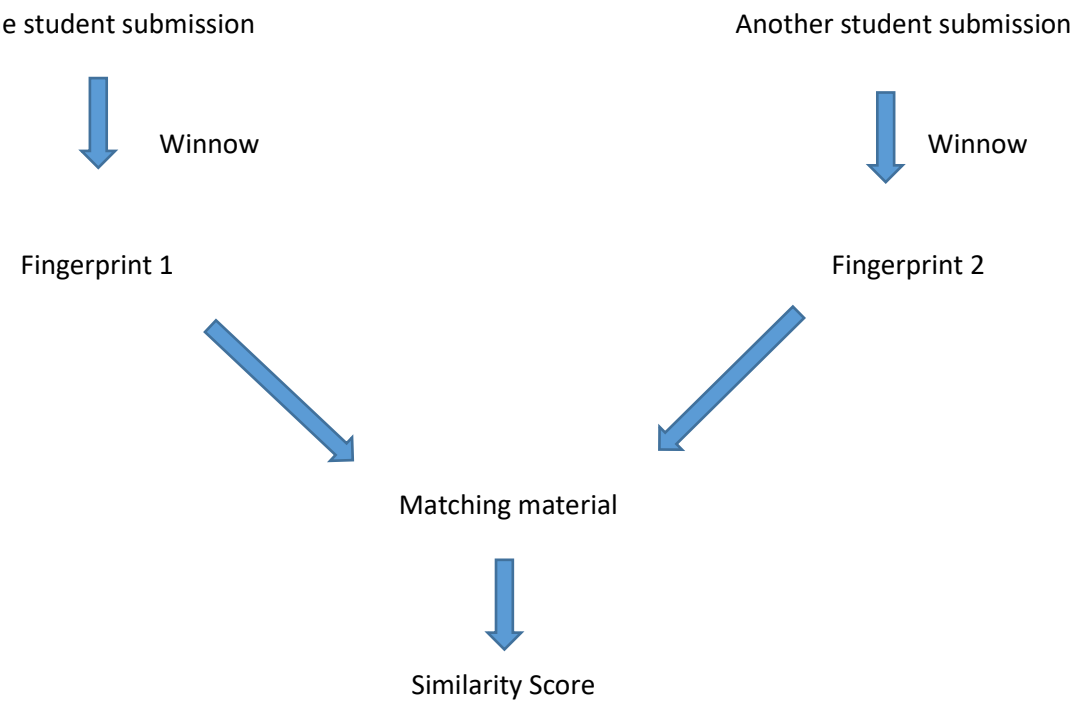
# How to catch a cheater: using winnowing to create document fingerprints

**Project due date:** You will need to upload your code to MATLAB Grader before

**11:59 pm on Saturday the 10th of September**.




NOTE: This is version 1.2 of the project document. If you find a typo, please post it to the Piazza project typo thread. If any significant typos (beyond minor grammar and spelling errors) are found, a newer version of the project will be uploaded to Canvas, with those typos corrected.

# Table of Contents

# Introduction

This project requires you to write code that will compare how similar two documents are, using a technique known as **winnowing**[1] to produce document fingerprints. It is much more efficient to compare the document fingerprints to each other than the documents themselves. Winnowing is the algorithm at the heart of the misconduct detection software we use to catch cheaters[2].

## The problem

Every year some students beg, "borrow", or steal solutions to assessments from others and then copy some (or even all) of the answers they obtained. Submitting work that isn't your own is cheating and unfair to those who have worked hard and done the assessment themselves. It even disadvantages the person cheating, as they miss learning important skills they will need later in their degree/career.

To combat cheating, we need a robust way of detecting it, to identify the students who have cheated and apply appropriate penalties. Penalties could include awarding a failing grade in the course so they can repeat the course and learn the material they need to grasp. This is particularly important in a professional degree such as engineering, where mistakes can literally cost lives.

For small courses, it is relatively simple to compare one student's work against that of every other student in the class simply by having a person look at all the solutions in quick succession. This becomes much more difficult to do at scale when there can be over 1000 students in the class (as is the case for ENGGEN 131). Hence, we need a way to automate the comparison of one student's work with that of others.

It can be computationally expensive to compare an entire document against another one to see if they are the same. When many files and students are involved, the amount of work can become significant, even for a computer. For example, to check the MATLAB project submissions for misconduct would require over 5,000,000 document comparisons. Rather than comparing an entire document against other documents, a more efficient approach is to calculate a **fingerprint** for each document, which identifies key aspects (i.e., substrings) of that document. We can then compare the document fingerprints rather than the documents themselves (which is much faster to do). There is admittedly some work required to generate the fingerprints, but we only need to do this once for each file. The savings can be significant if we are then comparing that fingerprint against 1000 others.

[1] Winnowing is an agricultural term for the process of separating wheat (the bits of the plant you want) from chaff (the bits you don't want).

[2] We use a software package called Measure of Software Similarity (MOSS) developed at Stanford university.

### Document Fingerprinting

Document fingerprinting is used, not only because it allows for faster comparisons but because it has several other important properties, which are highly desirable in any misconduct detection algorithm. These properties are as follows:

1) We want to ignore irrelevant features, such as whitespace or case. For example, we should still detect two documents as similar even if a student has altered the indentation of their code and changed the case of their variables in an attempt to disguise the similarities[3].
2) We don't want to return matches for very short substrings that will be in all solutions, such as the word "function". To do so would drown out true matches in unnecessary noise.
3) We need to identify two solutions as similar, even if chunks of material have been reordered or other material has been added (e.g., a student may have added in different comments but still have chunks of code that are the same). If the same long substring is present in both documents, we want to identify the match, regardless of where those substrings are.

There are various document fingerprinting algorithms. However, we will focus on implementing a particularly elegant method called winnowing, which yields good results.

While we refer to fingerprinting "documents", in practice, we read the contents of a document (i.e., a file) into a string and then work on fingerprinting that string.

## The Winnowing Algorithm

The following overview is based on the explanation provided in the original paper that describes winnowing called *Winnowing: Local Algorithms for Document Fingerprinting* by Schleimer, Wilkerson and Aiken[4]. For the technically minded, you might like to check this paper out for a more in-depth explanation and analysis (Warning: it contains many mathematical formulae and computer science terminology).

Winnowing has two user-specified parameters that control its behaviour:

- A noise threshold $k$, which specifies the minimum substring length we are interested in matching (any substring matches that are shorter than $k$ will be ignored).
- A guarantee threshold $t$, which specifies the minimum substring length for which we are *guaranteed* to detect a match.

