the substring is not found on the first index, the nature of the algorithm remains unchanged, when given the same string without the first character, since the algorithm never goes back in the source string. The substring will then be found at the same index, but at one index earlier, since the algorithm starts at one index in advance in regards to the assumed call.

$$\forall\ s\ t\ a\ a0\ c\ pi.\ c > 2 * |(\texttt{String}\ a0\ s)| \rightarrow a \neq a0 \rightarrow$$
$$\texttt{pi\_correct}\ (\texttt{string}\ a\ t)\ pi\ \rightarrow$$
$$\texttt{kmp\_candy}\ (\texttt{String}\ a0\ s)\ (\texttt{String}\ a\ t)\ 0\ 0\ c\ pi\ \neq \texttt{Some}\ (\texttt{Some}\ 0).$$

If the first character of the input strings differ, the substring can never be found on the very first index of the source string. This is trivial from inspecting the KMP implementation, in which the only case that can result in `Some` (`Some` 0) is when the two first characters match.

$$\forall\ s\ t\ a\ c\ pi\ pi'.\ c > 2 * |(\texttt{String}\ a\ s)| \rightarrow \texttt{pi\_correct}\ (\texttt{String}\ a\ t)\ pi\ \rightarrow$$
$$\texttt{pi\_correct}\ t\ pi'\ \rightarrow \texttt{kmp\_candy}\ (\texttt{String}\ a\ s)\ (\texttt{String}\ a\ t)\ 0\ 0\ c\ pi$$
$$= \texttt{Some}\ (\texttt{Some}\ 0) \rightarrow \texttt{kmp\_candy}\ s\ t\ 0\ 0\ c\ pi\ = \texttt{Some}\ (\texttt{Some}\ 0).$$

Due to inconsistencies in how edge-cases of the naive and KMP algorithms are handled, this lemma was made expecting that the empty string is always an immediate substring. The implemented definition of KMP returns `None` in this case, and so the lemma is false.

Otherwise, the lemma can be argued to be true, in that if the substring is found at the first index of the source string, then removing the first character from both non-empty strings and using them as input must result in the same output.

KMP correctness miss

$$\forall\ s\ t\ pi\ c\ m.\ c > 2 * |s| \rightarrow \texttt{pi\_correct}\ t\ pi \rightarrow$$
$$\texttt{kmp\_candy}\ s\ t\ 0\ 0\ c\ pi = \texttt{None} \rightarrow m < |s| - |t| \rightarrow \texttt{substring}\ m\ |t|\ s \neq t$$

The proof relies on the `KMP_match_none_implies_string_match_false` lemma which is unproven. The correctness of the lemma can, however, be shown by use of case analysis on the algorithm. The alogrithm, returns `None`, only if `candy` matches with 0. Assuming, `candy` to be more than the upper bound of the algorithm, we just need to show that the shifts that are made with the help of prefix table are genuine. At each recursive call, the shifts in the string `s` are such that the largest suffix of the string `s` till index `m` is the prefix of string `t` till index `i`.This brings consistency in the skips and hence, when `kmp_candy` return none, there can not be a `string_match`

KMP correctness first

$$\forall\ s\ t\ pi\ c\ n.\ c > 2 * |s| \rightarrow \texttt{pi\_correct}\ t\ pi \rightarrow \texttt{kmp\_candy}\ s\ t\ 0\ 0\ c\ pi = \texttt{Some}\ (\texttt{Some}\ n) \rightarrow \forall\ i.\ i < n \rightarrow \texttt{substring}\ i\ |t|\ s \neq t$$

The theorem was attempted to be proven by using induction on string `s`, base case of which was solved using lemma `KMP_match_result_less_than_length`. The lemma `KMP_match_result_less_than_length` states that if KMP finds a match, the index that it returns is always smaller than the length of string `s`. This is correct since, `kmp_candy` always return the index of string `s`.

The inductive case is attempted to be proven using multiple, but trivial, lemmas and doing case destruct on `pi`. This result in reducing the inductive case to the problem of inductively proving optimality of KMP search for searching string that is one character larger than string `t` in the string `s`, when given that search of string `t` is optimal on `s`.

The difficulty faced in proving the case in hand is by definition of the table `pi` defined on `t` can't be used to recreate the table for string one larger than `t` that is `a++t` for any given ascii `a`.

This theorem is thus, left unproven.

KMP time complexity above upper bound

$$\forall\ s\ t\ pi\ candy.\ \rightarrow t \neq \lambda \rightarrow |t| <= |s| \rightarrow candy > 2 * |s| \rightarrow \texttt{kmp\_candy}\ s\ t\ 0\ 0\ candy\ pi \neq \texttt{None}$$

The upper bound of KMP string search was figured out to be $2 * |\texttt{s}|$[8]. Using this upper bound, one can argue that if the `kmp_candy` is called with `candy` more than this bound, the algorithm should always successfully terminate.

The proof of time complexity for KMP is expressed in a more general form. The proof does not assume a correct table for the string `t` but any given table. A proof of upper bound for all tables will trivially hold true for the correct table as well. This choice was made to simplify the condition for which the theorem holds and thus, simplifying the proof outline.

The theorem is proven using induction on string `s`. The base case of which could be solved by omega, after doing some simplifications. For the inductive case, inversion is done on the length of string `t`, which is further solved by the use of two intricate lemmas `kmp_candies_enough_when_equal_length` and `kmp_candy_inc`.

The lemma `kmp_candies_enough_when_equal_length` states that for the case when $|\texttt{t}| = |\texttt{s}|$ then $2 * |\texttt{t}| < c$. The lemma `kmp_candy_inc` uses case analysis to state that if `kmp_candy` $s\ t\ 0\ 0\ candy\ pi \neq$ `None` then
`kmp_candy` (`String` $a\ s$) $t\ 0\ 0$ (`S S` $candy$) $pi \neq$ `None`

The upper bound for the KMP can be proven intuitively. It should be noted that from the implementation that the algorithm can terminate from two causes; the decreasing `candy` and the increasing `m + i`. If `m + i` becomes equal to the length of string `s`, when `candy` is still not 0, the successful termination will

[8] `http://www.eecs.tufts.edu/~mcao01/2010f/COMP-160.pdf` (visited on 2016-05-24)

occur. Hence, the upper bound would be greater than number of recursive calls required before $\texttt{m} + \texttt{i} = |s|$.

One can also see from the implementation that apart from a single case of `kmp_candy s t (m+i−(look_up i pi)) (look_up i pi) candy' pi`, every other recursive call increases the `m+i` with exactly `1`. However, the case `kmp_candy s t (m+ i − (look_up i pi)) (look_up i pi) candy' pi` does not increase the sum at all. This case can be destructed according to value of `look_up i pi` for it. If the value of it is 0, then such case of recursive calls can only be made once consecutively. In the other case the value of `look_up i pi`, `m` increases and `i` decreases. These types of consecutive calls can only be made `i` times, before `i` reaches 0, after which this type of recursive calls can be applied any more.

In totality, the number of recursive calls before KMP finishes working is twice the length of string `s`. This establishes the upper bound of the KMP algorithm.

KMP time complexity at upper bound

$$\forall\ c.\ \exists\ t\ s\ pi.\ c = 2 * |s| \rightarrow \texttt{pi\_correct}\ t\ pi \rightarrow \texttt{kmp\_candy}\ s\ t\ 0\ 0\ pi = \texttt{None}$$

The theorem tries to prove that there can't be a lower upper bound then `2 * length s` as proven in 3.3.2. The theorem was proven by inspecting the case of searching for the empty string in an empty string.

KMP time complexity below lower bound

$$\forall\ s\ t\ pi\ candy.\ \texttt{pi\_correct}\ t\ pi \rightarrow t \neq \lambda \rightarrow |t| <= |s| \rightarrow candy < 1 \rightarrow \texttt{kmp\_candy}\ s\ t\ 0\ 0\ candy\ pi = \texttt{None}$$

The theorem is trivially true, as $candy < 1$ implies that $candy = 0$. The algorithm can never run with $candy = 0$.

KMP time complexity at lower bound

$$\forall\ pi.\ \exists\ t\ s.\ \texttt{kmp\_candy}\ s\ t\ 0\ 0\ 1\ pi \neq \texttt{None}$$

The theorem tries to argue with case the lower bound actually works. The case of searching a single character in a bigger string, where the first character matches was considered. The prove was done by basic simplification.

3.4 KMP Table

3.4.1 KMP Table implementation

When implementing KMP Table we had the same problem as for the KMP algorithm, in that there is no constantly decreasing element. We therefore chose to use the candy variant as the base variant, where the function is fixed on candy. The prefix table algorithm is implemented as a tail recursive function.

```
Fixpoint prefix (t: string) (j i: nat) (pi : list nat) (candy : nat):
    ↪ option (list nat) :=
  match candy with
    | 0 ⇒  None
    | S candy ⇒  if ge_dec i (String.length(t)) then Some(pi) else
      if((0 <? j) && negb(char_at_eq t j i)) then
        prefix t (nth (j-1) pi 0) i pi (candy)
      else if(char_at_eq t j i) then
          prefix t (j+1) (i+1) (app pi [j+1]) (candy)
        else prefix t j (i+1) (app pi [j]) (candy)
    end.
```

3.4.2 KMP Table theorems

To prove the correctness of the KMP table, we have to prove that it returns the correct table on all possible strings `t`. For complexity we follow the definition which states that the time complexity of the prefix function is $\mathcal{O}(|t|)$, in practice the best case is $(|t| - 1) + 1$ and the worst case is $2 * |t|$.

KMP Table correctness

We have defined the following theorem as a proof of correctness of the prefix function:

$$\forall\ t\ c\ j\ i.\ i = 1 \rightarrow j = 0 \rightarrow c > 2 * |t| \rightarrow \texttt{pi\_correct}\ t\ (\texttt{pi\_theorem}\ t\ j\ i\ [0]\ c)$$

This theorem was proven using the following lemma.

$$\forall\ t\ c\ j\ i\ pi.\ c > 2 * (|t| - (i - 1)) \rightarrow j < i \rightarrow i = |pi| \rightarrow$$
$$\texttt{pi\_correct}\ (\texttt{substring}\ 0\ i\ t)\ pi \rightarrow \texttt{pi\_correct}\ t\ (\texttt{pi\_theorem}\ t\ j\ i\ pi\ c)$$

This lemma have not been proven in Coq, but we argue it is correct nonetheless. Since we assume `pi` is correct for the string until `i` then it has always been made starting with $i = 1$ and $j = 0$, backtracking through the proof shows this is true. This also validates $i = |pi|$. $j < i$ always holds, since `j` starts smaller than `i` and we only increase `j` in a case where we also increase `i`.

$c > 2 * (|t| - (i - 1))$ holds since at the first step $i = 1$, we are above upper bound, and since we take i steps this is always enough for the rest of the string. Since the algorithm is fixed on `c` we would do induction on this. The base case is trivial due to the definition of upper bound for the KMP Table.

In the inductive case we would do case analysis for each branch and unwrap `pi_theorem` in the inductive hypothesis. In the second and third branch applying the induction hypothesis would be straight forward due to the straightforwardness of the recursive call, the definitions of the bounds and the quantification of the variables. The first branch would be more tricky, but since we know that `look_up` $(j - 1)\ pi < j$, by definition, we know that $j < i$ would still be true and we could therefore apply the induction hypothesis, concluding the proof on all branches.

KMP Table time complexity above upper bound

The theorem for above upper bound complexity is as follows:

$$\forall\ t\ c.\ c > 2 * |t| \rightarrow \texttt{pi\_candy\_prop}\ t\ c\ 0\ 1$$

This lemma have not been proven in Coq, but we argue it is correct nonetheless. The second and third branch the algorithm both increments `i`, therefore we take at most $(|t| - 1)$ steps in those branches. The first branch always decrements `j` as by definition `look_up` $(j - 1)\ pi < j$. Since we at most increase `j` by one and the number of decreases is at most the number of increases, the first branch takes at most $|t| - 1$ steps. Including the step that checks that we are done and the step that "breaks out" of the first branch, we reach a total number of steps $((|t| - 1) + 1) + ((|t| - 1) + 1) = 2 * |t|$ in the worst case.

