Tidewater Big Data Enthusiasts: Exploring Kaprekar's Constant

Chuck Cartledge, Phd

July 16, 2024

Contents

1 Introduction **1**

2 Discussion **1**

3 Algorithms **2**

4 Listings **9**

5 Results **37**
5.1 Overview . . . 38
5.2 Width = 2 . . . 41
5.3 Width = 3 . . . 45
5.4 Width = 4 . . . 48
5.5 Width = 5 . . . 51
5.6 Width = 6 . . . 54
5.7 Width = 7 . . . 56
5.8 Width = 8 . . . 58

6 Future Work **60**

7 Conclusion **60**

8 Embedded files **62**

List of Tables

1 Kaprekar terminating decimal values based on numeric width. 61

List of Algorithms

1 The “classic” Kaprekar routine. 3
2 The Kaprekar routine without 6174. 4
3 The Kaprekar routine with other numeric widths. 4
4 The Kaprekar routine operations. 5
5 The Kaprekar routine operations in different bases. 6
6 The Kaprekar routine range of values. 7
7 Recreating Kaprekar’s original values. 8
8 Parallel computation of Kaprekar routine across a range of values. 9

List of Figures

1 Sample “classic” Kaprekar header length histogram. 38
2 Sample “classic” Kaprekar cycle length histogram. 39
3 Sample “classic” Kaprekar cycle and header length frequency plot. 40
4 Sample “classic” Kaprekar repeat value. 40
5 Width = 2, all values processed. 42
6 Width = 2, only unique values processed. 44
7 Width = 3, all values processed. 46
8 Width = 3, only unique values processed. 47
9 Width = 4, all values processed. 49
10 Width = 4, only unique values processed. 50
11 Width = 5, all values processed. 52
12 Width = 5, only unique values processed. 53
13 Width = 6, only unique values processed. 55
14 Width = 7, only unique values processed. 57
15 Width = 8, only unique values processed. 59

List of Listings

1 Implementing the classic Kaprekar routine (see Algorithm 1) (R script: kaprekar-classic.R). 10

2 Implementing the Kaprekar routine without knowing 6174 (see Algorithm 2) (R script: kaprekar-unknown.R). 12
3 Implementing the Kaprekar routine with different numeric width (see Algorithm 3) (R script: kaprekar-width.R). 14
4 Extracting the number of operations to compute a repeating value (see Algorithm 4) (R script: kaprekar-where.R). 16
5 A support library used when computing numeric bases other than 10. This file is needed when exploring with Kaprekar with different bases (R script: basesLibrary.R). 19
6 Exploring Kaprekar's routine using different numeric bases (see Algorithm 5) (R script: kaprekar-bases.R). 23
7 Computing the number of unique values after Kaprekar's first step (see Algorithm 6) (R script: uniqueNumbers.R). 27
8 Comparing the parallel computation times Kaprekar's constant (see Algorithm 8) (R script: kaprekar-parallel-stats.R). 30
9 Reconstructing original values from a Kaprekar mapped value (see Algorithm 7) (R script: recreateNumbers.R). 37

1 Introduction

We explore the world hinted at by Kaprekar's routine to manipulate four digit decimal numbers. Kaprekar's routine takes a number, manipulates the digits, subtracts two newly created values, and repeats these steps until the value 6174 is computed. Once 6174 is computed, the routine is "stuck" at this value and will never change. The simplicity of the routine cries for computerization to determine if all four digit decimal numbers get stuck at 6174, or not.

Initially we explore four digit numbers, then expand the exploration into two, three, five, six, seven, and eight digit numbers. Along the way, we discover that of the 10,000 four digit values, only 750 need to be evaluated because of the numeric mapping of Kaprekar's routine. We also explore the space where numerical bases other than decimal are used in Kaprekar's routine.

This report contains high level algorithms, R scripts listing implementing those algorithms, and actual R script files.

2 Discussion

Kaprekar's Constant is a trivial mathematical exercise that can be done with paper and pencil to amaze your friends and confound your enemies. The routine is:

1. Take any four-digit number, using at least two different digits (leading zeros are allowed).
2. Arrange the digits in descending and then in ascending order to get two four-digit numbers, adding leading zeros if necessary.
3. Subtract the smaller number from the bigger number.
4. Go back to step 2 and repeat until the number is 6174.

The number of times you have to repeat the steps varies from 1 to 7 and never more than 7 times. Kaprekar's Constant has a long and rich history. Only a few are mentioned here [2, 3]. Because of the simplicity of Kaprekar's routine, a number of questions came to mind:

1. Do all 4 digit numbers terminate at 6174?
2. Do all values terminate at 6174?
3. Does the routine terminate?
4. Does the routine terminate with numeric bases other than decimal?
5. Do all numeric width values need to be explored?
6. Will all values, regardless of base or width terminate?