Note that we must have $k \leq t$. Any substrings with a length $n$, between $k$ and $t$ might be identified but aren't guaranteed to be. That is, there is no guarantee we will identify matches for a substring of length n where $k \leq n < t$.

The larger $k$ is, the more confident we are that the matches aren't just coincidental. If we set $k$ to be too large, we may miss identifying smaller fragments of code that have been copied.

[3] When comparing code, it is good practice to disregard the variable names they have used so that we can still detect the similarities even if students have changed their variable names. We won't get that sophisticated in this project though (note, however that MOSS **is** that sophisticated – so changing variable names is no defence against detection).

[4] A copy is available on Canvas in the MATLAB Project directory.

Here are the five steps of winnowing

1) Remove irrelevant features,
2) Break up text into a sequence of k-grams (i.e., substrings of length $k$),
3) Calculate hash values for the sequence of k-grams,
4) Create a set of windows from the hash values, each of length $w = t - k + 1$,
5) Create a fingerprint for the document by selecting the rightmost minimum value from each window and saving all such selected values (+ relative positions) as the fingerprint.

We illustrate this algorithm by working through a small example for the following text, taken from a popular 80's song:

```
'Cold, cold, heart.'
```

For this example, we will choose a $k$ value of 5 and a $t$ value of 8 (i.e., $k = 5, t = 8$).

**Step 1: Remove irrelevant features.**

We need to strip out any whitespace and convert any uppercase characters to lowercase ones. In some contexts, it is also useful to strip out the punctuation characters, but for simplicity, we leave them in. Therefore, the stripped string is:

```
'cold,cold,heart.'
```
(Note the lowercase c at the start.)

**Step 2: Break up text into a sequence of k-grams (i.e., substrings of length k).**

We form all possible substrings of length k to create k-grams. For our example, we will work with 5-grams (i.e., all possible substrings of length 5). There are 12 of them. Here they are in order (note it is possible to have the same substring appear more than once (as shown highlighted in red)):

```
'cold,'   'old,c'   'ld,co'   'd,col'   ',cold'   'cold,'
'old,h'   'ld,he'   'd,hea'   ',hear'   'heart'   'eart.'
```

**Step 3: Calculate hash values for the sequence of k-grams.**

A hash converts a substring into a (hopefully unique) integer value. This means we can work with the integer values rather than the original substrings. Here are some hash values for the above k-grams (notice how the two identical substrings (highlighted in red above) have the same hash value).

```
470472   968222   4384     438328   702736   470472
968227   4529     442812   841754   585798   858128
```

**Step 4: Create a set of windows from the hash values, each of length w = t-k+1.**

Our window size is: $w = t - k + 1 = 8 - 5 + 1 = \mathbf{4}$

Recall that our hash values (in order from left to right) are:

```
470472    968222    4384      438328    702736    470472
968227    4529      442812    841754    585798    858128
```

We now form all possible sets of sequential hash values of length $w = 4$, to create a set of windows (this is quite similar to Step 2, but now we are working with hash values instead of characters).

For clarity, I have highlighted in bold the minimum value in each window (as we will need these values in Step 5).

```
470472    968222    4384      438328    % window 1
968222    4384      438328    702736    % window 2
4384      438328    702736    470472    % window 3, etc.
438328    702736    470472    968227
702736    470472    968227    4529
470472    968227    4529      442812
968227    4529      442812    841754
4529      442812    841754    585798
442812    841754    585798    858128
```

**Step 5: Create a fingerprint for the document by selecting the rightmost minimum value from each window and saving all selected values (+ relative positions) as the fingerprint.**

We now select the minimums (highlighted in bold in the previous step), along with their relative position in the list of hash values, to form our fingerprint (and if there is more than one minimum, we choose the **rightmost** one). If the value in a particular position has already been selected, we don't add it again (i.e. each value in a particular position is only added to the fingerprint once).

Here are the selected minimums (shown in bold) and their positional information in the row below (positions of selected minimums are highlighted in red).

```
470472 968222 4384 438328 702736 470472 968227 4529 442812 841754 585798 858128
    1      2    3      4      5      6      7    8      9     10     11     12
```

This means our document fingerprint will be

```
4384      438328    4529      442812
3         4         8         9
```

Once we have a fingerprint for several documents, it is possible to compare the fingerprint of one document against another to see if the documents have any hash values in common. If there are hash values in common, we can use the substring length $k$, and the corresponding positional information for the matches to calculate a similarity score for two documents. This score estimates how much material is shared between the two documents. Two documents with a high similarity score likely come from students copying someone else's work.

**A note on dealing with duplicates**

The previous worked example produced a fingerprint that didn't contain any duplicate hash values. This is not always the case. Consider the following hash value sequence (along with relative positions).

```
77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98 % hash values

 1  2  3  4  5  6  7  8 9 10 11 12 13 14 15 16 17 % relative positions
```

We will form a set of windows from these hash values, using a window length of $w = 4$. The set is shown below, with the rightmost minimum in each window identified in bold.

```
    77    74    42    17  % window 1, minimum 17, from position 4
    74    42    17    98  % window 2, minimum 17, from position 4
    42    17    98    50  % window 3, minimum 17, from position 4
    17    98    50    17  % window 4, minimum 17, from position 7
    98    50    17    98  % window 5, minimum 17, from position 7
    50    17    98     8
    17    98     8    88
    98     8    88    67
     8    88    67    39
    88    67    39    77
    67    39    77    74
    39    77    74    42
    77    74    42    17 % window 13, minimum 17, from position 16
    74    42    17    98 % window 14, minimum 17, from position 16
```

As we have already observed, it is common for a value/position pair to be selected multiple times. For instance, in the above example, the first three windows have the rightmost minimum with a value of 17, which comes from index position 4 in the original hash value sequence. We only add this value/position pair (of 17 in position 4) to the fingerprint once.

It is, however, possible to select the same hash value *but in a different position*, in which case we would still add it to the fingerprint. For example, in window 4, the rightmost value is 17, but it comes from index position 7. Since we have a different value/position pair (i.e., 17 in position 7), this will be added to the fingerprint. The same value/position pair is selected again in window 5 (but remember, we only add it to the fingerprint once).

Below, are the selected minimum hash values (highlighted in bold) along with their associated relative positions (highlighted in red).

```
77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98  % hash values

 1  2  3  4  5  6  7  8 9 10 11 12 13 14 15 16 17  % relative positions
```

Hence the fingerprint for this window is the bold hash values and their associated relative positions are:

```
    17    17     8    39    17
     4     7     9    12    16
```

Notice how 17 occurs three times in our fingerprint (but each occurrence has a different position associated with it).