KMP Table time complexity at upper bound

The theorem for upper bound complexity is as follows:

$$\forall\ c.\ \exists\ t.\ c = 2 * |t| \rightarrow \neg\ \texttt{pi\_candy\_prop}\ t\ c\ 0\ 1$$

This theorem was proven by inspecting the case where `t` is the empty string, which would give 0 candy. If the algorithm takes no candy, it never runs and therefore never produces a result, and as such is defined as having not terminated.

KMP Table time complexity below lower bound

The theorem for below lower bound complexity is as follows:

$$\forall\ t\ c.\ c < (|t| - 1) + 1 \rightarrow \neg\ \texttt{pi\_candy\_prop}\ t\ c\ 0\ 1$$

This theorem have not been proven in Coq, but we argue that it is correct nonetheless. The best case scenario for the algorithm is where `t` is either a string where all prefixes are suffixes, such as "AAAAAAA", or where there are no prefixes that are suffixes, such as "ABCDEFG".

In these cases the algorithm will in all iteration reach either branch two or three respectively, where `i` is incremented and `j` is either never incremented or incremented in every iteration. Since `i` is incremented in every iteration, and we terminate once `i` is larger or equal to the length of the string, we will at least use the number of steps equal to the length of the string minus 1, as `i` starts at 1, plus 1, since we need a final step to reach the check that returns `Some`. We will never hit the first branch as either `j` is 0 or the `i`th and `j`th elements are equal in `t`.

KMP Table time complexity at lower bound

The theorem for lower bound complexity is as follows:

$$\forall\ c.\ \exists\ t.\ c = (|t| - 1) + 1 \rightarrow \texttt{pi\_candy\_prop}\ t\ c\ 0\ 1$$

This theorem was proven by inspecting the case where `t` is the empty string, which would give 1 candy. If we build a table for the empty string, it is trivial that taking one step would hit the branch where `i` is larger or equal to the length empty string, as `i` is at least 1, and thus terminate.

4 | Discussion

In this chapter we will discuss some of the major revelations we had throughout the project.

4.1 The Complexities of Proving String Algorithms

In common programming environments strings are represented as arrays of characters and algorithms on these tend to access the various indices. Although this may be done in a structured fashion, they do not necessarily advance the algorithm one strictly defined step at a time. If they went forward in a strictly inductive fashion, they would run much slower, which can be seen in our naive implementation which does exactly that.

Inductively defined algorithms rely on some constantly decreasing element, which may then not be present in various string algorithms. This was the cause of most of the problems which we faced when proving KMP properties, as discussed in the following.

From our implementation section it is apparent that Naive and KMP differ a lot in regards to their inductive definition, and their proofs. The Naive algorithm approaches string search inductively, in that every iteration advances the source string by one. Other variables such as the current index, and the iterations are also advanced by one, but it remains the source string that ultimately is the deciding factor. Assuming the algorithm is not in the terminal state, each iteration of Naive can thus be described as follows: **Naive** (`String` $a\ s$) $t\ m \rightarrow$ **Naive** $s\ t\ (m+1)$.

The KMP algorithm approaches string search very differently, as there are no single non-terminating branch. Furthermore, information is not discarded in the same way, so that instead of reducing the source string, the algorithm maintains and updates a state for each iteration. This gives ud several ways of expressing the same state, such as **KMP** (`String` $a\ s$) $t\ 1\ 0\ pi =$ **KMP** $s\ t\ 0\ 0\ pi$.

Due to the state-based nature of the KMP algorithm, it is hard to do induction in the same way as we did in the Naive implementation. When looking at the inductive case in Naive, only the source string will be altered, it will have an additional character, as shown above. It is true that this might change the outcome, but it does not change the other variables. In KMP however, the KMP Table however could be drastically different in this case, and thus it is not as straight forward to prove.

4.1.1 Odd recursions and time complexity

The time complexity of KMP is linear, even though it iterates over two arguments; the source string and the target string. This goes to show that the algorithm skips some iterations that are usually expected, namely the repeated checks of indices of the source string, which is another witness that the algorithm does not approach the problem as straight forward and linear as the Naive implementation.

This was also a root of some of the problems we faced, when proving time complexity of the algorithms; Naive's remaining iterations corresponds directly to the size of the source string, making it easier to prove, while KMP skips redoing comparisons, making it hard to argue about the remaining iterations based solely on the input for a given recursion. This gave us no straight forward way to use induction to prove the theorems, as a failed comparison could potentially mean very little.

4.2 Using Iterations as Decreasing Element

In section 3 it was explained how we altered the definition of some of the algorithms to use the iteration limiter as the decreasing element.

From our proofs on time complexity of Naive, we can see that it is still relatively straight forward to prove the algorithm, even though we introduce the decreasing limitation on iterations. It would thus seem from this that having a non-relevant decreasing element does not affect the difficulty of the proof greatly, however, when it is the only definite decreasing element in the algorithm, it becomes much trickier to proof anything. This is made evident in our failure to prove parts of the KMP algorithm, and the reasons for this.

This means that introducing iterations as the decreasing element in Naive would not make the proofs much harder, as the nature of the algorithm remains the same, but the nature of KMP is simply difficult to prove in general, as discussed in section 4.1.

4.3 Algorithms That Depends on Non-Inductive Input

During our proofs, we discovered that proving something that is based on non-inductive input makes the proof more complex. In our case, this was the KMP Table.

For most proofs we assumed that the given KMP Table was correct, however once we do induction or case analysis on the proof, we have to deal with cases in which the correctness of the KMP Table relates to a new input string.

In most proofs, the change in the correctness is directly reflected by the change in input, however in this case, there is no way of knowing how the correct table for the new string should look, based on the prior string.

For the KMP Table this resulted in the need to provide a definition which would hold for every iteration of the function, and then proving the needed case based on this. This was apparent since the target string never changes in any iteration of the function, so while the functions is fixed on the amount of candy, the value of the string is what produces the result.

Therefore when doing proofs for KMP that assumed the correctness of a table for a target string `t`, doing induction or case analysis on `t` alters the assumption of the correct table.

5 | Conclusion

This report elaborates on the analysis of the problem at hand, and specifies various theorems, that collectively proofs the designated properties of both algorithms.

The translation of the algorithms to the Coq proof assistant language has been documented, along with considerations associated with either algorithm.

While we did not manage to prove all of the specified theorems, we did prove most of them. While doing this, we realised the very different approaches needed to prove two so different algorithms, and we managed to elaborate on the problems that made us unable to prove all of them on time. It was mainly the KMP algorithm that gave us trouble, due to its more imperative nature, as it does not rely on a clearly defined decreasing argument, but instead on one artificially introduced as a limiter. Thus, none of the approaches that worked for the naive implementation was of any help.

While we have tried our best at making a concise and proof-friendly definition, we realise that it may be possible to re-define the KMP algorithm to better encapsulate the step taken during each iteration. This may be achieved by using sub-algorithms that encapsulate different cases, such that each recursive call to the main KMP algorithm has some ensured properties.

Albeit the problems we faced, we deem the project a success as we have still managed to gain a high understanding of some of the problems that can be apparent when proving more intricate algorithms, while also deepening our understanding of proof theory in general. We managed to analyse what theorems were needed to prove the properties of the algorithms we wanted to prove, and furthermore, we managed to argue for the correctness of our implementation in the cases we could not prove, whilst still actually proving most of the problems.

All in all, we know that we have a proven, albeit trivial, naive implementation that will always be correct and provide the first index of a target string in a source string. At the same time, we have implemented the non-trivial and more complex KMP algorithm in Coq, and, whilst not proving everything, proven parts of it and argued that the missing parts are also true.

Maybe explicitly stating that we don't definitely argue the correctness of everything?

A | Appendix

A.1 Work Load Distribution

This section elaborates on how the work load was distributed between the members of the group.

- Naive
  - Main Responsible: Jonas Kastberg
  - Support: Simon Malone
- String Match
  - Main responsible: Jonas Kastberg
  - Support: Simon Malone
- KMP
  - Main responsible: Ankush Jindal
  - Support: Jonas Kastberg
- KMP Table
  - Main Responsible: Kasper Henningsen
  - Support: Simon Malone

The author(s) of each individual Definition, Fixpoint, Theorem and Lemma can be seen in the source code, found in appendix B (though this is only guiding, as others may have put in almost equal amounts of work).

A.2 String Match Implementation

This section elaborates on the implementation and proof of the string match algorithm, used by the naive substring search algorithm[9].

A.2.1 String Match implementation

The implementation of the string match algorithm in Coq is defined as seen in figure A.1.

The algorithm is defined as a fixpoint, with the source string `s`, as the decreasing element. For each recursive call, both strings `s` and `t`, are decremented with one character.

[9]See footnote 3

```
Fixpoint string_match (s t:string) : bool :=
  match s,t with
    | EmptyString, EmptyString ⇒ true
    | String a s', EmptyString ⇒ true
    | EmptyString, String b t' ⇒ false
    | String a s', String b t' ⇒ if ascii_dec a b then
↪ string_match s' t' else false
    end.
```

Figure A.1: String Match implementation

The base cases are defined as either (or both) string(s) being the empty string, in which case it is returned whether `t` is the empty string. The general case asserts if the decremented characters match. The algorithm continues if they do, but returns false if they do not.

A.2.2 String Match theorems

In the following, we outline the proofs mentioned in 2.1, 2.2.1 and 3.1.

The theorems and their proofs as defined in Coq, can be found in appendix B.20. The following is an overview of the theorems as defined in Coq, and a short description on how they were proven.

The time complexity for `string_match` is trivial, as it either run once, or it compares each character in `t`. Therefore it is dependent on the length of `t`.

String Match correctness hit

$$\forall\ s\ t.\ \texttt{string\_match}\ s\ t = true \rightarrow \texttt{substring}\ 0\ |t|\ s = t$$

The theorem was proven by applying a single lemma, which states the same as the theorem[10]. The lemma, however, was proven through induction on `s`

String Match correctness miss

$$\forall\ s\ t.\ \texttt{string\_match}\ s\ t = false \rightarrow \texttt{substring}\ 0\ |t|\ s \neq t$$

Once again, we proved this by induction on `s`.

String Match time complexity above upper bound

$$\forall\ s\ t\ c.\ c > |t| \rightarrow \texttt{string\_match\_candy\_prop}\ s\ t\ c$$

The theorem was also proven through induction on the decreasing element `s`.

[10]This construction was opted for, because the proofs of the naive implementation also relied on this lemma. To avoid having to import the file containing this theorem (and thus proving a theorem called `correctness_miss` by applying a lemma called `correctness_miss`) we created it as a lemma and simply proved the correctness via this lemma.

String Match time complexity at upper bound

$$\forall\ s\ c.\ \exists\ t.\ c = |t| \rightarrow \neg\ \texttt{string\_match\_candy\_prop}\ s\ t\ c$$

We proved this by inspecting the case in which `s` = `t`, with induction on `s`.

String Match time complexity at lower bound

$$\forall\ s\ c.\ \exists\ t.\ c = 1 \rightarrow \neg\ \texttt{string\_match\_candy\_prop}\ s\ t\ c$$

The theorem was proven by inspecting the case where `s` is the empty string, and then simplifying the goal until it became trivially $True$, as it terminates immediately, which is within the bound of 1 candy.

String Match time complexity below lower bound

$$\forall\ s\ t\ c.\ c < 1 \rightarrow \texttt{string\_match\_candy\_prop}\ s\ t\ c$$

The theorem is trivially true, as $c < 1$ implies that $c = 0$. The algorithm will never even look at the strings when given $c = 0$.

String Match candy correspondence

$$\forall\ s\ t\ c\ b.\ c > |t| \rightarrow \texttt{string\_match\_candy}\ s\ t\ c = \texttt{Some}\ b \iff \texttt{string\_match}\ s\ t = b$$

The theorem was proven through induction on `s`, for both implications. Each base case destruct each of the inputs, and observes that each algorithm resolves in the same result.