These questions are explored and at least partially answered using the included algorithms. The algorithms are implemented in various R scripts that are shown as listings and are embedded in this report.

3 Algorithms

The routines to compute Kaprekar's constant [2, 3] do not layout an algorithm in the strict sense. They provide a series of steps, but do not define input requirements, end condition, and outputs. We will take the original routine and expand it to address these shortages.

Algorithm 1: The "classic" Kaprekar routine. There are several assumptions built into this routine that we will extract to create "tweakable knobs." Most people select a 4 digit number when the routine starts. In reality, any number commonly written with 1 or more digits has an infinite number of leading zeros that are not commonly written. The first step in the Kaprekar routine is to pad the current number with as many zeros as needed to make the number of digits four. The algorithm continues until the value 6174 is reached with no explanation of why the routine terminates when this value is reached.

Algorithm 2: The Kaprekar routine without 6174. The routine continues until a previously encountered value is computed. Encountering a number the second time means that the system has entered into a circular sequence and will be stuck there forever, so it doesn't make sense to continue computing.

Algorithm 3: The Kaprekar routine with other numeric widths. "Classic" Kaprekar depends on a numbers that are 4 digits wide. Width is a tweakable knob we can change to see if Kaprekar behavior appears with different widths.

Algorithm 4: The Kaprekar routine operations. Exploring the Kaprekar space breaks down to taking a value, diddling the value into two values, subtracting those values, and detecting when a value is repeated. The number of computations until the repeated value is computed, and how many operations it takes to compute the repeated value is of interest. Based on the initial value, the number of operations to compute and recompute the repeated value differ.

Algorithm 5: The Kaprekar routine operations in different bases. Inherent in the Kaprekar routine is the idea that the mathematical operations are based on using decimal values. The numeric base is a tweakable knob that can be changed to see how the Kaprekar routine behaviors on different bases.

Algorithm 6: The Kaprekar routine range of values. "Classic" Kaprekar routine claims that all 4 digit values will eventually collapse to 6174. It is easy enough to test all 4 digit

values to see if the claim is true. Testing wider numbers is also reasonably easy to test. But do all values have to be tested? Kaprekar decomposes the value into to individual characters, sorts those characters, and then performs mathematical operations based on the sorted values. When the initial value is decomposed, it is no longer a unique value. By understanding how Kaprekar removes uniqueness from a value, the total number of unique values within a "width" space can be greatly reduced.

Algorithm 7: Recreating Kaprekar's original values. Reducing the number of unique values that need to be processed by the Kaprekar routine (see Algorithm 6) to see the distribution of operations as a function of the initial value. The original values can be "reconstructed" by permuting the unique values, essentially "unmapping" the values.

Algorithm 8: Parallel computation of Kaprekar routine across a range of values. Regardless of how often, or how deeply the Kaprekar data is reduced, the computation time is linear. Exploring the Kaprekar space is a trivially paralizable process because each value be considered is independent of every other value. The total number of values to be considered can be spread across the available cores for processing.

The algorithms described in this section are instantiated as R scripts (see Section 8). The R scripts may include "library" type files that are also embedded in this report.

Algorithm 1: The "classic" Kaprekar routine. There are several assumptions built into this routine that we will extract to create "knobs" that we can tweak.

Data: num ← a random 4 digit number with at least two different digits
Result: *None*

```
1 initialization:None;
2 repeat
3     num ← num;
      /* num is padded with leading zeros to 4 digits with at least 2
         different digits                                            */
4     digits ← individual digits from num;
5     minuend ← sort digits high to low and convert into value;
6     subtrahend ← sort digits low to high and convert into value;
7     num ← minuend - subtrahend;
8 until num == 6174;
```

Algorithm 2: The Kaprekar routine without 6174. The routine will continue until the current computed number has been previously found.

```
   Data: num ← a random 4 digit number with at least 2 different digits
   Result: None
 1 initialization;
 2 list ← Empty a way to keep track of previously computed values;
 3 repeat
 4     list ← num is appended to list;
 5     num ← num;
       /* num is padded with leading zeros to 4 digits with at least 2
          different digits                                            */
 6     digits ← individual digits from num;
 7     minuend ← sort digits high to low and convert into value;
 8     subtrahend ← sort digits low to high and convert into value;
 9     num ← minuend - subtrahend;
10 until num in list;
```