# Overview of code to write

You will be provided with a function called `Winnow` and a script called `CompareFiles` (which repeatedly calls `Winnow` to find document fingerprints and then identifies how similar files contained within a specified directory are, by comparing the fingerprints). You do not need to modify either of these supplied files (and will not be submitting them).

`Winnow` calls a number of functions that you will write. Once you have written all the specified functions that it calls, you can then use the `Winnow` function to generate a document fingerprint.

Note that `CompareFiles` will only work if you successfully write **all** the specified functions that both it and `Winnow` call. Each function that you write is submitted and marked separately via MATLAB Grader. As every function can be marked independently, it is possible to skip over a function if you are finding it challenging and instead have a go at one of the other functions.

Try and submit as many of the **ten** functions as you can. You don't have to tackle them in any particular order, but the helper functions are a good place to start, as they can be completed using just the concepts from Weeks 1 to 3. Hence a great warm-up.

**Part 1: Helper functions**

Functions to write

| | |
|---|---|
| `Hash31` | Calculates a hash value from an array of values (a string or an array of integers). |
| `RightMin` | Finds the value and position of the rightmost minimum in an array of values. |
| `FindMatchIndices` | Compares two arrays and returns a list of index positions from array 1 of any values that also appear in array 2. |

These three helper functions are intended to make writing some of the other more complicated functions easier, as you can use them to perform one step of a larger task. You will probably find it helpful to call each of these from one of the other functions you write. Note that you do not HAVE to call these helper functions from any of the other functions you write, but you should submit them to earn marks, even if you decide not to call them from your other functions. You should be able to attempt these three functions after you have done Lab 3.

**Part 2: The Winnow algorithm**

Functions to write for the `Winnow` function to work

| | |
|---|---|
| `StripString` | Will strip out whitespace and other unprintable characters, plus convert all uppercase characters to lowercase. |
| `Kgram` | Take a string and divide it into a sequence of k-grams, where each element is $k$ characters long. Returns a cell array where each cell array element is one of the k-grams. |
| `HashList` | Calculates the hash31 value for every string stored in a cell array to generate a sequence of hash values (which will be integers). |
| `Window` | Takes a sequence of integer values and creates a 2D array of windows, where each row of the array is a window containing $w$ integer values. |
| `Fingerprint` | Takes a set of windows (in the form of a 2D array) and calculates a document fingerprint using the winnowing algorithm (i.e., we select the rightmost minimum from each window). Will return the document fingerprint as a 2D array where each column contains a winnowed value and its corresponding position. |

**Part 3: Comparing files**

Functions to write for the `CompareFiles` function to work

| | |
|---|---|
| `FindMatchPositions` | Given two document fingerprints, return two lists of position matches (one for each document). |
| `SimilarityScore` | Calculate a similarity score by taking a list of matched positions and determining the proportion of characters in a string matched. |

In the following pages, you will find detailed explanations of each of the **ten** functions you need to write to compare files using the winnowing algorithm.

This document shows some example calls for each function, but it is expected that you will create your own test values to test your code (in many cases, you can work out the answers on some very small sets of test data by hand, so that you know what to expect as output).

You have a limited number of submission attempts, (**six**) for each function. This means it is important to thoroughly test each function before submission so you don't waste attempts.

This project is designed to take the average student around **15 hours** to code, but experience has shown that some people spend upwards of 60 hours on it (while others finish it in just a few hours). You will not know how long it will take you, so please don't leave it to the last minute to make a start!

## Remember the golden rule

**Write your own code, and don't give your code to anyone!** Friends don't let friends share code. This project needs to be YOUR work; it is an individual project, not a group project. Copying and/or sharing code is academic misconduct. Please note that every project submission is passed through software called MOSS, which detects plagiarism (using the same algorithm you are coding up!). If you copy, **you WILL be caught**. Unfortunately, students discover the hard way every year that I'm not kidding. Let me repeat,

**DO NOT COPY: YOU WILL BE CAUGHT.**

If you give your code to another student and they copy it, you will both be guilty of misconduct. Copying or supplying code typically results in both the person who copied and the person who supplied the code being awarded a mark of zero for the project, as well as your names going on the register of academic misconduct. Suppose you have already been found guilty of academic misconduct on another course or assignment. In that case, the penalty may be even greater (e.g. a fine, grade reduction or failure of the course).

To help you avoid academic misconduct, I have put together a very short best practices guide which you should read.

If you have any queries about what is or isn't academic misconduct, please ask so that I can make sure everyone understands what is acceptable behaviour and what isn't.

## How to tackle the project

Do not be daunted by the length of this document. It is long because a lot of explanation is given for exactly what each function needs to do.

The best way to tackle a big programming task is to break it down into small manageable chunks. A lot of this work has already been done for you in the form of ten functions to write. Each function has a detailed description of what it needs to do, and you can write most of the functions independently of each other (the exception being that you might want to use the helper functions to help you write some of the harder ones). Read through the entire document and then pick a function to start on.

Remember, you don't have to start with `Hash31`, although it is one of the simpler functions to write. You may prefer to start with one of the other straightforward functions such as `RightMin`, `Kgram` or `Window`.

Several functions require careful thought, particularly those that form the heart of the winnowing algorithm. Remember those 5 steps of problem-solving!. If you have trouble understanding how a function should work, remember to work through the problem by hand with a small data set.

Note that I don't necessarily expect everyone to write all the functions. Some are relatively straightforward (e.g., `Hash31` and `RightMin`), while others are a bit trickier (e.g., `Fingerprint`). You can still get a pretty good mark even if you don't complete all the functions.

You will receive marks for functionality (does your code work?) and style (is it well written?). An "A" grade student should be able to nut out almost everything. B and C grade students might not get a fully working solution. Take heart; even if you only get half of the functions working, you can still get over 50% for the project, as long as the code you submit is well written (since you get marks for using good style).

## What MATLAB functions do I need to know?

The entire project can be completed using just the functions we have covered in the course manual, lectures, and labs. Once you have completed Lab 5, you should have all the skills required to tackle all ten tasks.

## What MATLAB functions can I use?

When writing your code, while you don't need to use functions we haven't covered in class, you may wish to do so. As well as being able to use any functions we have covered in class, you can also use any other functions that are part of MATLAB's **core** distribution (i.e., not functions from any toolboxes you may have installed). Note that for this project, ***the use of functions only available in toolboxes is not permitted***. MATLAB features a large number of optional toolboxes that contain extra functions. The purpose of this project is for you to get practice at coding, not to simply call some toolbox functions that do all the hard work for you.

**IMPORTANT**: We have configured MATLAB Grader to only allow the use of **core** functions. Your code won't pass MATLAB Grader tests if you have used toolbox functions that aren't in the core distribution.