The inductive steps were solved in the same fashion, using the inductive step after taking one step with either algorithm.

A.3 Unproven Lemmas

Some of the proven theorems were based on unproven lemmas as described previously.

The list of theorems and their unproven lemmas are as follows:

- KMP Table Correctness
  - pi_correctness_recursive
- KMP_Correctness Hit
  - kmp_decr_idx
  - kmp_first_char_diff_no_0
  - kmp_decr_idx_0_same
- KMP Correctness Miss
  - kmp_match_none_implies_string_match_false
- KMP Correctness First
  - kmp_match_result_less_than_length

  - kmp_match_if_empty_then_none
  - kmp_match_n_then_first_differ
  - pi_correct_if_pi_nill
  - pi_inc_awkward
- KMP Time Complexity Above Upper
  - kmp_candies_enough_when_equal_length
  - kmp_recur_candy_inc_no_strong_no_pi

B | Source Code

B.1 AsciiLemmas.v

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.

(* If ascii_dec of two instances of the same ascii is called, the result is b *)
(* Jonas Kastberg *)
Lemma ascii_same_branch : forall (a : ascii) (b c :bool),
  (if ascii_dec a a then b else c) = b.
Proof.
  intros.
  destruct ascii_dec.
  + reflexivity.
  + destruct n.
    * reflexivity.
Qed.

(* ascii neq is commutative *)
(* Jonas Kastberg *)
Lemma ascii_neq_comm : forall ( b c : ascii),
  c <> b ↔ b <> c.
Proof.
  intros.
  split;
  intros;
  unfold not in *;
  intros;
  apply H;
  subst;
  reflexivity.
Qed.
```

B.2 BoolLemmas.v

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.

(* Equality of bools is commutative *)
(* Jonas Kastberg *)
Lemma eq_comm : forall (b c : bool),
  c = b ↔ b = c.
Proof.
  intros.
  split; intros; rewrite H; reflexivity.
Qed.

(* Inequality of bools is commutative *)
(* Jonas Kastberg *)
Lemma neq_comm : forall ( b c : bool),
  c <> b ↔ b <> c.
Proof.
  intros.
  split; intros; unfold not in *; intros; apply H; subst; reflexivity.
Qed.

(* bool true is the same as bool not false *)
(* Jonas Kastberg *)
Lemma true_not_false : forall (b : bool),
  b = true ↔ b <> false.
Proof.
  intros.
  split.
  - unfold not.
```

```
    intros.
   rewrite H in H0.
   inversion H0.
  - intros.
    destruct b.
    + reflexivity.
    + unfold not in H.
      contradiction H.
      reflexivity.
Qed.

(* bool not true is the same as bool false *)
(* Jonas Kastberg *)
Lemma not_true_false : forall (b : bool),
  b <> true ↔ b = false.
Proof.
  intros.
  split.
  - intros.
    destruct b.
    + unfold not in H. contradiction H. reflexivity.
    + reflexivity.
  - unfold not.
    intros.
   rewrite H in H0.
   inversion H0.
Qed.
```

B.3 KMPLemmas.v

```
Require Import KMPStringSearch.
Require Import KMPTable.
Require Import KMPTableTheorems.
Require Import BoolLemmas.
Require Import AsciiLemmas.
Require Import NaiveStringSearch.
Require Import StringExtensions.
Require Import StringMatchLemmas.
Require Import SubstringLemmas.
Require Import StringLemmas.
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Compare_dec.
Require Import Coq.Arith.PeanoNat.
Require Import Coq.omega.Omega.

(* For correct amount of candies and right pi, searching any string t in string s will
     ↪ return a nat smaller than length of s *)
(* Ankush Jindal *)
Lemma kmp_match_result_less_than_length : forall (s t : string) (pi : list nat) (c n : nat),
  c > (2*(length s)) →
  pi_correct t pi →
  kmp_candy s t 0 0 c pi = Some(Some(n)) →
  n < length(s).
Proof.
Admitted.

(* If you search emptystring in string s, the output will be none *)
(* Ankush Jindal *)
Lemma kmp_match_if_empty_then_none : forall (s : string) (pi : list nat) (c : nat),
  c > (2*(length s)) →
  kmp_candy s "" 0 0 c pi = None.
Proof.
Admitted.

(* For correct amount of candies and right pi, if searching any string t in a string with
     ↪ first character as 'a' returns a nat larger than 0, then the first character of two
     ↪ strings differ *)
(* Ankush Jindal*)
Lemma kmp_match_n_then_first_differ : forall (s t : string) (a : ascii) (pi : list nat) (c
     ↪ n: nat),
  c > (2*(length (String a s))) →
```

```
39    pi_correct t pi →
40    kmp_candy (String a s) t 0 0 c pi = Some(Some(S n)) →
41    (Some a) <> (get 0 t).
42  Proof.
43  Admitted.
44
45  (* If a search returns None, then for all i, the string_match on string s from i and t
        ↪ returns false *)
46  (* Ankush Jindal *)
47  Lemma kmp_match_none_implies_string_match_false : forall (s t : string) (c: nat) (pi : list
        ↪ nat),
48    pi_correct t pi →
49    kmp_candy s t 0 0 c pi = None →
50    forall (i:nat), string_match (string_from s i) t = false.
51  Proof.
52  Admitted.
53
54  (* If kmp finds an index at non-zero n, querying with a source string that has removed the
        ↪ first letter will result in n-1 *)
55  (* Jonas Kastberg *)
56  Lemma kmp_decr_idx : forall (s t : string) (a:ascii) (c :nat) (pi : list nat) (n : nat),
57   c > (2*(length (String a s))) → n > 0 → pi_correct t pi → kmp_candy (String a s) t 0
        ↪ 0 c pi = Some (Some (n)) → kmp_candy s t 0 0 c pi = Some (Some (n-1)).
58  Proof.
59  Admitted.
60
61  (* If the first character of source and target string are different, kmp can never find the
        ↪ target substring at the first index *)
62  (* Jonas Kastberg *)
63  Lemma kmp_first_char_diff_no_0 : forall (s t : string) (a a0:ascii) (c :nat) (pi : list
        ↪ nat),
64   c > (2*(length (String a0 s))) → a <> a0 → pi_correct (String a t) pi → kmp_candy
        ↪ (String a0 s) (String a t) 0 0 c pi <> Some (Some (0)).
65  Proof.
66    Admitted.
67
68  (* If a substring is found at index 0, removed the first character from source and target
        ↪ string, also results in index 0  *)
69  (* Jonas Kastberg *)
70  Lemma kmp_decr_idx_0_same : forall (s t : string) (a :ascii) (c :nat) (pi pi' : list nat),
71    c > (2*(length (String a s))) → pi_correct (String a t) pi → pi_correct t pi' →
        ↪ kmp_candy (String a s) (String a t)  0 0 c pi = Some (Some (0)) → kmp_candy (s)
        ↪ (t) 0 0 c pi' = Some (Some (0)).
72  Proof.
73  Admitted.
74
75  (* for for any valid pi, if kmp_candy terminated succesfully for string s given candy,
        ↪ kmp_candy will terminate succesfully for string one larger than s and candy two
        ↪ larger than previous one *)
76  (* Ankush Jindal *)
77  Lemma kmp_candy_inc : forall (s t : string) (pi : list nat) (candy : nat) (a : ascii) (j:
        ↪ nat),
78    j < length t →
79    (look_up j pi) >= 0 →
80    t <> EmptyString →
81    length t <= length (String a s) →
82    kmp_candy s t 0 0 candy pi <> None →
83    kmp_candy (String a s) t 0 0 (S(S(candy))) pi <> None.
84  Proof.
85  intros.
86  induction s.
87  - unfold kmp_candy.
88    destruct (0+0<?length(String a "")).
89    + replace (0+0) with 0 by omega. replace (0+1) with 1 by omega.
90      replace (0+1) with 1 by omega. replace (1+0) with 1 by omega.
91      apply length_empty_string in H1. simpl in H2. inversion H2.
92       * (* length t is 1 *)
93         destruct (string_dec (substring 0 (length t) t) (substring 0 (length t) (String a
        ↪ ""))).
94        ** (* t character is equal to s character *)
95           simpl. rewrite H5. simpl.
96           unfold not. intros. inversion H4.
97        ** (* t character is not equal to s character *)
98           simpl. rewrite H5. simpl.
```

```
99            unfold not. intros. inversion H4.
100         * inversion H5. omega.
101     + unfold not. intros. inversion H4.
102   - unfold kmp_candy.
103     replace (0+0) with 0 by omega. simpl.
104     apply length_empty_string in H1.
105     simpl in H2.
106     destruct (string_dec (substring 0 1 t) (String a "")).
107     + (* first char of t is first char of bigger string (= a) *)
108       inversion H2. (* on length of t *)
109       * (* length w is equal to S S length s -> not in inductive hypothesis *)
110         destruct (length t).
111         ** omega.
112         ** destruct n.
113            *** unfold not. intros. inversion H4.
114            *** (* case of matching second character of t with second character of string *)
115                (* when t is bigger then hypothesis *)
116                admit. (* is this case similar to inductive hypothesis ?~ *)
117       * (* length t is equal or less than to S length s -> as in inductive hypothesis *)
118         destruct (length t).
119         ** omega.
120         ** destruct n.
121            *** unfold not. intros. inversion H6.
122            *** (* case of matching second character of t with second character of string *)
123                (* when t is same as hypothesis *)
124                admit. (* is this case similar to inductive hypothesis ?~ *)
125     + (* first char of t is not first char of bigger string (<> a) *)
126       rewrite substring_0_0_s_is_empty.
127       destruct (string_dec (substring 0 1 t) (String a0 "")).
128       * (* first character of t matches with second character of s *)
129         destruct (length t).
130           ** omega.
131           ** destruct n0.
132              *** unfold not. intros. inversion H4.
133              *** admit.
134       * (* first character of t does not matches with second character of s *)
135         admit. (* recursive case *)
136   Admitted.
137
138   (* Case when t and s are of same length then, 2*length(t) are enough candies - special
          ↪ case, since there is no position to shift m in the implementation *)
139   (* Ankush Jindal *)
140   Lemma kmp_candies_enough_when_equal_length : forall (s t : string) (pi : list nat) (candy:
          ↪ nat),
141     (length t) = (length s) ->
142     candy > (2*length(t)) ->
143     kmp_candy s t 0 0 candy pi <> None.
144   Proof.
145   Admitted.
146
147   (* kmp will never terminate if it is given 0 candy *)
148   (* Ankush Jindal *)
149   Lemma kmp_candy_zero_when_not_emptystring_is_none : forall (s t : string) (pi : list nat)
          ↪ (candy : nat),
150     pi_correct t pi ->
151     t <> EmptyString ->
152     length t <= length s ->
153     candy = 0 ->
154     kmp_candy s t 0 0 candy pi = None.
155   Proof.
156    intros. unfold kmp_candy. subst. reflexivity.
157   Qed.
```

B.4 KMPStringSearch.v

```
1   Require Import KMPTable.
2   Require Import Coq.Strings.String.
3   Require Import Coq.Strings.Ascii.
4   Require Import Coq.Arith.Compare_dec.
5   Require Import Coq.Arith.PeanoNat.
6   Require Import Coq.omega.Omega.
7
```

```
(** KMP_candy implementation**)
(* Implementing a recursive definition of KMP with the use of candy and pi as a list *)
(* Ankush Jindal *)
Fixpoint kmp_candy (s t : string) (m i candy : nat) (pi : list nat) :=
  match candy with
  | 0 ⇒  None
  | S candy' ⇒
    if (m+i) <? (length s)
      then
         if string_dec (substring i 1 t) (substring (m+i) 1 s)
         then
            if (i+1 =? length t)
            then Some(Some(m))
            else kmp_candy s t m (i+1) candy' pi
         else
            if i =? 0
            then kmp_candy s t (m+i+1) 0 candy' pi
            else kmp_candy s t (m+i-(look_up i pi)) (look_up i pi) candy' pi
      else Some(None)
  end.

(* A wrapper around kmp_recur_candy_list to initiate the variables, that results the output
    ↪ *)
(* Ankush Jindal *)
Fixpoint KMP_candy (s t : string) (pi : list nat) (n: nat) :=
  kmp_candy s t 0 0 n pi.

(* A wrapper around KMP_candy, that results False if candies were unsufficient, True
    ↪ otherwise *)
(* Ankush Jindal *)
Fixpoint KMP_candy_prop (s t : string) (pi : list nat) (n: nat) : Prop :=
 match KMP_candy s t pi n with
  | None ⇒  False
  | _ ⇒  True
 end.

(* The worst case for amount of candies required was calculated to be 2*length(s) *)
(* Ankush Jindal *)
Example kmp_test: KMP_candy "11121" "12"  [0;0] (2 * length "11121") = Some(Some(2)).
Proof. reflexivity. Qed.
(** KMP_candy ends **)


(** KMP starts **)
(* This is a wrapper around the whole thing, so as one don't need to initiate with bounded
    ↪ variables *)
(* Ankush Jindal *)
Definition KMP (s t : string) :=
  kmp_candy s t 0 0 (2*(length s)) (pi t).

(* Ankush Jindal *)
Example kmp_test2: KMP "11121" "121" = Some(Some(2)).
Proof. reflexivity. Qed.
(** KMP ends **)
```

B.5 KMPTable.v

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Lists.List.
Require Import Coq.Arith.Compare_dec.
Require Import Coq.Arith.PeanoNat.
Require Import Coq.omega.Omega.
Require Import Coq.Bool.Bool.
Require Import SubstringLemmas.