Algorithm 3: The Kaprekar routine with other numeric widths. The routine will continue until a previously computed value is found. Different number widths and different initial values will result in different previously computed values being found.

```
   Data: width ← maximum number of digits wide for any number
   Data: num ← a random width digit number with at least 2 different digits
   Result: None
 1 initialization;
 2 list ← Empty a way to keep track of previously computed values;
 3 repeat
 4     list ← num is appended to list;
 5     num ← num;
       /* num is padded with leading zeros to width digits with at least 2
          different digits                                                */
 6     digits ← individual digits from num;
 7     minuend ← sort digits high to low and convert into value;
 8     subtrahend ← sort digits low to high and convert into value;
 9     num ← minuend - subtrahend;
10 until num in list;
```

Algorithm 4: The Kaprekar routine operations. The number of operations to compute the repeated value, and the number of operations to re-compute the repeated value differs based on the initial value and the selected width.

```
   Data: width ← maximum number of digits wide for any number
   Data: num ← a random width digit number with at least 2 different digits
   Result: How many subtractions to reach the repeated value the first time
   Result: How many subtractions to reach the repeated value the second time
1  initialization;
2  list ← Empty a way to keep track of previously computed values;
3  repeat
4  |  list ← num is appended to list;
5  |  num ← num with leading zeros to width digits wide;
6  |  digits ← individual digits from num;
7  |  minuend ← sort digits high to low and convert into value;
8  |  subtrahend ← sort digits low to high and convert into value;
9  |  num ← minuend - subtrahend;
10 until num in list;
11 repeatLocation ← location of num in list /* The location of the repeated value
      from the front of the list.                                              */
12 repeatLength ← length of list - repeatLocation/* The number of operations from
      the location of the repeated value until the value is repeated.          */
13 return repeatLocation, repeatLength;
```

Algorithm 5: The Kaprekar routine operations in different bases. Integral to the Kaprekar routine is the ordering of the digits in the number prior to subtracting the two numbers. Different numeric bases have different characters. Sorting the characters (either high to low, or low to high) is often based on the lexical ordering used to represent the values. Using different bases results in different numbers of operations before the repeated value and the number of operations needed to reach it a second time. This algorithm is based on [1].

```
   Data: width ← maximum number of digits wide for any number
   Data: num ← a random width digit number with at least 2 different digits
   Data: base ← a numeric base for all operations
   Result: How many subtractions to reach the repeated value the first time
   Result: How many subtractions to reach the repeated value the second time
1  initialization;
2  list ← Empty a way to keep track of previously computed values;
3  repeat
4  |   list ← num is appended to list;
5  |   num ← num with leading zeros to width digits wide;
6  |   digits ← individual digits from num;
7  |   minuend ← sort digits high to low and convert into value;
8  |   subtrahend ← sort digits low to high and convert into value;
9  |   num ← minuend - subtrahend;
10 until num in list;
11 repeatLocation ← location of num in list /* The location of the repeated value
       from the front of the list.                                              */
12 repeatLength ← length of list - repeatLocation/* The number of operations from
       the location of the repeated value until the value is repeated.   */
13 return repeatLocation, repeatLength;
```

Algorithm 6: The Kaprekar routine range of values. Initial steps of the Kaprekar's routine are to split the digits of the numeric value, and then sort the digits. This splitting and sorting reduces the number of unique values based on the numeric width. While the values 1234 and 1243 are different in the "normal" sense, splitting and sorting the digits for both initial values result in 4321 and 4321, i.e. 4321. Permuting the digits in 1234 results in 24 different values, all of which result in 4321 in the Kaprekar routine. 4321 need only be evaluated once to see how the other 23 values would be handled.

Data: width $\leftarrow$ maximum number of digits wide for any number
Data: base $\leftarrow$ a numeric base for all operations
Result: The set of functionally unique Kaprekar values

1. initialization;
2. file $\leftarrow$ a temporary file to store values;
3. num = 1;
4. **for** $num \leq (base^{width} - 1)$ **do**
5. temp $\leftarrow$ num with leading zeros to width digits wide;
6. digits $\leftarrow$ individual digits from temp;
7. uniqueDigits $\leftarrow$ (sort(digits) $\rightarrow$ unique() $\rightarrow$ length());
8. **if** *uniqueDigits* $\geq$ *2* **then**
9. value $\leftarrow$ sort digits low to high and convert to string;
10. write *value* to *file*;
11. **end**
12. num ++;
13. **end**
14. read *file* $\rightarrow$ sort strings $\rightarrow$ unique strings;
`/* Use OS read, sort, and unique functions to cull duplicates from the file. */`
15. **return** *Unique sorted strings*;