You can find a list of some of the core MATLAB functions in the MATLAB command reference appendix at the end of the course manual. If you are uncertain whether a particular function is part of the MATLAB's core distribution or not, type the following at the command line

```
which <function>
```

where the term *<function>* is replaced by the name of the function you are interested in. Check the text displayed to see if the directory directly after the word toolbox is **MATLAB** (which means it is core) or something else (which means it is from a toolbox).

For example:

```
>> which median
```

```
C:\Matlab\R2022a\Pro\toolbox\matlab\datafun\median.m
```

This shows us that the median function is part of the standard core MATLAB distribution (notice how the word *matlab* follows the word toolbox). You may use the median function if you wish to.

```
>> which imadd
```

```
C:\Matlab\R2017b\Pro\toolbox\images\images\imadd.m
```

This shows us that the imadd function is part of the image processing toolbox and is therefore not permitted to be used for the project (notice how the word **images** follows the word toolbox).

## Summary of Functions to Write

There are **ten** in total:

| Function | Difficulty Level | Where used |
|---|---|---|
| `Hash31` | Easy | Likely to be handy when writing your `HashList` function |
| `RightMin` | Easy | Likely to be handy when writing your `Fingerprint` function. |
| `FindMatchIndices` | Medium | Likely to be handy when writing your `FindMatchPositions` function. |
| `StripString` | Easy/Medium | Called from `Winnow` |
| `Kgram` | Easy | Called from `Winnow` |
| `HashList` | Easy | Called from `Winnow` |
| `Window` | Easy | Called from `Winnow` |
| `Fingerprint` | Hard | Called from `Winnow` |
| `FindMatchPositions` | Medium/Hard | Called from `CompareFiles` |
| `SimilarityScore` | Medium/Hard | Called from `CompareFiles` |

The difficulty level column estimates how hard a task will be for a typical student. Don't worry if you find an easy task difficult (or vice-versa); it is just a guideline, and we can't anticipate what bugs you might encounter or how you will approach a particular task.

The pages following have detailed specifications of what each function should do. Remember, while the functions are designed to work together with the supplied code, many functions can be written and tested in isolation. Even if you get stuck on one function, you should still try and write the others.

Keep the following general principles in mind:

- Note the capital letters used in function names. Case matters in MATLAB, and you should take care that your function names EXACTLY match those in the projection specification. (i.e., if the function name is specified as `RightMin` don't use the name `rightmin`, or `rightMin)`.
- When writing functions the input argument's' order, type, and dimension matter. Make sure you have the required number of inputs in the correct order, e.g., if a function requires two inputs, as `Kgram` does (where the first input is the $k$ value and the second is the character array to divide into k-grams), then the order matters (i.e., the first input should be the $k$ value, to match the specifications). The number of inputs also matters (so don't change a function that requires two inputs to require more inputs).
- The type and dimensions of the outputs matter, e.g., `Kgram` returns a 1D cell array where each element of this array is a character array. Returning a 2D array of characters would cause your code to fail.

## The Hash31 function

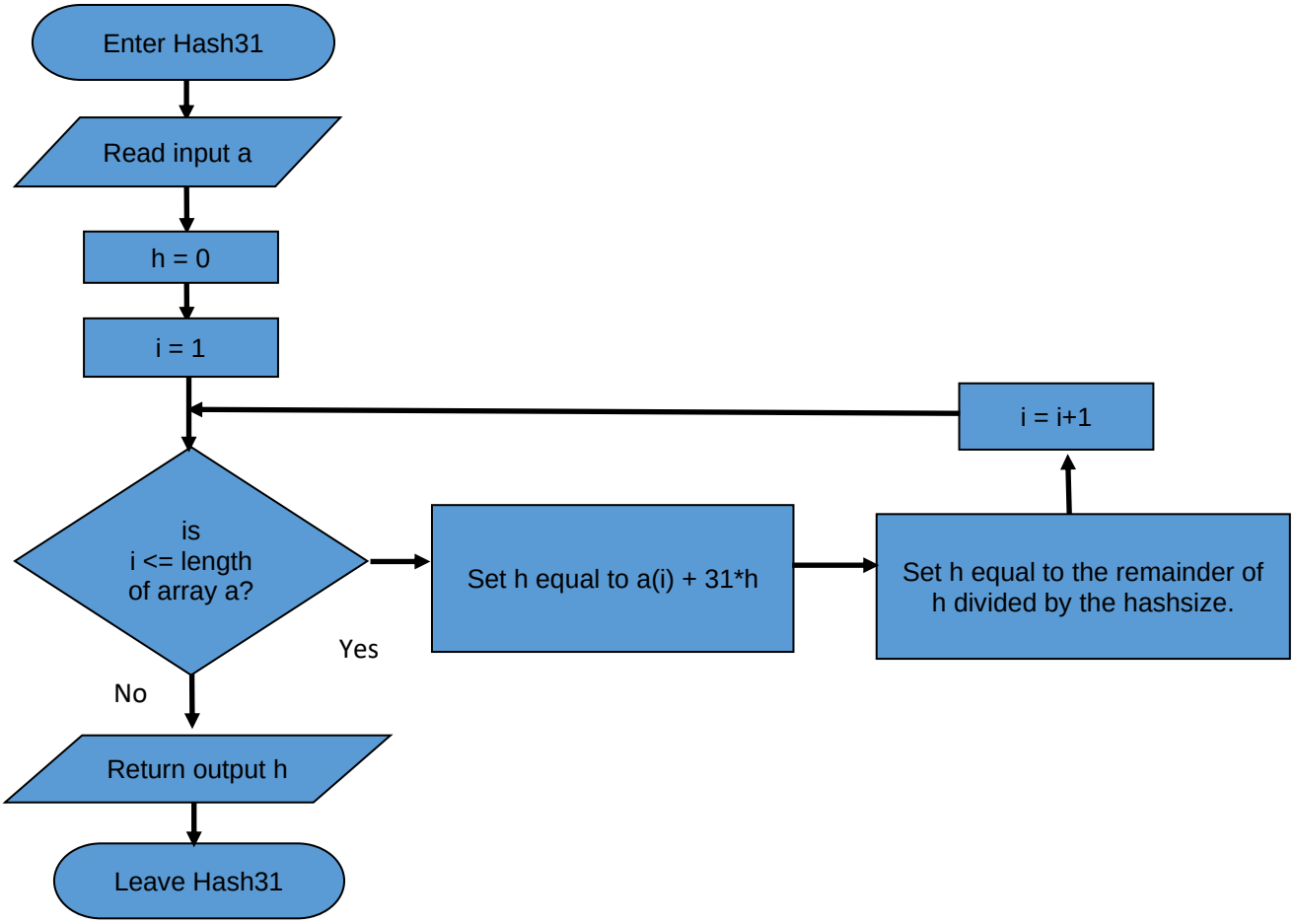
| | |
|---|---|
| Purpose | Calculates a hash value from an array of values |
| Input | A 1D array of values (which could be a character array or an array of integers). It is possible the input could be an array of length zero (i.e. an empty array). |
| Output | An integer value calculated using the hash 31 algorithm |

The Hash31 function is one of the simplest functions to write. It requires you to implement an algorithm to calculate a hash value for a 1D array.