(* Notation *)
Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation " [ x ] " := (cons x nil).
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

```
16
17  (* Helper functions *)
18  (* Kasper Henningsen *)
19  Definition char_at_eq (t : string) (i j : nat) : bool := if string_dec (substring i 1 t)
        ↪ (substring j 1 t) then true else false.
20
21  (* Kasper Henningsen *)
22  Definition look_up (i : nat) (pi : list nat) :=
23     match i with
24     | 0 ⇒ 0
25     | x ⇒ nth x pi 0
26  end.
27
28  (* Prefix (pi) function *)
29  (* Kasper Henningsen *)
30  Fixpoint prefix (t: string) (j i: nat) (pi : list nat) (candy : nat): option (list nat) :=
31     match candy with
32     | 0 ⇒ None
33     | S candy ⇒ if ge_dec i (String.length(t)) then Some(pi) else
34           if((0 <? j) && negb(char_at_eq t j i)) then
35                 prefix t (nth (j-1) pi 0) i pi (candy)
36           else if(char_at_eq t j i) then
37                  prefix t (j+1) (i+1) (app pi [j+1]) (candy)
38               else prefix t j (i+1) (app pi [j]) (candy)
39      end.
40
41  (* Kasper Henningsen & Simon Malone *)
42  Definition make_prefix (t : string) (candy j i : nat) := prefix t j i [0] candy.
43
44  (* Kasper Henningsen *)
45  Definition pi_theorem (t: string) (j i: nat) (pi : list nat) (candy : nat) :=
46     match prefix t j i pi candy with
47     | None ⇒ []
48      | Some l ⇒ l
49      end.
50
51  (* Kasper Henningsen & Simon Malone *)
52  Definition pi (t : string) :=
53     match make_prefix t (2 * String.length(t)) 0 1 with
54     | None ⇒ []
55     | Some l ⇒ l
56     end.
57
58  (* Kasper Henningsen & Simon Malone *)
59  Definition pi_candy (t : string) (c j i : nat) :=
60     match make_prefix t c 0 1 with
61     | None ⇒ []
62     | Some l ⇒ l
63     end.
64
65  (* Kasper Henningsen & Simon Malone *)
66  Definition pi_candy_prop (t : string) (c j i: nat) : Prop :=
67     match make_prefix t c 0 1 with
68     | None ⇒ False
69     | Some _ ⇒ True
70     end.
71
72  (* Predicates defining correctness of a KMP prefix table *)
73  (* Kasper Henningsen and Simon Malone *)
74  Definition prefix_is_suffix (k m : nat) (t : string) : Prop := (substring 0 m t) =
        ↪ (substring (k-(m-1)) m t).
75  Definition pi_correct (t : string) (pi : list nat) : Prop :=
76     forall (k m : nat), 0 < k → k < String.length(t) → look_up k pi = m →
        ↪ prefix_is_suffix k m t ∧ ∼prefix_is_suffix k (m+1) t.
```

B.6 KMPTableLemmas.v

```
1  Require Import KMPTable.
2  Require Import BoolLemmas.
3  Require Import AsciiLemmas.
4  Require Import NaiveStringSearch.
5  Require Import StringExtensions.
```

```
Require Import StringMatchLemmas.
Require Import SubstringLemmas.
Require Import StringLemmas.
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Compare_dec.
Require Import Coq.Arith.PeanoNat.
Require Import Coq.omega.Omega.
Require Import Coq.Lists.List.

(* Lemma stating that if we have made a correct table so far, and that we have enough candy
     ↪ to continue,
   then the table created by continuing in the string is also correct. *)
(* Kasper Henningsen *)
Lemma pi_correctness_recursive : forall (t : string) (c j i : nat) (pi : list nat),
  c > 2 * (String.length(t) - (i-1)) →
  j < i →
  i = length(pi) →
  pi_correct (substring 0 i t) (pi) →
  pi_correct t (pi_theorem t j i pi c).
Proof.
Admitted.

(* For any string that is not EmptyString, empty list is not a correct pi *)
(* Ankush Jindal *)
Lemma pi_incorrect_if_pi_nill : forall (a0 : ascii) (t : string),
  ~ pi_correct (String a0 t) [ ].
Proof.
unfold not. intros.
Admitted.

(* If for a given string t, you have a pi and appending charater to the string and the new
     ↪ pi is just the old table appended after a nat, the string and the table computed
     ↪ holds special properties *)
(* Ankush Jindal *)
Lemma pi_inc_awkward : forall (t : string) (a0 : ascii) (n0 : nat) (pi : list nat),
  pi_correct t pi →
  pi_correct (String a0 t) (n0::pi) →
  forall (i : nat), i < (String.length t) →
  (n0=0 ∧  (look_up i pi)=0 ∧  Some(a0)<>(get i t)).
Proof.
Admitted.
```

B.7 KMPTableTheorems.v

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Lists.List.
Require Import Coq.Arith.Compare_dec.
Require Import Coq.Arith.PeanoNat.
Require Import Coq.omega.Omega.
Require Import Coq.Bool.Bool.
Require Import SubstringLemmas.
Require Import KMPTable.
Require Import StringLemmas.
Require Import KMPTableLemmas.

(* If we start with i = i, j = 0 and candy above the upper bound,
   creating the prefix table for the entire string t is correct. *)
(* Kasper Henningsen *)
Theorem correctness_pi : forall (t : string) (c j i : nat),
   i = 1 →  j = 0 →
 c > 2 * String.length(t) →
 pi_correct t (pi_theorem t j i [0] c).
Proof.
  intros.
  apply pi_correctness_recursive.
  - rewrite H. replace (String.length(t) - (1 - 1)) with (String.length(t)) by omega. apply
     ↪ H1.
  - omega.
  - simpl. apply H.
  - rewrite H.
```

```
27      unfold pi_correct.
28      intros.
29      destruct t.
30      + rewrite substring_n_m_empty_is_empty in H3.
31        simpl in H3.
32        omega.
33      + rewrite substring_0_1_s_length in H3.
34        * omega.
35        * simpl. omega.
36  Qed.
37
38  (* If we have more candy than twice the length of the string, we always have enough *)
39  (* Kasper Henningsen *)
40  Theorem time_complexity_above_upper : forall (t : string) (c : nat),
41    c > 2 * String.length(t) →
42    pi_candy_prop t c 0 1.
43  Proof.
44  Admitted.
45
46  (* If we have candy equal to the length of the string,
47     there exists a string which does not terminate *)
48  (* Kasper Henningsen *)
49  Theorem time_complexity_at_upper : forall  (c : nat),
50    exists (t:string), c = 2 * String.length(t) →
51    ~pi_candy_prop t c 0 1.
52  Proof.
53    exists EmptyString.
54    intros.
55    subst.
56    unfold pi_candy_prop.
57    simpl.
58    unfold not.
59    intros.
60    contradiction.
61    Qed.
62
63  (* If we never have more than the length of the string, minus one, plus one, we never
        ↪ terminate.
64     The reason for length of t minus one is because the algorithm starts with i = 1, since
        ↪ we do
65     not care about strings of length 1 *)
66  (* Kasper Henningsen *)
67  Theorem time_complexity_below_lower : forall (t : string) (c : nat),
68    c < (String.length(t) - 1) + 1 →
69    ~pi_candy_prop t c 0 1.
70  Proof.
71  Admitted.
72
73  (* If we have exactly the length of the string, minus one, plus one we there exists a t
        ↪ where
74     we terminate. *)
75  (* Kasper Henningsen *)
76  Theorem time_complexity_at_lower : forall (c : nat),
77    exists (t:string), c = (String.length(t) - 1) + 1 →
78    pi_candy_prop t c 0 1.
79  Proof.
80    exists EmptyString.
81    intros.
82    subst.
83    unfold pi_candy_prop.
84    simpl.
85    reflexivity.
86  Qed.
87
88  Compute pi_candy_prop "AAAAAAAB" 10 0 1.
```

B.8 KMPTheorems.v

```
1  Require Import KMPStringSearch.
2  Require Import KMPLemmas.
3  Require Import KMPTable.
4  Require Import KMPTableLemmas.
```

```
5  Require Import KMPTableTheorems.
6  Require Import StringLemmas.
7  Require Import SubstringLemmas.
8  Require Import MiscLemmas.
9  Require Import AsciiLemmas.
10 Require Import StringMatchLemmas.
11 Require Import Omega.
12 Require Import Coq.Bool.Bool.
13 Require Import Coq.Strings.String.
14 Require Import Coq.Strings.Ascii.
15
16
17 (** Proof for correctness *)
18 (* Given ample candies and correct pi, if kmp_candy returns some nat, it must be a
       ↪ substring *)
19 (* Jonas Kastberg *)
20 Theorem correctness_hit : forall (s t : string) (pi : list nat) (c n :nat),
21   c > (2*(length s)) →
22   pi_correct t pi →
23   kmp_candy s t 0 0 c pi = Some(Some(n)) →
24   substring n (length t) s = t.
25 Proof.
26   intros.
27   generalize dependent t.
28   generalize dependent pi.
29   generalize dependent n.
30   induction s; intros.
31   - simpl in *.
32     unfold kmp_candy in H0.
33     destruct c.
34     + simpl in *. inversion H1.
35     + simpl in *. inversion H1.
36   - destruct n.
37     + simpl.
38       destruct t.
39       * reflexivity.
40       * simpl.
41         destruct (ascii_dec a a0).
42         -- subst. simpl.
43            rewrite IHs with 0 (pi_candy t ((2*length t)+1) 0 1) t.
44            ++ reflexivity.
45            ++ simpl in H. omega.
46            ++ apply correctness_pi; omega.
47            ++ apply kmp_decr_idx_0_same with (pi := pi) (pi' := (pi_candy t ((2*length
       ↪ t)+1) 0 1)) in H1.
48               apply H1. apply H. apply H0.
49               apply correctness_pi; omega.
50         -- assert (False).
51            { apply kmp_first_char_diff_no_0 with s t a0 a c pi in H1. apply H1. apply H.
       ↪ apply ascii_neq_comm. apply n. apply H0. }
52            inversion H2.
53     + rewrite ← substring_incr_same by omega.
54       apply IHs with pi.
55       simpl in H. omega.
56       apply H0.
57       apply kmp_decr_idx in H1.
58       apply H1.
59       omega.
60       omega.
61       apply H0.
62 Qed.
63
64 (* Given ample candies and correct pi, if kmp_candy reported no match, then for any index,
       ↪ there mustn't be a match *)
65 (* Ankush Jindal *)
66 Theorem correctness_miss : forall (s t : string) (pi : list nat) (c m :nat),
67   c > (2*(length s)) → pi_correct t pi →
68   kmp_candy s t 0 0 c pi = None →
69   m < length(s)-length(t) →
70   (substring m (length t) s) <> t.
71 Proof.
72   intros.
73   generalize dependent t.
74   generalize dependent pi.
```