Algorithm 7: Recreating Kaprekar's original values. Processing all values within numeric width constraints can take an excess amount of time (see Section 5.6). Because one of the first steps in the Kaprekar routine is to decompose the input value into its component characters, then sort those, components, and create a new value based on the reordered components, many initially unique values are "mapped" to a common representation. This common representation can then be "unmapped" to retrieve the "mapped" values by permuting the components of the mapped value.

```
   Data: KaperkarUniqueString ← a Kaprekar formatted string
   Result: The set of functionally unique Kaprekar values
 1 initialization;
 2 digits ← KaperkarUniqueString expanded into single characters;
 3 permutedSet ← permute digits into set individual subsets;
 4 outputList ← empty list;
 5 num = 1;
   /* "Recreate" equivalent Kaprekar values */
 6 for num ≤ |permutedSet| do
 7 |   outputList[num] ← Concatenate permutedSet[num] members into one string;
 8 |   num ++;
 9 end
10 return unique values from outputList
```

Algorithm 8: Parallel computation of Kaprekar routine across a range of values. The essence of any trivially paralizeable algorithm is to reduce the data into a "consistent" form for processing, distribute a portion of that data to each computing element, and then consolidate each computing element's results into coherent composite.

Data: width $\leftarrow$ maximum number of digits wide for any number
Data: base $\leftarrow$ a numeric base for all operations
Data: cores $\leftarrow$ the number of cores in the local CPU
Result: The head and cycle lengths, and repeat values for range of values

1 initialization;
2 values $\leftarrow$ either all values, or only unique values;
3 breaks $\leftarrow$ "break" the values to be spread equally across all cores;
4 num = 1;
5 Configure cores as a cluster;
6 **for** *num* $\leq$ *cores* **do**
7   Distribute values, base, and width to individual cores;
8   Begin individual core execution;
9   num ++;
10 **end**
11 **for** *num* $\leq$ *cores* **do**
12   Collect Kaprekar values from each core;
13   num ++;
14 **end**
15 **return** *Collected data from all cores*;

4 Listings

This section includes R language listings that implement the algorithms (see Section 3). The scripts are fragile in that they do not perform any verifications on user inputs. They run as written, providing nonsensical values as parameters to the individual *main* function may have unexpected behaviors.

```r
rm(list=ls())

`%+=%` = function(e1,e2) eval.parent(substitute(e1 <- (e1) + (e2)))

getDigits <- function(number, format)
{
    singletons <- sprintf(format, number) |>
        strsplit("") |>
        unlist()
    return(singletons)
}

orderDigits <- function(digits, sortDecreasing=TRUE)
{
    returnValue <- sort(digits, decreasing=sortDecreasing) |>
        paste0(collapse="") |>
        as.integer()

    return(returnValue)
}

main <- function(num)
{
    width <- 4

    format <- sprintf("%s0%0.f.0f", "%", width)

    subtractionFormat <- sprintf("%s %s - %s = %s",
                                 "(%3.0f.)", format, format, format)

    counter <- 0
    repeat
    {
        counter %+=% 1

        digits <- getDigits(num, format)

        minuend <- orderDigits (digits, sortDecreasing=TRUE)

        subtrahend <- orderDigits (digits, sortDecreasing=FALSE)

        num <- minuend - subtrahend

        print(sprintf(subtractionFormat, counter, minuend, subtrahend, num))

        if (num == 6174)
        {
            break
```

```
49            }
50        }
51
52        print("The program has ended.")
53 }
54
55 ## https://en.wikipedia.org/wiki/6174
56 initialValue <- 1549
57
58 main(initialValue)
```

Listing 1: Implementing the classic Kaprekar routine (see Algorithm 1) (R script: kaprekar-classic.R).

Implementing the "classic" search for Kaprekar's constant. Running the script with the default argument results in:

```
[1] "(  1.) 9541 - 1459 = 8082"
[1] "(  2.) 8820 - 0288 = 8532"
[1] "(  3.) 8532 - 2358 = 6174"
[1] "The program has ended."
```