A hashing algorithm takes a sequence of values and converts them into a (hopefully unique) integer value. This means we can then work with the integer value (rather than the original sequence of values). Hashes are very useful and are used a lot in programming.

There are many different hashing algorithms, and this function is based on one of the simplest[5], which is taken from the second edition of the book "The C Programming Language." by Brian Kernighan and Dennis Ritchie[6], who developed the C language.

A flow chart for the algorithm to calculate a hash value **h**, for an array of integer values **a**, is provided below




[5] The hashing algorithm presented here is mathematically equivalent to the one presented in the book but less prone to rounding errors (this comes at the cost of it being more computationally expensive to calculate).
[6] This book is so famous it is sometimes just referred to as K&R, after the authors' surname initials, Kernighan and Ritchie.

The hashsize value sets a limit on the largest possible hash value we can get. We will use a hash size of $2^{20} = 1{,}048{,}576$ when implementing this algorithm (which gives us just over a million possible hash values). Note that we can use the `mod` function to calculate the remainder when dividing by a value. Use MATLAB help to become familiar with the `mod` function if you are unsure how to use it.

**A note on working with character arrays and mathematical operations**

While the algorithm is described as working with an array of integer values, it can also be applied to an array of characters (i.e., a string), as every character can be interpreted as an ASCII value (which will be an integer). We will be using this algorithm to create hash values for character arrays, but you may find it easier to initially test your function using arrays of integer values – make sure it also works on character arrays.

Recall that if you have a character array and start working with mathematical operations (such as addition or multiplication), MATLAB will use the ASCII values of the characters, e.g., `'aced'+0` will give `[97 99 101 100]`.

You can also get ASCII values from a string by using the double function, e.g., `double('x=123')` returns `[120 61 49 50 51]`. Note how the ASCII value of the character `'1'` is 49, not 1.

**Work the problem by hand**

To understand an algorithm, it is always a good idea to work through it by hand a few times (it can be helpful to keep a table of values as you do so, updating them as they change). Suppose we wanted to calculate the hash value for the character array `'aced'` using a hash size of 1,048,576.

The ASCII values for this four-element array `'aced'` are `[97 99 101 100]`.

We begin by setting the $h$ equal to 0 and the index value $i$ equal to 1.

$i = 1$ is $\leq$ the length of our array, so we update our hash value by setting it to

$$h = a(1) + 31h = 97 + 31 \times 0 = 97$$

Dividing $h$ by the hash size gives a remainder of 97. Now we increment $i$ to 2.

$i = 2$ is $\leq$ the length of our array, so we update our hash value by setting it to

$$h = a(2) + 31h = 99 + 31 \times 97 = 3106$$

Dividing $h$ by the hash size gives a remainder of 3106. Now we increment $i$ to 3.

$i = 3$ is $\leq$ the length of our array, so we update our hash value by setting it to

$$h = a(3) + 31h = 101 + 31 \times 3106 = 96387$$

Dividing $h$ by the hash size gives a remainder of 96387. Now we increment $i$ to 4.

$i = 4$ is $\leq$ the length of our array, so we update our hash value by setting it to

$$h = a(4) + 31h = 100 + 31 \times 96387 = 2988097$$

Note this is bigger than our hash size.

Dividing $h$ by the hash size gives a remainder of 890945. Now we increment $i$ to 5.

| $h$ | $i$ |
|---|---|
| **0** | **1** |
| 97 | 2 |
| 3106 | 3 |
| 96387 | 4 |
| 890945 | 5 |

$i = 5$ is NOT $\leq$ the length of our array, so we finish the loop and are left with $h = 890945$.

Hence the hash value for the array `'aced'` is 890945.

**Example calls**

```
>> h = Hash31([1 2 3])

h =
        1026

>> h = Hash31([97 99 101 100])

h =
       890945

>> h = Hash31('aced')

h =
       890945

>> h = Hash31('function')

h =
     1012696

>> h = Hash31('hello there')

h =
      146838

>> h = Hash31('')

h =
     0
```

## The RightMin function

| | |
|---|---|
| Purpose | Finds the value and position of the rightmost minimum in an array of values. |
| Input | A 1D array of values (the array must contain at least one value). |
| Outputs | The minimum value and the rightmost position of that minimum |

MATLAB has a built-in minimum function which you can use to find a minimum value from a list of values **and** the leftmost position of that minimum (i.e., if there are duplicates, it returns the index position of the **first** occurrence of the minimum value).

For example, we can find the leftmost minimum as follows:

```
>> [m, pos] = min([3 1 4 1 5])

m =

     1

pos =

     2
```

This will give us `m = 1` and `pos = 2` (i.e., the first occurrence of our minimum value 1, is in index position 2 of the array).

For the winnowing algorithm, we need to find the **rightmost** minimum value (i.e., if there are duplicates, we want the index position of the **last** occurrence of the minimum value). For example, for the array [3 1 4 1 5], we would want to return the minimum value of 1 and the index position 4 as the last occurrence of 1 is in position 4.

Write a function that will find the minimum value and the **rightmost** position of that minimum.

See the next page for some example calls.

**Example calls**

```
>> [m, pos] = RightMin([3 1 4 1 5])
m =
     1
pos =
     4
>> [m, pos] = RightMin([1 0 2 3 4])
m =
     0
pos =
     2
>> [m, pos] = RightMin([1 2 3 4])
m =
     1
pos =
     1
>> [m, pos] = RightMin([5 5 5])
m =
     5
pos =
     3
>> [m, pos] = RightMin([3 1 -2 4])
m =
    -2
pos =
     3
>> [m, pos] = RightMin([10])
m =
    10
pos =
     1
```

## The FindMatchIndices function

| | |
|---|---|
| Purpose | Compares two non-empty 1D arrays and returns a list of index positions from array one of any values that also appear in array 2. Note the arrays being compared may be of different lengths. |
| Inputs | 1) Array 1<br>2) Array 2 |
| Output | A 1D array containing a list of index positions in array 1 of all values that are also found in array 2. Note this will be a **row array** with values listed in order from smallest to largest. If no matches were found, an **empty array** is returned (i.e., an empty 0x0 double array). |

This function finds values from array 1 that are also in array 2 and returns the index positions of these values in array 1 (i.e., if the *nth* element of array 1 appears anywhere in array 2, then we add position n to the list of positions to return).