```
75    generalize dependent m.
76    induction s.
77    - intros. simpl in *.
78      assert(t<>EmptyString).
79      inversion H2. inversion H2.
80    - unfold not. intros.
81      apply kmp_match_none_implies_string_match_false with (String a s) t c pi m in H1.
82      apply string_match_implies_substring in H1.
83      contradiction. apply H0.
84  Qed.
85
86  (* Given ample candies and correct pi, kmp_candy reutrns the first occurrence of the
        ↪ substring *)
87  (* Ankush Jindal *)
88  Theorem correctness_first : forall (s t : string) (pi : list nat) (c n :nat),
89    c > (2*(length s)) →  pi_correct t pi →
90    kmp_candy s t 0 0 c pi = Some(Some(n)) →
91    forall (i : nat), i < n →
92    substring i (length t) s <> t.
93  Proof.
94    intros.
95    generalize dependent i. generalize dependent pi. generalize dependent c. generalize
        ↪ dependent t. generalize dependent n. induction s.
96    - intros. apply kmp_match_result_less_than_length in H1. simpl in H1. inversion H1.
97      subst. inversion H2. apply H. apply H. apply H0.
98    - unfold not. intros. destruct i.
99      + remember (length t). destruct n0.
100       * destruct t.
101         ** rewrite kmp_match_if_empty_then_none in H1. inversion H1. apply H.
102         ** inversion Heqn0.
103       * destruct n; [inversion H2 | ].
104         apply kmp_match_n_then_first_differ in H1.
105         ** destruct t.
106            *** inversion Heqn0.
107            *** simpl in H1. destruct H1. simpl in H3.
108                apply string_equal_first_char_equal in H3. rewrite H3. reflexivity.
109         ** apply H.
110         ** apply H0.
111     + destruct n; [inversion H2 | ]. destruct t; simpl in *.
112       * rewrite kmp_match_if_empty_then_none in H1. inversion H1. apply H.
113       * destruct pi.
114         ** apply pi_incorrect_if_pi_nill in H0. apply H0.
115         ** eapply pi_inc_awkward in H0.
116            destruct H0. destruct H4.
117  admit.
118  Admitted.
119
120  (** Proof for correctness ends *)
121
122
123  (** Proof for time complexity **)
124  (* A query will always complete when given candy is more than upper bound(2*length(s)),
        ↪ given any valid pi. *)
125  (* Ankush Jindal *)
126  Theorem time_complexity_above_upper : forall (s t: string) (pi : list nat) (candy: nat),
127    t <> EmptyString →
128    length t <= length s →
129    candy > (2*(length s)) →
130    kmp_candy s t 0 0 candy pi <> None.
131  Proof.
132  intros.
133  generalize dependent t. generalize dependent candy.
134  induction s.
135   - simpl. unfold not. intros.
136     apply length_empty_string in H. omega.
137   - rewrite length_inc. intros.
138     inversion H0.
139     + eapply kmp_candies_enough_when_equal_length.
140       apply H3. omega.
141     + rewrite double_inc in H1.
142       (* erewrite kmp_recur_candy_inc_no_strong_no_pi. *)
143       admit.
144  Admitted.
145
```

```
146
147 (* A query will not complete for some string pair when given candy is less than upper
        ↪ bound(2*length(s)), even for correct pi *)
148 (* Ankush Jindal *)
149 Theorem time_complexity_at_upper : forall (c : nat),
150   exists (t s:string) (pi : list nat),
151   c = (2*(length s)) → pi_correct t pi →
152   kmp_candy s t 0 0 c pi = None.
153 Proof.
154  exists EmptyString.
155  exists EmptyString.
156  exists [ ].
157  intros. simpl in *. subst. simpl. reflexivity.
158 Qed.
159
160 (* A query will never complete when given candy less than lower bound(1), even for correct
        ↪ pi  *)
161 (* Ankush Jindal *)
162 Theorem time_complexity_below_lower : forall (s t : string) (pi : list nat) (candy : nat),
163   pi_correct t pi →
164   t <> EmptyString →
165   length t <= length s →
166   candy < 1 →
167   kmp_candy s t 0 0 candy pi = None.
168 Proof.
169   intros.
170   inversion H2.
171   - apply kmp_candy_zero_when_not_emptystring_is_none.
172     apply H. apply H0. apply H1. reflexivity.
173   - omega.
174 Qed.
175
176 (* A query will complete for some string pair when given candy is equal to the lower
        ↪ bound(1), given any valid pi *)
177 (* Ankush Jindal *)
178 Theorem time_complexity_at_lower : forall (pi : list nat),
179   exists (t s:string),
180   kmp_candy s t 0 0 1 pi <> None.
181 Proof.
182   exists (String "a" EmptyString).
183   exists (String "a" (String "b" EmptyString)).
184   intros. simpl.
185   unfold not. intros.
186   inversion H.
187 Qed.
188 (** Proof for time complexity ends *)
```

B.9 KMPWrapper.v

```
1  Extraction Language Ocaml.
2
3  Require Import KMPStringSearch.
4  Require Import ExtrOcamlString.
5  Require Import Ascii String.
6  Extract Inductive ascii ⇒ char
7  [
8  "×( If this appears, you're using Ascii internals. Please don't *) (fun
       ↪ (b0,b1,b2,b3,b4,b5,b6,b7) → let f b i = if b then 1 lsl i else 0 in Char.chr (f b0
       ↪ 0 + f b1 1 + f b2 2 + f b3 3 + f b4 4 + f b5 5 + f b6 6 + f b7 7))"
9  ].
10
11 Extract Constant zero ⇒ "'\000'".
12 Extract Constant one ⇒ "'\001'".
13 Extract Constant shift ⇒
14  "fun b c → Char.chr (((Char.code c) lsl 1) land 255 + if b then 1 else 0)".
15 Extract Inlined Constant ascii_dec ⇒ "(=)".
16
17 Extract Inductive nat ⇒ "int"
18   [ "0" "(fun x → x + 1)" ]
19   "(fun zero succ n →
20       if n=0 then zero () else succ (n-1))".
21
```

```
22
23  Extract Constant plus ⇒  "( + )".
24  Extract Constant mult ⇒  "( * )".
25  Extract Constant minus ⇒   "( - )".
26
27  Extract Inductive sumbool ⇒  "bool" ["true" "false"].
28
29  Extract Inductive list ⇒  "list" [ "[]" "(::)" ].
30
31
32  Extraction "kmp.ml" KMP.
```

B.10 MiscLemmas.v

```
1   Require Import SubstringLemmas.
2   Require Import NaiveStringSearch.
3   Require Import Omega.
4   Require Import StringExtensions.
5   Require Import StringLemmas.
6   Require Import Coq.Bool.Bool.
7   Require Import AsciiLemmas.
8   Require Import BoolLemmas.
9   Require Import Coq.Strings.String.
10  Require Import Coq.Strings.Ascii.
11
12  (* Successor of inequality is logically equivalent to non-successor *)
13  (* Simon Malone *)
14  Lemma lt_eq_smaller : forall (n m : nat),
15    S n <= S m ↔  n <= m.
16  Proof.
17    destruct n, m; omega.
18  Qed.
19
20  (* Successor of lt is logically equivalent to non-successor *)
21  (* Simon Malone *)
22  Lemma lt_smaller : forall (n m : nat),
23    S n < S m ↔  n < m.
24  Proof.
25    destruct n, m; omega.
26  Qed.
27
28  (* Rewriting for multiplication of successor *)
29  (* Simon Malone *)
30  Lemma double_inc : forall (n: nat),
31    (2 * S (n)) = (2 + (2*n)).
32  Proof.
33    intros. omega.
34  Qed.
```

B.11 NaiveLemmas.v

```
1   Require Import SubstringLemmas.
2   Require Import NaiveStringSearch.
3   Require Import Omega.
4   Require Import StringExtensions.
5   Require Import StringLemmas.
6   Require Import StringExtensionLemmas.
7   Require Import Coq.Bool.Bool.
8   Require Import AsciiLemmas.
9   Require Import BoolLemmas.
10  Require Import Coq.Strings.String.
11  Require Import Coq.Strings.Ascii.
12
13  (* If we query the empty string, we always return m *)
14  (* Jonas Kastberg *)
15  Lemma naive_s_empty_is_m : forall (s : string) (m : nat),
16    naive s EmptyString m = Some m.
17  Proof.
18    intros.
```

```
19    induction s.
20    - reflexivity.
21    - reflexivity.
22  Qed.
23
24  (* If we query a non-empty string in an empty string we won't find anything*)
25  (* Jonas Kastberg *)
26  Lemma naive_empty_at_is_none : forall (t : string) (a : ascii) (m : nat),
27    naive EmptyString (String a t) m = None.
28  Proof.
29    intros.
30    reflexivity.
31  Qed.
32
33  (* If looking for a target in an empty string yields None, then the target is not empty *)
34  (* Jonas Kastberg *)
35  Lemma naive_empty_t_is_none_implies_at : forall (t : string) (m : nat),
36    naive EmptyString t m = None → t <> EmptyString.
37  Proof.
38    intros.
39    simpl in *.
40    destruct t.
41    - inversion H.
42    - unfold not.
43      intros.
44      inversion H0.
45  Qed.
46
47  (* A succesful query on an empty string means that m is equal to n *)
48  (* Jonas Kastberg *)
49  Lemma naive_empty_t_is_some : forall (t : string) (m n : nat),
50    naive EmptyString t m = Some n → m=n.
51  Proof.
52    intros.
53    simpl in *.
54    destruct t.
55    - inversion H.
56      omega.
57    - inversion H.
58  Qed.
59
60  (* A successful query on an empty string means that t is empty too *)
61  (* Simon Malone *)
62  Lemma naive_empty_t_is_some_then_t_is_empty : forall (t : string) (m n : nat),
63    naive EmptyString t m = Some n → t = EmptyString.
64  Proof.
65    intros. simpl in *.
66    destruct t.
67    - reflexivity.
68    - inversion H.
69  Qed.
70
71  (* A succesful query means n >= m *)
72  (* Jonas Kastberg *)
73  Lemma naive_boxing : forall (s t : string) ( n m : nat),
74    naive s t m = Some n → n >= m.
75  Proof.
76    intros.
77    generalize dependent t.
78    generalize dependent n.
79    generalize dependent m.
80    induction s.
81    - intros.
82      apply naive_empty_t_is_some in H.
83      omega.
84    - intros.
85      unfold naive in H.
86      remember (string_match (String a s) t) as e.
87      simpl in *.
88      destruct e.
89      + inversion H.
90        omega.
91      + apply IHs in H.
92        omega.
```