The *getDigits* function (see Listing 1) ensures that *number* has the appropriate number of leading zeros (if needed), then splits the values into individual digits. Ordering and converting digits is executed at different places in the Kaprekar routine, and is collected into a single function *orderDigits*. *main* continues blindly looking for 6174. The program works because of outside knowledge that 6174 will be computed eventually.

```r
rm(list=ls())

`%+=%` = function(e1,e2) eval.parent(substitute(e1 <- (e1) + (e2)))

getDigits <- function(number, format)
{
    singletons <- sprintf(format, number) |>
        strsplit("") |>
        unlist()
    return(singletons)
}

orderDigits <- function(digits, sortDecreasing=TRUE)
{
    returnValue <- sort(digits, decreasing=sortDecreasing) |>
        paste0(collapse="") |>
        as.integer()

    return(returnValue)
}

main <- function(num)
{
    width <- 4

    format <- sprintf("%s0%0.f.0f", "%", width)

    subtractionFormat <- sprintf("%s %s - %s = %s",
                                  "(%3.0f.)",format, format, format)

    counter <- 0

    previousValues <- c()
    repeat
    {
        counter %+=% 1

        digits <- getDigits(num, format)

        minuend <- orderDigits (digits, sortDecreasing=TRUE)

        subtrahend <- orderDigits (digits, sortDecreasing=FALSE)

        num <- minuend - subtrahend

        print(sprintf(subtractionFormat, counter, minuend, subtrahend, num))

        if (num %in% previousValues)
```

```
49          {
50              break
51          }
52
53          previousValues <- c(previousValues, num)
54      }
55
56      print("The program has ended.")
57  }
58
59  ## https://en.wikipedia.org/wiki/6174
60  initialValue <- 1549
61
62  main(initialValue)
```

Listing 2: Implementing the Kaprekar routine without knowing 6174 (see Algorithm 2) (R script: kaprekar-unknown.R).

Implementing a "naïve" search for Kaprekar's constant assuming that the previous computed value is the final value (see Listing 2). Running the script with the default argument results in:

```
[1] "(  1.) 9541 - 1459 = 8082"
[1] "(  2.) 8820 - 0288 = 8532"
[1] "(  3.) 8532 - 2358 = 6174"
[1] "(  4.) 7641 - 1467 = 6174"
[1] "The program has ended."
```

The function *main* keeps track of all previously computed values, and terminates when a computed value is found in the list of previously computed values. There is a subtlety in the check for a previous value, in that all previous values are checked, not just the last one (see Listing 1).

```r
rm(list=ls())

`%+=%` = function(e1,e2) eval.parent(substitute(e1 <- (e1) + (e2)))

getDigits <- function(number, format)
{
    singletons <- sprintf(format, number) |>
        strsplit("") |>
        unlist() |>
        sort()
    return(singletons)
}

orderDigits <- function(digits, sortDecreasing=TRUE)
{
    returnValue <- sort(digits, decreasing=sortDecreasing) |>
        paste0(collapse="") |>
        as.integer()

    return(returnValue)
}

main <- function(num, width)
{
    format <- sprintf("%s0%0.f.0f", "%", width)

    subtractionFormat <- sprintf("%s %s - %s = %s",
                                 "(%3.0f.)", format, format, format)

    counter <- 0
    previousValues <- c()
    repeat
    {
        counter %+=% 1
        digits <- getDigits(num, format)

        minuend <- orderDigits (digits, sortDecreasing=TRUE)

        subtrahend <- orderDigits (digits, sortDecreasing=FALSE)

        num <- minuend - subtrahend

        print(sprintf(subtractionFormat, counter, minuend, subtrahend, num))

        if (num %in% previousValues)
        {
            break
        }
```

```
49
50         previousValues <- c(previousValues, num)
51     }
52
53     print("The program has ended.")
54 }
55
56 ## https://en.wikipedia.org/wiki/6174
57 initialValue <- 1549
58 width <- 4
59
60 main(initialValue, width)
61
62 initialValue <- 49
63 width <- 3
64
65 main(initialValue, width)
```

Listing 3: Implementing the Kaprekar routine with different numeric width (see Algorithm 3) (R script: kaprekar-width.R).

Both the “classic” and “unknown” Kaprekar’s constant search are predicated on looking at values that are 4 digits wide (leading zeros are added as needed). Now that we understand how searching can work, what happens if we remove the 4 digit restriction? Running the script with the default values (see Listing 3), results in this output:

```
[1] "(  1.) 9541 - 1459 = 8082"
[1] "(  2.) 8820 - 0288 = 8532"
[1] "(  3.) 8532 - 2358 = 6174"
[1] "(  4.) 7641 - 1467 = 6174"
[1] "The program has ended."
[1] "(  1.) 940 - 049 = 891"
[1] "(  2.) 981 - 189 = 792"
[1] "(  3.) 972 - 279 = 693"
[1] "(  4.) 963 - 369 = 594"
[1] "(  5.) 954 - 459 = 495"
[1] "(  6.) 954 - 459 = 495"
[1] "The program has ended."
```