For example, suppose array 1 = [3 1 4 1 5] and array 2 = [ 1 2 3 0].

We can see that both arrays contain the values 3 and 1.

- The first element of array 1 (i.e., 3) occurs within array 2.
- The second element of array 1 (i.e., 1) occurs within array 2.
- The third element of array 1 (i.e., 4) does **not** occur within array 2.
- The fourth element of array 1 (i.e., 1) occurs within array 2.
- The fifth element of array 1 (i.e., 5) does **not** occur within array 2.

Hence, we should return the array [1, 2, 4], indicating that the values in index positions 1, 2 and 4 of array 1 had matches in array 2.

This function may be helpful to call when writing the `FindMatchPositions` function.

**Example calls**

```
>> indices = FindMatchIndices([3 1 4 1 5],[1 2 3 0])

indices =
    1     2     4

>> indices = FindMatchIndices([1 2 3 0],[3 1 4 1 5])

indices =
     1     3

>> indices = FindMatchIndices([3 2 1],[5 4 3 2 1 0])

indices =
     1     2     3

>> indices = FindMatchIndices([10 20 30 40 50],[20 3 2 1 30])

indices =
     2     3

>> indices = FindMatchIndices([3 1 4 1 5],[10 20 30 0])

indices =
     []
```

## The StripString function

| | |
|---|---|
| Purpose | Strip out whitespace and other unprintable characters, plus convert all uppercase characters to lowercase. |
| Input | A string (i.e., an array of characters). |
| Output | An array of characters that has unprintable characters stripped out and any uppercase characters converted to lowercase. If the original string only contained unprintable characters or was empty, then an empty string is returned (i.e., an empty 0x0 char array). |

This function removes irrelevant features we want to discard from our string, so we can work with a string that only contains the information we wish to work with. The two key things this function does are, remove any whitespace (plus any other unprintable characters) and convert any uppercase characters to lowercase. Note that the ASCII values for some common white space characters are 9 (tab), 10 (line feed), 13 (carriage return), and 32 (space). These all have a value below 33.

To remove white space and unprintable characters we will discard any character with an ASCII value that falls outside the range of 33 to 126 (inclusive).

In addition, any uppercase characters will be converted to lowercase.

This function does NOT remove punctuation characters such as commas, semicolons, question marks and periods.

**Example calls**

Here are some example calls to `StripString`

```
>> s = StripString('Do you want to feel how it feels? (Ye-yeah, yeah, yo)')

s =

    'doyouwanttofeelhowitfeels?(ye-yeah,yeah,yo)'

>> exampleString = sprintf('This string spans\n3 lines\nand\tuses\ttabs!')

exampleString =

    'This string spans
     3 lines
     and    uses    tabs!'

>> stripped = StripString(exampleString)

stripped =

    'thisstringspans3linesandusestabs!'

>> stripped = StripString('     ') % array containing only spaces

stripped =

  0×0 empty char array              % equivalent to setting stripped = ''
```

## The Kgram function

| | |
|---|---|
| Purpose | Take a string and divide it into a sequence of k-grams, where each element is k characters long. |
| Inputs | 1) A k value (a positive integer greater than 0)<br>2) A string in the form of an array of characters |
| Output | A 1xn cell array where each element of the array is one of the k-grams. If k is greater than the length of the string, we return a 1x1 cell array containing the string. |

A k-gram is a substring containing k characters. We can divide up a string into a sequence of k-grams by taking the first k characters to be the first k-gram, then moving one position along the string and grabbing another k characters to form the next k-gram (most of the letters will overlap with the first). We continue in this way until we have generated all possible k-grams.

Suppose we have the string
`'doyouwantto'`

The sequence of 5-grams (i.e., k=5) for this string is

```
'doyou'    'oyouw'    'youwa'    'ouwan'    'uwant'    'wantt'    'antto'
```

The sequence of 7-grams (i.e., k=7) for this string is

```
'doyouwa'    'oyouwan'    'youwant'    'ouwantt'    'uwantto'
```

**Example calls**

```
>> kg = Kgram(5,'doyouwantto')

kg =

  1×7 cell array

    {'doyou'}  {'oyouw'}  {'youwa'}  {'ouwan'}  {'uwant'}  {'wantt'}  {'antto'}

>> kg = Kgram(7,'doyouwantto')

kg =

  1×5 cell array

    {'doyouwa'}    {'oyouwan'}    {'youwant'}    {'ouwantt'}    {'uwantto'}

>> kg = Kgram(12,'doyouwantto') %note k=12 is greater than length of the string

kg =

  1×1 cell array

    {'doyouwantto'}
```

## The HashList function

| | |
|---|---|
| Purpose | Calculates the hash31 value for every string stored in a cell array to generate a sequence of hashed values (which will be integers). |
| Input | A 1xn cell array where each element of the array is a string (i.e., an array of characters). |
| Output | A 1xn array containing hash values for the corresponding elements of the cell array. |

This function calculates and returns the hash values for every string contained within a cell array, using the hash 31 algorithm. For example, suppose the input is a 1x5 cell array with five elements (where each element is a character array). In that case, a 1x5 array will be returned (where the first value is the hash for the first element of the cell array, the second value is the has for the second element of the cell array, etc.).

**Hint**: you will likely find it helpful to call your Hash31 function.

**Example calls**

```
>> kg = {'doyou', 'oyouw', 'youwa', 'ouwan', 'uwant', 'wanto'};

>> hashes = HashList(kg)

hashes =

      358324      319783      874281      207678      544773      700511

>> words = {'this' 'is' 'a four element' 'cell array'};

>> hashes = HashList(words)

hashes =

      413342        3370       948065      917019

>> ex = {'a' 'short' 'example'};

>> hashes = HashList(ex)

hashes =

          97      361596      332138
```

# The Window function

| | |
|---|---|
| Purpose | Takes a sequence of integer values and creates a 2D array of windows for a specified window size w. |
| Inputs | 1) The window size, w (a positive integer greater than 0).<br>2) A 1D array of integer values. |
| Output | A 2D array of integer values where each row of the array is a window containing w integer values. If w is greater than the length of the array, we return the array. |

We can divide an array of integer values into a set of windows by taking the first w values from the array and assigning them to row 1. We then move along one position and grab another w values starting from position 2 (and assign them to row 2). Note that most values in the second window will overlap with the first). We continue in this way until we have generated all possible windows (in order).