```
93  Qed.
94
95  (* If a search returns None then the value of m is irrelevant *)
96  (* Jonas Kastberg *)
97  Lemma naive_none_m_incr_same : forall(s t : string) (m n: nat),
98    naive s t m = None → naive s t n = None.
99  Proof.
100   intros.
101   generalize dependent n.
102   generalize dependent m.
103   generalize dependent s.
104   generalize dependent t.
105   induction s.
106   - intros.
107     simpl in *.
108     destruct t.
109     + simpl in *. inversion H.
110     + simpl in *. reflexivity.
111   - intros.
112     unfold naive in H.
113     remember (string_match (String a s) t) as e.
114     destruct e.
115     + inversion H.
116     + simpl in *.
117       rewrite ← Heqe.
118       apply IHs with (m+1).
119       apply H.
120 Qed.
121
122 (* If a search on a non-empty s returns None, then the search on s returns None as well *)
123 (* Jonas Kastberg *)
124 Lemma naive_at_none_t_none_m : forall (s t : string) (a : ascii) (m : nat),
125   naive (String a s) t m = None → naive s t m = None.
126 Proof.
127   intros.
128   unfold naive in H.
129   destruct (string_match (String a s) t).
130   - inversion H.
131   - apply naive_none_m_incr_same with (m + 1). apply H.
132 Qed.
133
134 (* If a search returns None, then for all i, the string_match on string s from i and t
      ↪ returns false *)
135 (* Jonas Kastberg *)
136 Lemma naive_none_implies_string_match_false : forall (s t : string) (m: nat),
137   naive s t m = None → forall (i:nat), string_match (string_from s i) t = false.
138 Proof.
139   intros.
140   generalize dependent i.
141   generalize dependent t.
142   generalize dependent m.
143   induction s.
144   - intros.
145     assert(t<>EmptyString). { apply naive_empty_t_is_none_implies_at with m. apply H. }
146     simpl.
147     destruct t.
148     + destruct H0. reflexivity.
149     + reflexivity.
150   - intros.
151     destruct i.
152     + unfold naive in H.
153       remember (string_match (String a s) t) as e.
154       destruct e.
155       * inversion H.
156       * simpl in *.
157         apply eq_comm in Heqe.
158         apply Heqe.
159     + replace (S i) with (i+1) by omega.
160       rewrite ← string_from_incr_same.
161       apply IHs with m.
162       apply naive_at_none_t_none_m in H.
163       apply H.
164 Qed.
```

B.12 NaiveStringSearch.v

```
1  Require Import Coq.Strings.String.
2  Require Import Coq.Strings.Ascii.
3  Require Import Coq.Arith.Compare_dec.
4  Require Import Coq.Arith.PeanoNat.
5  Require Import Coq.omega.Omega.
6  Require Import Coq.Arith.Plus.
7  Require Import Coq.Init.Nat.
8  Require Import Coq.Logic.Decidable.
9  Require Import Coq.Bool.Bool.
10
11 (* Check if a target string exists at the beginning of a source string *)
12 (* Jonas Kastberg *)
13 Fixpoint string_match (s t:string) : bool :=
14    match s,t with
15    | EmptyString, EmptyString ⇒ true
16     | String a s', EmptyString ⇒ true
17    | EmptyString, String b t' ⇒ false
18     | String a s', String b t' ⇒ if ascii_dec a b then string_match s' t' else false
19     end.
20
21 (* Find the first index where target string is found in source string, if any *)
22 (* Jonas Kastberg *)
23 Fixpoint naive (s t:string) (n:nat) : option nat :=
24    match s with
25     | EmptyString ⇒ match t with
26                 | EmptyString ⇒ Some n
27                    | _ ⇒ None
28                    end
29     | String c s' ⇒  match string_match s t with
30                      | false ⇒ naive s' t (n+1)
31                      | true ⇒ Some n
32                      end
33     end.
34
35 (* Checks if target string exists in source string *)
36 (* Jonas Kastberg *)
37 Fixpoint naive_prop (s t:string) : Prop :=
38    match naive s t 0 with
39    | None ⇒ False
40    | _ ⇒ True
41       end.
42
43 (* Check if a target string is the prefix of a source string, if there is enough candy,
       ↪ return the result *)
44 (* Jonas Kastberg *)
45 Fixpoint string_match_candy (s t:string) (c : nat) : option (bool) :=
46    match c with
47     | 0 ⇒ None
48     | S c' ⇒ match s,t with
49          | EmptyString, EmptyString ⇒ Some(true)
50           | String a s', EmptyString ⇒ Some(true)
51          | EmptyString, String b t' ⇒ Some(false)
52           | String a s', String b t' ⇒ if ascii_dec a b then string_match_candy s' t' (c')
       ↪ else Some(false)
53          end
54     end.
55
56 (* Check if a string match terminates *)
57 (* Jonas Kastberg *)
58 Definition string_match_candy_prop (s t : string) (c : nat) : Prop :=
59    match string_match_candy s t c with
60    | None ⇒ False
61    | _ ⇒ True
62    end.
63
64 (* Find the first index where target string is found in source string, if any and there is
       ↪ enoough candy *)
65 (* Jonas Kastberg *)
66 Fixpoint naive_candy (s t:string) (n c :nat) : option (option nat) :=
67    match c with
```

```
    | O ⇒ None
    | S c' ⇒ match s with
          | EmptyString ⇒ match t with
                    | EmptyString ⇒ Some (Some n)
                       | _ ⇒ Some None
                       end
           | String c s' ⇒ match string_match s t with
                          | true ⇒ Some(Some(n))
                          | false ⇒ naive_candy s' t (n+1) (c')
                       end
           end
    end.

(* Checks if naive terminates *)
(* Jonas Kastberg *)
Definition naive_candy_prop (s t:string) (m candy:nat) : Prop :=
   match naive_candy s t m candy with
    | None ⇒ False
    | _ ⇒ True
      end.
```

B.13 NaiveTheorems.v

```
Require Import SubstringLemmas.
Require Import NaiveStringSearch.
Require Import Omega.
Require Import Coq.Bool.Bool.
Require Import NaiveLemmas.
Require Import AsciiLemmas.
Require Import StringLemmas.
Require Import StringExtensionLemmas.
Require Import BoolLemmas.
Require Import StringMatchLemmas.
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.

(* A successful query means we can find the substring on the resulting index *)
(* Jonas Kastberg *)
Theorem correctness_hit : forall (s t : string) (n m : nat),
  naive s t m = Some n → (substring (n-m) (length t) s) = t.
Proof.
  intros.
  generalize dependent t.
  generalize dependent m.
  generalize dependent n.
  induction s.
  - intros.
    rewrite substring_n_m_empty_is_empty.
    destruct t.
    + reflexivity.
    + rewrite naive_empty_at_is_none in H.
      * inversion H.
  - intros.
    unfold naive in H.
    remember (string_match (String a s) t) as e.
    simpl in *.
    destruct e.
    + inversion H.
      replace (n-n) with 0 by omega.
      simpl in *.
      destruct t.
      * reflexivity.
      * simpl in *.
        destruct ascii_dec.
        ** simpl in *.
           subst.
         rewrite string_match_substring.
         reflexivity.
           rewrite Heqe.
           reflexivity.
        ** simpl in *.
           subst.
```

```
50              inversion Heqe.
51        + assert(n >= (m+1)). { apply naive_boxing with s t. apply H. }
52          assert(n-m > 0). { omega. }
53           simpl in *.
54          apply IHs in H.
55           replace (n-(m+1)) with (n-m-1) in H by omega.
56           destruct (n-m).
57           * inversion H1.
58           * simpl in *.
59             replace (n0-0) with n0 in H by omega.
60             apply H.
61    Qed.
62
63    (* An unsuccessful query means that no substring of length t within s is equal to t *)
64    (* Jonas Kastberg *)
65    Theorem correctness_miss : forall (s t : string) (m : nat),
66      naive s t m = None → forall (i:nat), i <= length(s)-length(t) → (substring (i) (length
          ↪ t) s) <> t.
67    Proof.
68      intros.
69      apply naive_none_implies_string_match_false with s t m i in H.
70      apply string_match_implies_substring in H.
71      apply H.
72    Qed.
73
74    (* A successful query means that the resulting index is strictly the first index where a
          ↪ substring of length t is equal to t *)
75    (* Simon Malone *)
76    Theorem correctness_first : forall (s t : string) (m n : nat),
77      naive s t m = Some n → forall (i : nat), i < n - m → substring i (length t) s <> t.
78    Proof.
79      intros.
80      generalize dependent i. generalize dependent t. generalize dependent m. generalize
          ↪ dependent n. induction s.
81      - intros. apply naive_empty_t_is_some in H. subst. replace (n - n) with 0 in H0 by omega.
          ↪ inversion H0.
82      - unfold not. intros. destruct i.
83        + destruct n; [inversion H0 | ]. destruct t.
84          * rewrite naive_s_empty_is_m in H. inversion H. subst. replace (S n - S n) with 0 in
          ↪ H0 by omega. inversion H0.
85          * simpl in H. destruct (ascii_dec a a0); subst; simpl in *.
86            -- remember (string_match s t). destruct b; subst; simpl in *.
87               ++ destruct m; [inversion H | ]; simpl in *. inversion H. subst. replace (n - n)
          ↪ with 0 in H0 by omega. inversion H0.
88               ++ apply string_rm_a in H1. apply substring_eq_implies_string_match_true in H1.
89                  rewrite string_from_0_same in H1. apply eq_comm in Heqb. rewrite H1 in Heqb.
          ↪ inversion Heqb.
90            -- inversion H1. contradiction.
91        + destruct n; [inversion H0 | ]. destruct t; simpl in *.
92          * inversion H. subst. replace (n - n) with 0 in H0 by omega. inversion H0.
93          * destruct (ascii_dec a a0); subst; simpl in *.
94            -- remember (string_match s t). destruct b; subst; simpl in *.
95               ++ destruct m; [inversion H; subst; inversion H1 | ]. inversion H. subst.
          ↪ replace (n - n) with 0 in H0 by omega. inversion H0.
96               ++ unfold not in IHs. apply IHs with (S n) (S m) (String a0 t) i.
97                  ** replace (m + 1) with (S m) in H by omega. apply H.
98                  ** destruct m; omega.
99                  ** simpl. apply H1.
100           -- unfold not in IHs. apply IHs with (S n) (S m) (String a0 t) i.
101              ++ replace (m + 1) with (S m) in H by omega. apply H.
102              ++ destruct m; omega.
103              ++ simpl. apply H1.
104   Qed.
105
106   (* A query will always complete when given candy greater than length s *)
107   (* Simon Malone *) (* Jonas Kastberg *)
108   Theorem time_complexity_above_upper : forall (s t : string) (m c : nat),
109     c > length s → naive_candy_prop s t m c.
110   Proof.
111     intros. generalize dependent t. generalize dependent c. generalize dependent m.
112     induction s; intros.
113     - destruct t, c; try (inversion H); try (reflexivity).
114     - unfold naive_candy_prop.
115       unfold naive_candy.
```

```
    destruct c.
    + inversion H.
    + destruct (string_match (String a s) t).
      * reflexivity.
      * apply IHs.
        simpl in H.
        omega.
Qed.

(* A query will always complete when given candy greater than length s *)
(* Jonas Kastberg *)
Theorem time_complexity_at_upper : forall (t : string) (m c : nat),
  exists (s:string), c = length s → not (naive_candy_prop s t m c).
Proof.
  exists EmptyString.
  intros.
  unfold naive_candy_prop.
  subst. simpl.
  unfold not.
  intros.
  contradiction.
Qed.

(* A query will never complete when given candy less than 1 *)
(* Jonas Kastberg *)
Theorem time_complexity_below_lower : forall (s t : string) (m c : nat),
  c < 1 → not (naive_candy_prop s t m c).
Proof.
  intros.
  inversion H.
  - unfold naive_candy_prop.
    unfold naive_candy.
    destruct s; unfold not; intros; contradiction H0.
  - inversion H1.
Qed.

(* A query will never complete when given candy less than 1 *)
(* Jonas Kastberg *)
Theorem time_complexity_at_lower : forall (t : string) (m c : nat),
  exists (s:string), c = 1 → (naive_candy_prop s t m c).
Proof.
  intros.
  exists EmptyString.
  intros.
  subst.
  unfold naive_candy_prop.
  simpl.
  destruct t; simpl; reflexivity.
Qed.