The first block is uses the default values from [3]. The second block comes from running the same routines, only changing the initial value, and the width of all digits. It is apparent that different width values control the terminating “constant.” Now that we can control the width of the values, we can see how different values affect the results.

```
rm(list=ls())

`%+=%` = function(e1,e2) eval.parent(substitute(e1 <- (e1) + (e2)))

getDigits <- function(number, format)
{
    singletons <- sprintf(format, number) |>
        strsplit("") |>
        unlist() |>
        sort()
    return(singletons)
}

orderDigits <- function(digits, sortDecreasing=TRUE)
{
    returnValue <- sort(digits, decreasing=sortDecreasing) |>
        paste0(collapse="") |>
        as.integer()

    return(returnValue)
}

main <- function(num, width)
{
    format <- sprintf("%s0%0.f.0f", "%", width)

    subtractionFormat <- sprintf("%s %s - %s = %s",
                                 "(%3.0f.)", format, format, format)

    previousValues <- c()

    counter <- 0
    repeat
    {
        counter %+=% 1

        digits <- getDigits(num, format)

        minuend <- orderDigits (digits, sortDecreasing=TRUE)

        subtrahend <- orderDigits (digits, sortDecreasing=FALSE)

        num <- minuend - subtrahend

        print(sprintf(subtractionFormat, counter, minuend, subtrahend, num))

        if (num %in% previousValues)
        {
```

```
                break
            }

            previousValues <- c(previousValues, num)
        }

    first <- which(previousValues == num)
    cycle <- length(previousValues) - first + 1

    print(sprintf("Cycle starts at %.0f and is %0.f operations long.", first,
        cycle))
    print("The program has ended.")
}

## https://en.wikipedia.org/wiki/6174
initialValue <- 1549
width <- 4

main(initialValue, width)

initialValue <- 2
width <- 2

main(initialValue, width)
```

Listing 4: Extracting the number of operations to compute a repeating value (see Algorithm 4) (R script: kaprekar-where.R).

By keeping track of the order of computing the potential constant, we can extract additional information about how Kaprekar's routine works. The two parts of the routine are delimited by where the previously computed value is detected. We can call part of the routine the "head" and the second part the "cycle." So, all Kaprekar searches have a head part that is some number of operations long until the previous value is computed. When the previous value is initially computed, it is not known that it will be computed a second time. When a previously computed value is again computed, then the routine is "stuck" in a cycle forever. When the cycle is detected, then the number of operations to compute the value can be computed, and the number of operations in the cycle can be computed. Running the script with the default values (see Listing 3), results in this output:

```
[1] "(  1.) 9541 - 1459 = 8082"
[1] "(  2.) 8820 - 0288 = 8532"
[1] "(  3.) 8532 - 2358 = 6174"
[1] "(  4.) 7641 - 1467 = 6174"
[1] "Cycle starts at 3 and is 1 operations long."
[1] "The program has ended."
[1] "(  1.) 20 - 02 = 18"
[1] "(  2.) 81 - 18 = 63"
[1] "(  3.) 63 - 36 = 27"
[1] "(  4.) 72 - 27 = 45"
[1] "(  5.) 54 - 45 = 09"
[1] "(  6.) 90 - 09 = 81"
[1] "(  7.) 81 - 18 = 63"
[1] "Cycle starts at 2 and is 5 operations long."
[1] "The program has ended."
```