Suppose we have the 1D array:

```
[ 3 1 4 1 5 9 2 6 5]
```

The set of windows of size w=4 written out as an array is:

```
3     1     4     1
1     4     1     5
4     1     5     9
1     5     9     2
5     9     2     6
9     2     6     5
```

Window 1 (in row 1) is `3 1 4 1` (the first four values)
Window 2 (in row 2) is `1 4 1 5` (the values from position 2 to position 5).
There are seven windows in total.

Contrast this with the set of windows of size w=8 for the same array:

```
3     1     4     1     5     9     2     6
1     4     1     5     9     2     6     5
```

**Note**: we only have two windows here, as there were only 9 values in the 1D array.

See the next page for some example calls.

**Tip**: take care when naming and calling this function, `Window` is NOT the same as `window`

**Example calls**

```
>> windows = Window(4,[3 1 4 1 5 9 2 6 5])


windows =

     3     1     4     1

     1     4     1     5

     4     1     5     9

     1     5     9     2

     5     9     2     6

     9     2     6     5


>> windows = Window(8,[3 1 4 1 5 9 2 6 5])


windows =

     3     1     4     1     5     9     2     6

     1     4     1     5     9     2     6     5

>> windows = Window(2,[3 1 4 1 5 9 2 6 5])


windows =

     3     1

     1     4

     4     1

     1     5

     5     9

     9     2

     2     6

     6     5

>> windows = Window(12,[3 1 4 1 5 9 2 6 5]) % w greater than array length


windows =

     3     1     4     1     5     9     2     6     5
```

## The Fingerprint function

| | |
|---|---|
| Purpose | Takes a set of windows (in the form of a 2D array) and calculates a document fingerprint using the winnowing algorithm (i.e., we select the rightmost minimum from each window). It will return the document fingerprint as a 2D array in which each column contains a winnowed value and its corresponding position. |
| Input | A n x w 2D array representing a set of windows, where each window has w values in it. |
| Output | A 2 row 2D array representing a fingerprint, which consists of a set of winnowed values (the first row) and the relative positions of those values (the second row). Each column of the output, therefore, contains a hash value/position pair. |

This function performs the heart of the winnowing algorithm. It cycles through each window and finds the rightmost minimum from that window, keeping track of the position of that value (with respect to the original array that the windows were created from)

If the minimum in that relative position has not yet been added to the fingerprint, then it is added to the list (with its hash value added to the first row and its relative position added to the second row).

Consider the following hash value sequence:

```
[77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98]
```

We will form a set of windows from these values, using a window length of w = 4. The set is shown below, with the rightmost minimum in each window identified in bold.

```
77    74    42    17  % window 1
74    42    17    98  % window 2
42    17    98    50  % window 3
17    98    50    17  % window 4, etc.
98    50    17    98
50    17    98     8
17    98     8    88
98     8    88    67
 8    88    67    39
88    67    39    77
67    39    77    74
39    77    74    42
77    74    42    17
74    42    17    98
```

Note that in many cases the rightmost value in one window corresponds to the same hash value in the same position from the original array. For example, the first three windows all identify the rightmost minimum as the value 17, which comes from index position 4 in the original hash value sequence. However, it is possible to have the same hash value in a different position (e.g., the 17 in window 4 comes from index position 7).

This becomes more obvious if we identify all the bold values in the original hash value sequence (and then include their relative positions below that, with the positions associated with minimums highlighted in red):

```
[77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98]  % hash values
[ 1  2  3  4  5  6  7  8 9 10 11 12 13 14 15 16 17]  % relative positions
```

The fingerprint for this window is the bold hash values and their associated relative positions:

```
17    17     8    39    17
 4     7     9    12    16
```

**Example calls**

```
>> hashes = [77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98];
>> w = Window(4,hashes)

w =
    77    74    42    17
    74    42    17    98
    42    17    98    50
    17    98    50    17
    98    50    17    98
    50    17    98     8
    17    98     8    88
    98     8    88    67
     8    88    67    39
    88    67    39    77
    67    39    77    74
    39    77    74    42
    77    74    42    17
    74    42    17    98

>> f = Fingerprint(w)

f =

    17    17     8    39    17
     4     7     9    12    16

>> w = Window(8,[3 1 4 1 5 9 2 6 5])

w =

     3     1     4     1     5     9     2     6
     1     4     1     5     9     2     6     5

>> f = Fingerprint(w)

f =

     1
     4

>> w = Window(2,[3 1 4 1 5 9 2 6 5])

w =

     3     1
     1     4
     4     1
     1     5
     5     9
     9     2
     2     6
     6     5

>> f = Fingerprint(w)

f =

     1     1     5     2     5
     2     4     5     7     9
```

## The FindMatchPositions function

| | |
|---|---|
| Purpose | Compare two fingerprints and determine<br>1) which positions in fingerprint 1 have a value that is also found in string 2.<br>2) which positions in fingerprint 2 have a value that is also found in string 1. |
| Input | 1) A fingerprint for string 1 (i.e., a 2D array where each column contains a hash value and its corresponding position).<br>2) A fingerprint for string 2 (i.e., a 2D array where each column contains a hash value and its corresponding position). |
| Output | 1) A 1D array containing a list of index positions in **string 1** of all values that are also found in string 2. Note this will be a **row array.** If no matches were found, an **empty array** is returned.<br>2) A 1D array containing a list of index positions in **string 2** of all values that are also found in string 2. Note this will be a **row array.** If no matches were found, an **empty array** is returned. |

Suppose we have the following two fingerprints

| Fingerprint 1 | Fingerprint 2 |
|---|---|
| 10 30 20<br>2 4 7 | 30 40 30 10<br>1 3 5 7 |

Which positions in fingerprint 1 have a hash value that is also found in fingerprint 2?

**Recall that the first row of the fingerprint contains the hash values.**

The fingerprint 1 hash values are 10, 30 and 20. Fingerprint 2 has hash values 30, 40, 30 and 10.