(* If there is enough candy for the algorithm to terminate, then the behaviour of
    ↪ naive_candy and naive is logically equivalent *)
(* Jonas Kastberg *)
Theorem candy_correspondence : forall (s t : string) (m c : nat) (r : option nat),
  c > length s → (naive_candy s t m c = Some(r) ↔ naive s t m = r).
Proof.
  intros.
  split.
  - intros.
    generalize dependent t.
    generalize dependent m.
    generalize dependent c.
    induction s.
    + intros.
      destruct c, t; simpl in *; try ( reflexivity); try (inversion H; inversion H0;
    ↪ reflexivity).
    + intros. destruct c, t; simpl in *.
      * inversion H.
      * inversion H.
      * inversion H0. reflexivity.
      * destruct (ascii_dec a a0).
        -- destruct (string_match s t).
           ++ inversion H0. reflexivity.
           ++ apply IHs with (c).
```

```
188               omega.
189               apply H0.
190         -- apply IHs with c.
191            omega.
192            apply H0.
193    - intros.
194      generalize dependent t.
195      generalize dependent m.
196      generalize dependent c.
197      induction s.
198      + intros. simpl in *.
199        destruct c.
200        * inversion H.
201        * destruct t; inversion H0; reflexivity.
202      + intros. destruct c, t; simpl in *.
203        * inversion H0.
204          inversion H.
205        * inversion H.
206        * subst. reflexivity.
207        * destruct (ascii_dec a a0).
208          -- destruct(string_match s t).
209             ++ subst. reflexivity.
210             ++ apply IHs.
211                omega.
212                apply H0.
213          -- apply IHs.
214             omega.
215             apply H0.
216  Qed.
```

B.14 NaiveWrapper.v

```
1   Extraction Language Ocaml.
2
3   Require Import NaiveStringSearch.
4   Require Import ExtrOcamlString.
5   Require Import Ascii String.
6   Extract Inductive ascii ⇒ char
7   [
8   "×( If this appears, you're using Ascii internals. Please don't *) (fun
        ↪ (b0,b1,b2,b3,b4,b5,b6,b7) → let f b i = if b then 1 lsl i else 0 in Char.chr (f b0
        ↪ 0 + f b1 1 + f b2 2 + f b3 3 + f b4 4 + f b5 5 + f b6 6 + f b7 7))"
9   ].
10
11  Extract Constant zero ⇒ "'\000'".
12  Extract Constant one ⇒ "'\001'".
13  Extract Constant shift ⇒
14   "fun b c → Char.chr (((Char.code c) lsl 1) land 255 + if b then 1 else 0)".
15  Extract Inlined Constant ascii_dec ⇒ "(=)".
16
17  Extract Inductive nat ⇒ "int"
18    [ "0" "(fun x → x + 1)" ]
19    "(fun zero succ n →
20        if n=0 then zero () else succ (n-1))".
21
22
23  Extract Constant plus ⇒ "( + )".
24  Extract Constant mult ⇒ "( * )".
25  Extract Constant minus ⇒  "( - )".
26
27  Extract Inductive sumbool ⇒ "bool" ["true" "false"].
28
29  Extract Inductive list ⇒ "list" [ "[]" "(::)" ].
30
31
32  Extraction "naive.ml" naive.
```

B.15 StringExtensionLemmas.v

```
1   Require Import Coq.Strings.String.
```

```
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Compare_dec.
Require Import Coq.Arith.PeanoNat.
Require Import Coq.omega.Omega.
Require Import Coq.Arith.Plus.
Require Import Coq.Init.Nat.
Require Import Coq.Logic.Decidable.
Require Import Coq.Bool.Bool.
Require Import StringExtensions.
Require Import StringLemmas.

(* The string from 0 is the same as just the string *)
(* Simon Malone *)
Lemma string_from_0_same : forall (s :string),
   string_from s 0 = s.
Proof.
  intros.
  induction s.
  - reflexivity.
  - simpl in *. reflexivity.
Qed.

(* The string from i is the same as the string from i when the string is prepended with an
    ↪ ascii *)
(* Simon Malone *)
Lemma string_from_incr_same: forall (s : string) (a : ascii) (i : nat),
   string_from s i = string_from (String a s) (i + 1).
Proof.
  intros.
  simpl.
  replace (i + 1) with (S i) by omega.
  rewrite gt_branch.
  replace (S i - 1) with i by omega.
  reflexivity.
Qed.
```

B.16 StringExtensionTheorems.v

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Compare_dec.
Require Import Coq.Arith.PeanoNat.
Require Import Coq.omega.Omega.
Require Import Coq.Arith.Plus.
Require Import Coq.Init.Nat.
Require Import Coq.Logic.Decidable.
Require Import Coq.Bool.Bool.
Require Import StringExtensions.
Require Import SubstringLemmas.

(* string_from i on s is the same as a substring on s starting at i with length of s minus
    ↪ i *)
(* Simon Malone *)
Theorem string_from_is_substring : forall (s : string) (i : nat),
  string_from s i = substring i (length s - i) s.
Proof.
  intros. generalize dependent i. induction s; simpl in *; [ destruct i; reflexivity | ];
    ↪ intros. case gt_dec; intros.
  - destruct i; simpl in *; [ inversion g | ]. replace (i - 0) with i by omega. apply IHs.
  - destruct i; simpl in *; [ rewrite substring_from_zero_is_string; reflexivity | ].
    ↪ contradiction n. omega.
Qed.
```

B.17 StringExtensions.v

```
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.omega.Omega.
Require Import Coq.Bool.Bool.
```

```
5
6   (* Take a string from an index *)
7   (* Jonas Kastberg *)
8   Fixpoint string_from (s : string) ( i : nat) : string :=
9     match s with
10    | EmptyString ⇒ EmptyString
11    | String a s' ⇒ if (gt_dec i 0) then (string_from s' (i - 1)) else String a s'
12    end.
```

B.18 StringLemmas.v

```
1   Require Import Coq.Strings.String.
2   Require Import Coq.Strings.Ascii.
3   Require Import Coq.omega.Omega.
4
5   (* The successor of a nat is always greater than 0 *)
6   (* Jonas Kastberg *)
7   Lemma gt_branch : forall (n : nat) (b c : string),
8     (if gt_dec (S n) 0 then b else c) = b.
9   Proof.
10    intros.
11    destruct gt_dec.
12    - reflexivity.
13    - unfold not in n0. assert(False). { apply n0. omega. } contradiction.
14  Qed.
15
16  (* The length of a string prepended with an ascii is the same as the successor of the
        ↪ length of the string *)
17  (* Jonas Kastberg *)
18  Lemma string_length_same : forall (a : ascii) (s : string),
19    length (String a s) = S (length s).
20  Proof.
21    intros.
22    simpl.
23    reflexivity.
24  Qed.
25
26  (* If a string s is equal to another string t then it still is if the same ascii is
        ↪ prepended to both *)
27  (* Jonas Kastberg *)
28  Lemma string_rm_a : forall (a : ascii) (s t : string),
29    s = t ↔ String a s = String a t.
30  Proof.
31    split.
32    - intros.
33      rewrite H.
34      reflexivity.
35    - intros.
36      inversion H.
37      reflexivity.
38  Qed.
39
40
41  (* Equality of strings is commutative *)
42  (* Jonas Kastberg *)
43  Lemma string_eq_comm : forall (s t : string),
44    s = t ↔ t = s.
45  Proof.
46    intros.
47    split.
48    - intros.
49      rewrite H.
50      reflexivity.
51    - intros.
52      rewrite H.
53      reflexivity.
54  Qed.
55
56  (* Non-equality of strings is commutative *)
57  (* Jonas Kastberg *)
58  Lemma string_neq_comm : forall (s t : string),
59    s <> t ↔ t <> s.
```

```
Proof.
  intros.
  split;
    intros;
    unfold not in *;
    intros;
    apply H;
    rewrite H0;
    reflexivity.
Qed.

(* Unknown author *)
Lemma length_inc : forall (s : string) (a : ascii),
  (length (String a s)) = S (length s).
Proof.
intros.
reflexivity.
Qed.

Lemma string_equal_first_char_equal : forall (s t : string) (a a0 : ascii),
  String a s = String a0 t →
  a = a0.
Proof.
  intros.
  inversion H.
  reflexivity.
Qed.

Lemma length_gt_1 : forall (s : string) (a : ascii),
 (length (String a s)) >= 1.
Proof.
 intros.
 simpl.
 omega.
Qed.

Lemma length_empty_string : forall (w : string),
  w <> EmptyString →
  length w <> 0.
Proof.
  intros.
  unfold not in *.
  intros.
  apply H.
  simpl in *.
  destruct w.
  + reflexivity.
  + inversion H0.
Qed.

Lemma length_always_nat : forall (w : string),
  length w >= 0.
Proof.
  intros.
  omega.
Qed.
```

B.19 StringMatchLemmas.v

```
Require Import SubstringLemmas.
Require Import NaiveStringSearch.
Require Import Omega.
Require Import StringExtensions.
Require Import StringLemmas.
Require Import StringExtensionLemmas.
Require Import Coq.Bool.Bool.
Require Import AsciiLemmas.
Require Import BoolLemmas.
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import MiscLemmas.

```

```
(* string_match_candy is logically equivalent to string_match_candy_prop *)
(* Jonas Kastberg *)
Lemma string_match_candy_prop_to_normal : forall (s t : string) (c : nat),
  string_match_candy_prop s t c ↔ exists (v : (bool)), string_match_candy s t c = Some(v).
Proof.
  intros.
  split.
  - intros.
    unfold string_match_candy_prop in H.
    destruct s, t, c; simpl in *; try contradiction; try (exists (true); reflexivity); try
      ↪ (exists (false); reflexivity).
    + destruct (ascii_dec a a0).
      * destruct (string_match_candy s t (c)).
        ** exists b. reflexivity.
        ** contradiction H.
      * exists (false). reflexivity.
  - intros.
    unfold string_match_candy_prop.
     destruct s, t, c; simpl in *; try (inversion H; inversion H0); try (reflexivity).
     + destruct (ascii_dec a a0).
       * destruct (string_match_candy s t (c)).
         **  reflexivity.
         ** inversion H0.
       * reflexivity.
Qed.

(* if string_match_candy_prop is successful with s t c then it will be successful with any
     ↪ ascii appended to s and t and one more candy *)
(* Jonas Kastberg *)
Lemma string_match_candy_prop_step : forall (s t :string) (a b : ascii) (c : nat),
  string_match_candy_prop s t c →  string_match_candy_prop (String a s) (String b t) (S c).
Proof.
  intros.
  unfold string_match_candy_prop.
  unfold string_match_candy.
  destruct (ascii_dec a b).
  - unfold string_match_candy_prop in H.
    unfold string_match_candy in H.
    simpl in *.
    apply H.
  - reflexivity.
Qed.

(* A string_match on s and t that equals true is shows that a substring from 0 to length t
     ↪ on s equals t *)
(* Jonas Kastberg *)
Lemma string_match_substring : forall (s t :string),
  string_match s t = true →  substring 0 (length t) s = t.
Proof.
  intros.
  generalize dependent t.
  induction s.
  - intros.
    simpl in *.
    destruct t.
    + reflexivity.
    + inversion H.
  - intros.
    simpl in *.
    destruct t.
    + reflexivity.
    + simpl in *.
      destruct ascii_dec.
      * subst.
        rewrite IHs.
        reflexivity.
        apply H.
      * inversion H.
Qed.

(* A string match where target is empty is always true *)
(* Jonas Kastberg *)
Lemma string_match_s_empty_true : forall (s : string),
  string_match s EmptyString = true.
```

```
85  Proof.
86    intros.
87    unfold string_match.
88    destruct s.
89    reflexivity.
90    reflexivity.
91  Qed.
92
93  (* Adding the same ascii to the beginning of both s and t does not change the outcome of
        ↪ string_match *)
94  (* Jonas Kastberg *)
95  Lemma string_match_step : forall (s t : string) (a : ascii),
96    string_match (String a s) (String a t) = string_match s t.
97  Proof.
98    intros.
99    unfold string_match.
100   rewrite ascii_same_branch. simpl.
101   reflexivity.
102 Qed.
103
104 (* string_match where target is equal to source is always true *)
105 (* Jonas Kastberg *)
106 Lemma string_match_same : forall (s : string),
107   string_match s s = true.
108 Proof.
109   intros.
110   induction s.
111   - reflexivity.
112   - unfold string_match.
113     destruct ascii_dec.
114     apply IHs.
115     destruct n.
116     reflexivity.
117 Qed.
118
119 (* If a string_match on a source string from i and t equals false then the substring of s,
        ↪ starting at i with length t, is not equal to t *)
120 (* Jonas Kastberg *)
121 Lemma string_match_implies_substring : forall (s t : string) (i : nat),
122   string_match (string_from s i) t = false → substring i (length t) s <> t.
123 Proof.
124   intros.
125   generalize dependent i.
126   generalize dependent t.
127   induction s.
128   - intros.
129     simpl in *.
130     destruct t.
131     + inversion H.
132     + destruct i; unfold not; intros; inversion H0.
133   - intros.
134     unfold not.
135     intros.
136     destruct i.
137     + rewrite string_from_0_same in H.
138       induction t.
139       * rewrite string_match_s_empty_true in H. inversion H.
140       * destruct (ascii_dec a a0).
141         ** subst.
142          rewrite string_match_step in H.
143             apply substring_0_at_as_at_step in H0.
144            assert(substring 0 (length t) s <> t).
145              { apply IHs. rewrite string_from_0_same. apply H. }
146            contradiction.
147         ** assert(substring 0 (length (String a0 t)) (String a s) <> String a0 t).
148              { apply substring_0_a_b. apply n. }
149            contradiction.
150     + assert( substring (i) (length t) (s) <> t ).
151       { apply IHs. rewrite string_from_incr_same with s a i. replace (i+1) with (S i) by
        ↪ omega. apply H.  }
152      rewrite ← substring_incr_same in H0.
153        replace (S i - 1) with i in H0 by omega.
154       contradiction.
155       omega.
```

```
156  Qed.
157
158  (* If a substring of s starting at i with length of t is equal to t then a string_match on
         ↪ the source string from i and t equals true *)
159  (* Simon Malone  *)
160  Lemma substring_eq_implies_string_match_true : forall (s t : string) (i : nat),
161    substring i (length t) s = t →  string_match (string_from s i) t = true.
162  Proof.
163    intros. generalize dependent i. generalize dependent t. induction s.
164    - intros. simpl in *. destruct i; destruct t; try ( reflexivity ); try ( inversion H ).
165    - intros. simpl in *. destruct i.
166      + remember (length t). destruct n; (case gt_dec; [ intros; inversion g | ]).
167        * intros. rewrite ← H. apply string_match_s_empty_true.
168        * intros. simpl in *. destruct t; [ reflexivity | ]. case ascii_dec; intros; simpl in
          ↪ *; subst.
169          -- inversion H. rewrite H1. rewrite ← string_from_0_same with s. apply IHs.
          ↪ inversion Heqn. rewrite ← H2. apply H1.
170          -- inversion H. contradiction.
171      + case gt_dec; intros.
172        * replace (S i - 1) with i by omega. apply IHs. apply H.
173        * contradiction n. omega.
174  Qed.
```