The first block is uses the default values from [3]. The second block comes from running the same routines, only changing the initial value, and the width of all digits. Different values for initial value and width will result in different numbers of operations in the “head” and “cycle.” (The operations to search for the previously computed values can be thought of as a directed graph, with a cyclic component. Computed values could be considered as nodes, and operations as the arc between nodes. The length difference between nodes and arcs is one.)

```r
## https://stackoverflow.com/questions/64378066/how-can-i-convert-between-numeral-systems-in-r

glyphs <- function()
{
    return(c(0:9, LETTERS))
}

base <- function(b, currentBase = 10)
{
    currentBase <- as.integer(currentBase)

    if((currentBase < 2) | (currentBase > 36)) stop("'currentBase' must be between 2 and 36.")

    structure(lapply(b, function(x)
    {
        n   <- ceiling(log(x, currentBase))
        vec <- numeric()
        val <- x
        if (is.infinite(n) == FALSE)
        {
            while(n >= 0){
                rem <- val %/% currentBase^n
                val <- val - rem * currentBase^n
                vec <- c(vec, rem)
                n <- n - 1
            }
            while(vec[1] == 0 & length(vec) > 1) vec <- vec[-1]
        }
        else
        {
            vec <- c(0)
        }

        structure(x, base = currentBase, representation = vec)
    }), class = "base")
}

format.base <- function(b, ...)
{
    sapply(b, function(x)
    {
        localGlyphs <- glyphs()
        base    <- attr(x, "base")
        vec     <- attr(x, "representation")
        paste0(localGlyphs[vec + 1], collapse = "")
    })
```

```
47 }
48
49 print.base <- function(b, ...) print(format(b), quote = FALSE)
50
51 Ops.base <- function(e1, e2) {
52     base <- attr(e1[[1]], "base")
53     e1   <- unlist(e1)
54     e2   <- unlist(e2)
55     base(NextMethod(.Generic), base)
56 }
57
58 Math.base <- function(e1, e2) {
59     base <- attr(e1[[1]], "base")
60     e1   <- unlist(e1)
61     e2   <- unlist(e2)
62     base(NextMethod(.Generic), base)
63 }
64
65 as.data.frame.base <- function(b, ...)
66 {
67     structure(list(b),
68               class = "data.frame",
69               row.names = seq_along(b))
70 }
71
72 zeroPadd <- function(n, width)
73 {
74     returnValue <- sprintf("%s",format(n))
75
76     padding <- ""
77
78     if(nchar(returnValue) < width)
79     {
80         padding <- paste0(rep("0", (width-nchar(returnValue))), collapse="")
81     }
82
83     returnValue <- paste0(padding, returnValue, collapse="")
84
85     return(returnValue)
86 }
87
88 sortAndSubtract <- function(n, width , counter)
89 {
90     localBase <- function(string, orig)
91     {
92         return(base(strtoi(string, attr(orig[[1]], "base")),attr(orig[[1]], "
              base")))
93     }
94
```

```
95     localWorker <- function(orig, wid, ascend)
96     {
97         temp <- orig |>
98             zeroPadd(wid) |>
99             splitAndOrder(ascend) |>
100            localBase(orig)
101        return(temp)
102    }
103
104    minuend <- localWorker(n, width, FALSE)
105    subtrahend <- localWorker(n, width, TRUE)
106
107    returnValue <- (base(minuend - subtrahend, attr(n[[1]], "base")))
108
109    format <- sprintf("%s0%0.fs", "%", width)
110
111    subtractionFormat <- sprintf("%s %s - %s = %s",
112                                 "(%3.0f.)", format, format, format)
113
114    print(sprintf(subtractionFormat, counter, format(minuend),format(
           subtrahend), format(returnValue)))
115
116    return(returnValue)
117 }
118
119 splitAndOrder <- function(s, ascending=TRUE)
120 {
121     returnValue <- strsplit(s, "")[[1]] |>
122         sort(decreasing=!ascending) |>
123         paste0(collapse="")
124
125     return(returnValue)
126 }
```

Listing 5: A support library used when computing numeric bases other than 10. This file is needed when exploring with Kaprekar with different bases (R script: basesLibrary.R).

One of the assumptions in all previous Kaprekar explorations is that the numeric operations are carried out using the decimal numeric system. This leads to the question, do the same kinds of operations lead to a numeric constant using different numeric bases? This library (see Listing 5) is a collection of numeric base routines (and a few other functions) to enable exploration in these other bases. This is a catalog of those functions:

- **as.data.frame.base** – optional, useful to create a data frame based in class “base” structures
- **base** – takes a decimal value and a restricted base value, and returns an structure of class “base”, whose contents are: (1) the number in decimal format, (2) the desired base, and (3) the representation of the decimal value in the desired base.

- **format.base –** required to show the class "base" structure in the desired representation
- **glyphs** – a consistent list of glyphs to represent digits in different bases
- **Math.base** – required to allow decimal operations on class "base" structures
- **Ops.base –** required to allow decimal operations on class "base" structures
- **print.base** – required to print the class "base" structure
- **sortAndSubtract –** performs one Kaprekar operation on two class "base" structures
- **splitAndOrder –** a utility function that comes in handy from time to time
- **zeroPad –** leading 0 pads class "base" representations to the desired width

```
rm(list=ls())

source("basesLibrary.R")

`%+=%` = function(e1,e2) eval.parent(substitute(e1 <- (e1) + (e2)))

getDigits <- function(number, format)
{
    singletons <- sprintf(format, number) |>
        strsplit("") |>
        unlist() |>
        sort()
    return(singletons)
}

main <- function(num, width, currentBase)
{
    previousValues <- c()

    counter <- 0

    print(sprintf("Evaluating: %s base %.0f",
                            zeroPadd(num, width),
                    currentBase))

    num <- num |>
        strtoi (currentBase) |>
        base(currentBase)

    repeat
    {
        counter %+=% 1

        num <- sortAndSubtract(num, width, counter)

        if (unlist(num) %in% previousValues)
        {
            break
        }

        previousValues <- c(previousValues, unlist(num))
    }

    first <- which(previousValues == unlist(num))
    cycle <- length(previousValues) - first + 1

    print(sprintf("Cycle starts at %.0f and is %0.f operations long.", first,
        cycle))
```

```
48      print("The program has ended.")
49  }
50
51  ## https://en.wikipedia.org/wiki/6174
52  initialValue <- 1549
53  width <- 4
54  baseOfInterest <- 10
55
56  main(initialValue, width, baseOfInterest)
57
58  initialValue <- 1549
59  width <- 4
60  baseOfInterest <- 16
61
62  main(initialValue, width, baseOfInterest)
63
64  initialValue <- 1549
65  width <- 4
66  baseOfInterest <- 23
67
68  main(initialValue, width, baseOfInterest)
```