We can see that the two fingerprints both have the hash values **10** and **30**.

| Fingerprint 1 | Fingerprint 2 |
|---|---|
| **10** **30** 20<br>**2** **4** 7 | **30** 40 **30** **10**<br>**1** 3 **5** **7** |

**Recall that the second row of the fingerprint contains the positional information.**

For fingerprint 1, we have therefore found matches that correspond to positions 2 and 4, so our first output would contain [2 4]

For fingerprint 2, we have found matches that correspond to positions 1, 5 and 7, so our second output would contain [1 5 7]

**Note:** If there are no matches, then empty arrays will be returned for both outputs.

**Example call**

```
>> f1 = [10 30 20; 2 4 7]

f1 =

    10    30    20
     2     4     7

>> f2 = [30 40 30 10; 1 3 5 7]

f2 =

    30    40    30    10
     1     3     5     7

>> f3 = [21 50 60; 1 4 7]

f3 =

    21    50    60
     1     4     7

>> [p1,p2] = FindMatchPositions(f1,f2)

p1 =

     2     4


p2 =

     1     5     7

>> [p1,p2] = FindMatchPositions(f1,f1)

p1 =

     2     4     7


p2 =

     2     4     7

>> [p1,p2] = FindMatchPositions(f1,f3)

p1 =

     []


p2 =

     []
```

## The SimilarityScore function

| | |
|---|---|
| Purpose | Calculates a similarity score by taking a list of matched positions for a string and determining what proportion of characters in a string matched. |
| Inputs | 1) A 1D array containing a list of indices which correspond to position matches (this could be an empty array).<br>2) The k value indicates the length of each match (which will be a positive integer greater than zero).<br>3) The total length of the string that was converted into k-grams. |
| Output | The percentage of the string that matched, returned as a decimal (e.g., 77% would be returned as 0.77). |

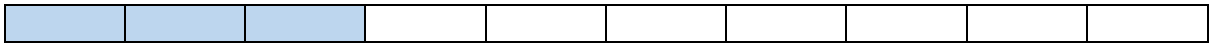
This function calculates what proportion of the stripped string is similar, given a list of positions identified as having a match in another string. Each matched position corresponds to k characters that match from the original stripped string. Note that care needs to be taken when calculating the overall proportion of matching characters, as it is possible that matched positions could share some characters in common.

Consider the matches [1 2 6] for a k value of 3 and a stripped string length of 10.

We can mark the characters that match in the original string by considering each match in turn.

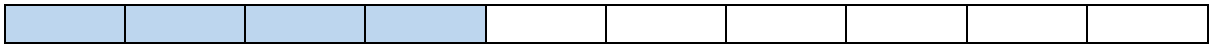
There is a match in position 1, and since k=3, that means characters 1, 2 and 3 must match.

I will highlight the appropriate characters to indicate this:

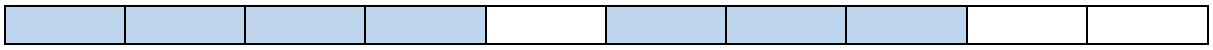


There is also a match in position 2, and since k=3, that means characters 2, 3 and 4 must match.

We already knew that characters 2 and 3 matched but can now highlight the fourth position as a match:



Finally, we have a match in position 6. With k=3, that means characters 6, 7 and 8 must match, so we can also highlight those:



We can see that 7 of the 10 characters matched, so the function will be return a score of 0.7 (i.e., 70%)

See the next page for some example calls.

**Example calls**

```
>> score = SimilarityScore([1 2 6], 3, 10)

score =

    0.7000

>> score = SimilarityScore([1 2 6], 3, 100)

score =

    0.0700

>> score = SimilarityScore([1 2 3 4 5], 4, 8)

score =

     1

>> score = SimilarityScore([1 3 5], 5, 10)

score =

    0.9000

>> score = SimilarityScore([], 5, 10)

score =

     0
```

## Checklist of Functions to Write in Alphabetical Order

Remember there are **ten** in total:

| Function | Difficulty Level | Where used |
|---|---|---|
| `FindMatchIndices` | Medium | Likely to be handy when writing your `FindMatchPositions` function. |
| `FindMatchPositions` | Medium/Hard | Called from `CompareFiles`. |
| `Fingerprint` | Hard | Called from `Winnow`. |
| `Hash31` | Easy | Likely to be handy when writing your `HashList` function. |
| `HashList` | Easy | Called from `Winnow`. |
| `Kgram` | Easy | Called from `Winnow`. |
| `RightMin` | Easy | Likely to be handy when writing your `Fingerprint` function. |
| `SimilarityScore` | Medium/Hard | Called from `CompareFiles`. |
| `StripString` | Easy/Medium | Called from `Winnow`. |
| `Window` | Easy | Called from `Winnow`. |

## How the project is marked

You will receive marks for style (is it well written?) and functionality (does your code work?) When you submit your code to MATLAB Grader, it will run some tests to determine the marks for functionality. You can submit **each** function up to **six** times, and your functionality mark for that function will be based on the function submission that performed the best. You will be marked for style by a human marker, who will download your function submissions (choosing the ones that performed the best) and mark these for style.

Each of the **ten** required functions will be marked independently for functionality, so even if you can't get everything to work, please do submit the functions you have written. Some functions may be harder to write than others, so if you are having difficulty writing a function, you might like to try working on a different function for a while and then come back to the harder one later.

Note it is still possible to get marks for good style, even if your code does not work at all! Style includes elements such as using sensible variable names, having header comments, commenting the rest of your code, avoiding code repetition, and using correct indentation.

Using only concepts taught in this course each function can be written in less than 20 lines of code (not including comments), and some functions can be written using just a few lines[7] (as an example, my longest function was 16 lines of code, and my shortest four were just 5 lines of code each, with an average length of under 8 lines per function). Do not stress if your code is longer. It is perfectly fine if your project solution runs to many lines of code, as long as your code works and uses a good programming style.

[7] Advanced coders can sometimes accomplish quite complicated tasks with just a few lines of code. One of my alpha testers managed to write every specified function using just three lines of code (that includes the function line and the end, leaving just one line for the body of the function!) To do this required using programming techniques that are beyond the scope of this course.

## How the project is submitted

Submit by uploading your code to MATLAB Grader. You should be able to work on your project from anywhere, assuming you have access to MATLAB (either installed on a computer or running online) and can access the internet to upload your final submission.

As for the labs, you will submit your project code by copying and pasting it into MATLAB Grader.

Before clicking the "Submit" button, we **strongly** recommend you try clicking on the "Run function" button to ensure you have copied and pasted everything over correctly (otherwise, you risk wasting a submission attempt on a copy/paste error).

IMPORTANT: When submitting a particular function, if your function calls helper functions (other than `Hash31`, `RightMin` and `FindMatchIndices`), then you will need to paste the code for those functions below the particular function that you are submitting so that MATLAB Grader has access to it.

## Any questions?

If you have any questions regarding the project, please first check through this document. If it does not answer your question, then feel free to ask the question on the class Piazza forum. Remember that you should **NOT** be publicly posting any of your project code on Piazza, so if your question requires that you include some of your code, make sure it is posted as a **PRIVATE** piazza query.

Have fun!