B.20 StringMatchTheorems.v

```
1   Require Import SubstringLemmas.
2   Require Import NaiveStringSearch.
3   Require Import Omega.
4   Require Import NaiveLemmas.
5   Require Import AsciiLemmas.
6   Require Import StringLemmas.
7   Require Import StringMatchLemmas.
8   Require Import Coq.Bool.Bool.
9   Require Import Coq.Strings.String.
10  Require Import Coq.Strings.Ascii.
11
12  (* When string_match s t is true then the substring of s starting at 0 with length t is
        ↪ equal to t *)
13  (* Jonas Kastberg *)
14  Theorem correctness_hit : forall ( s t : string ),
15    string_match s t = true →  substring 0 (length t) s = t.
16  Proof.
17    apply string_match_substring.
18  Qed.
19
20  (* When string_match s t is false then the substring of s starting at 0 with length t is
        ↪ not equal to t *)
21  (* Jonas Kastberg *)
22  Theorem correctness_miss : forall (s t : string),
23    string_match s t = false →  substring 0 (length t) s <> t.
24  Proof.
25     intros.
26     generalize dependent t.
27     induction s.
28     - intros.
29       simpl in *.
30       destruct t.
31       + inversion H.
32       + simpl. unfold not. intros. inversion H0.
33     - intros.
34       simpl in *.
35       destruct t.
36       + inversion H.
37       + simpl in *.
38         destruct (ascii_dec a a0).
39         * rewrite e.
40           unfold not. intros.
41           apply string_rm_a in H0.
42           assert (substring 0 (length t) s <> t).
43             { apply IHs. apply H. }
44           contradiction.
45         * apply substring_0_a_b.
```

```
46          apply n.
47  Qed.
48
49  (* if c is greater than the length of t then string_match_candy_prop will be successful
        ↪ with s t c *)
50  (* Jonas Kastberg *)
51  Theorem time_complexity_above_upper : forall (s t : string) (c : nat),
52    c > length t →  string_match_candy_prop s t c.
53  Proof.
54    intros.
55    generalize dependent t.
56    generalize dependent c.
57    induction s.
58    - intros.
59      destruct c, t; simpl; try ( reflexivity ); try ( inversion H; inversion H0 ).
60    - intros.
61      destruct t, c.
62      + inversion H.
63      + simpl in *. unfold string_match_candy_prop. unfold string_match_candy. reflexivity.
64      + inversion H.
65      + apply string_match_candy_prop_step.
66        apply IHs.
67        simpl in H.
68        omega.
69  Qed.
70
71  (* if c is equal to the length of t then there exists a t for which string_match_candy_prop
        ↪ will not be successful with s t c  *)
72  (* Jonas Kastberg *)
73  Theorem time_complexity_at_upper : forall (s : string) (c : nat),
74    exists (t:string), c = length t →  not (string_match_candy_prop s t c).
75  Proof.
76    intros.
77    exists s.
78    intros.
79    subst.
80    induction s.
81    - unfold not.
82      intros.
83      subst.
84      inversion H.
85    - intros.
86      unfold not in *.
87      intros.
88      apply IHs.
89      unfold string_match_candy_prop in *.
90      unfold string_match_candy in H.
91      destruct (ascii_dec a a).
92      + simpl in *. apply H.
93      + contradiction n. reflexivity.
94  Qed.
95
96  (* If c is less than 1 (lower bound), then string_match_candy_prop will never succeed *)
97  (* Jonas Kastberg *)
98  Theorem time_complexity_below_lower : forall (s t : string) (c : nat),
99    c < 1 →  not (string_match_candy_prop s t c).
100 Proof.
101   intros.
102   inversion H.
103   - unfold string_match_candy_prop.
104     unfold string_match_candy.
105     destruct s.
106     + unfold not. intros. contradiction H0.
107     + unfold not. intros. contradiction H0.
108   - inversion H1.
109 Qed.
110
111 (* If c is equal to 1 (lower bound), then there exists some t, for which
        ↪ string_match_candy_prop will succeed *)
112 (* Jonas Kastberg *)
113 Theorem time_complexity_at_lower : forall (s : string) (c : nat),
114   exists (t:string), c = 1 →  (string_match_candy_prop s t c).
115 Proof.
116   intros.
```

```
117     exists EmptyString.
118     intros.
119     subst.
120     induction s; reflexivity.
121   Qed.
122
123   (* If there is enough candy for the algorithm to terminate, then the behaviour of
          ↪ string_match_candy and string_match is logically equivalent *)
124   (* Jonas Kastberg *)
125   Theorem candy_correspondence : forall ( s t : string) (c : nat) (b: bool),
126     c > length t → (string_match_candy s t c = Some(b) ↔ string_match s t = b).
127   Proof.
128     intros.
129     split.
130     - intros.
131       generalize dependent t.
132       generalize dependent c.
133       induction s.
134       + intros.
135         destruct c, t; simpl in *; try ( reflexivity); try (inversion H; inversion H0;
           ↪ reflexivity).
136       + intros. destruct c, t; simpl in *.
137         * inversion H.
138         * inversion H.
139         * inversion H0. reflexivity.
140         * destruct (ascii_dec a a0).
141           ** apply IHs with c.
142              omega.
143            replace (c-0) with c in H0 by omega.
144              apply H0.
145           ** inversion H0.
146              reflexivity.
147     - intros.
148       generalize dependent t.
149       generalize dependent c.
150       induction s.
151       + intros. simpl in *.
152         destruct c.
153         * inversion H.
154         * destruct t; inversion H0; reflexivity.
155       + intros. destruct c, t; simpl in *.
156         * inversion H0.
157           inversion H.
158         * inversion H.
159         * subst. reflexivity.
160         * destruct (ascii_dec a a0).
161           ** apply IHs.
162              omega.
163            apply H0.
164           ** subst.
165              reflexivity.
166   Qed.
```

B.21 SubstringLemmas.v

```
1   Require Import Coq.Strings.String.
2   Require Import Coq.Strings.Ascii.
3   Require Import Coq.omega.Omega.
4   Require Import StringLemmas.
5
6   (* The substring starting at 0 with length 0 is empty *)
7   (* Jonas Kastberg *)
8   Lemma substring_0_0_s_is_empty : forall (s : string),
9     substring 0 0 s = EmptyString.
10  Proof.
11    intros.
12    induction s.
13    - reflexivity.
14    - simpl.
15      reflexivity.
16  Qed.
17
```

```
(* If s is not empty string, the length of the first char is 1 *)
(* Kasper Henningsen *)
Lemma substring_0_1_s_length : forall (s : string),
   0 < length(s) → length(substring 0 1 s) = 1.
Proof.
  intros.
  induction s.
  - inversion H.
  - simpl. rewrite substring_0_0_s_is_empty. simpl. reflexivity.
Qed.

(* The substring of s starting at 0 with length s is s *)
(* Simon Malone *)
Lemma substring_from_zero_is_string : forall (s : string),
  substring 0 (length s) s = s.
Proof.
  intros. induction s; simpl; [ reflexivity | ]. rewrite IHs. reflexivity.
Qed.

(* The substring of s prepended a starting at 0 with length of t prepended a is equal to t
     ↪ prepended a if and only if substring of s starting at 0 with length t is equal to t
     ↪ *)
(* Jonas Kastberg *)
Lemma substring_0_at_as_at_step : forall (s t : string) (a : ascii),
  substring 0 (length (String a t)) (String a s) = String a t ↔ substring 0 (length t) (s)
     ↪ = t.
Proof.
  split.
  - intros. simpl in *. apply string_rm_a in H. apply H.
  - intros. simpl in *. apply string_rm_a. apply H.
Qed.

(* If a is not equal to a0 then the substring of s prepended a from 0 with length t
     ↪ prepended a0 is not equal to t prepended a0 *)
(* Jonas Kastberg *)
Lemma substring_0_a_b : forall (s t : string) (a a0 : ascii),
  a <> a0 → substring 0 (length (String a0 t)) (String a s) <> String a0 t.
Proof.
  intros.
  simpl in *.
  induction s; unfold not; intros; inversion H0; contradiction.
Qed.

(* Any substring of the empty string is the empty string *)
(* Simon Malone *)
Lemma substring_n_m_empty_is_empty : forall (n m:nat),
  substring n m EmptyString = EmptyString.
Proof.
  intros.
  induction n.
  - simpl. destruct m. reflexivity. reflexivity.
  - simpl. reflexivity.
Qed.

(* Any substring of length 0 is the empty string *)
(* Jonas Kastberg *)
Lemma substring_n_0_s_is_empty : forall (s : string) (n : nat),
  substring n 0 s = EmptyString.
Proof.
  intros. generalize dependent n. induction s.
  - simpl. destruct n.
    + reflexivity.
    + reflexivity.
  - destruct n.
    + reflexivity.
    + apply IHs.
Qed.

(* If n is strictly greater than 0 the the substring of s from n - 1 of length m is the
     ↪ same as the substring of s prepended a starting at n of length m *)
(* Jonas Kastberg *)
Lemma substring_incr_same : forall (s :string) (n m : nat) (a : ascii),
   n > 0 → substring (n - 1) m s = substring n m (String a s).
Proof.
```

```
87    intros.
88    simpl.
89    destruct n.
90    - inversion H.
91    - simpl.
92      replace (n-0) with n by omega.
93      reflexivity.
94  Qed.
```