Listing 6: Exploring Kaprekar's routine using different numeric bases (see Algorithm 5) (R script: kaprekar-bases.R).

We have established that we can find previously computed values based on different initial values and different numeric widths, now the question becomes does the same type of behavior (the idea of "head" and "cycle") apply to numeric bases other than decimal. Running the script with the default values (see Listing 6), results in this output:

```
[1] "Evaluating: 1549 base 10"
[1] "(  1.) 9541 - 1459 = 8082"
[1] "(  2.) 8820 -  288 = 8532"
[1] "(  3.) 8532 - 2358 = 6174"
[1] "(  4.) 7641 - 1467 = 6174"
[1] "Cycle starts at 3 and is 1 operations long."
[1] "The program has ended."
[1] "Evaluating: 1549 base 16"
[1] "(  1.) 9541 - 1459 = 80E8"
[1] "(  2.) E880 -  88E = DFF2"
[1] "(  3.) FFD2 - 2DFF = D1D3"
[1] "(  4.) DD31 - 13DD = C954"
[1] "(  5.) C954 - 459C = 83B8"
[1] "(  6.) B883 - 388B = 7FF8"
[1] "(  7.) FF87 - 78FF = 8688"
[1] "(  8.) 8886 - 6888 = 1FFE"
[1] "(  9.) FFE1 - 1EFF = E0E2"
[1] "( 10.) EE20 -  2EE = EB32"
[1] "( 11.) EB32 - 23BE = C774"
[1] "( 12.) C774 - 477C = 7FF8"
[1] "Cycle starts at 6 and is 6 operations long."
[1] "The program has ended."
[1] "Evaluating: 1549 base 23"
[1] "(  1.) 9541 - 1459 = 80LF"
[1] "(  2.) LF80 -  8FL = L6F2"
[1] "(  3.) LF62 - 26FL = J8D4"
[1] "(  4.) JD84 - 48DJ = F4H8"
[1] "(  5.) HF84 - 48FH = D6FA"
[1] "(  6.) FDA6 - 6ADF = 92JE"
[1] "(  7.) JE92 - 29EJ = H4H6"
[1] "(  8.) HH64 - 46HH = DABA"
[1] "(  9.) DBAA - AABD = 30LK"
[1] "( 10.) LK30 -  3KL = LG52"
[1] "( 11.) LG52 - 25GL = JAB4"
[1] "( 12.) JBA4 - 4ABJ = F0L8"
[1] "( 13.) LF80 -  8FL = L6F2"
[1] "Cycle starts at 2 and is 11 operations long."
[1] "The program has ended."
```

The first block is the unknown Kaprekar constant search using a decimal value and decimal base from [3]. The second block simply changes the base from decimal to hexadecimal. The third

block is exploring the same initial value, width, but changing the base to 23 (because that is a nice base).

```r
rm(list=ls())

main <- function(width=3)
{
    format <- sprintf("%s0%.0f.0f", "%", width)

    number <- 0
    maxValue <- 10^width - 1

    tempFile <- tempfile()

    if (file.exists(tempFile) == TRUE)
    {
        file.remove(tempFile)
    }

    for (number in 0:maxValue)
    {
        value <- sprintf(format, number)

        temp <- strsplit(value, "")[[1]] |>
            sort() |>
            paste(collapse="") |>
            as.numeric()

        temp <- sprintf(format, temp)

        write(temp, tempFile, append=TRUE)
    }

    command <- sprintf("sort %s | uniq", tempFile)

    results <- system(command, intern=TRUE)

    print(sprintf("Width = %.0f and generated %s unique string combinations.",
                  width, formatC(length(results), big.mark=",", format="d")))

    print(sprintf("Vice %s possible values, a reduction of %.1f%s.",
                  formatC(maxValue + 1, big.mark=",", format="d"),
                  (1 - length(results)/(maxValue + 1)) * 100,
                  "%"))

    print("The program has ended.")
    invisible(results)
}


widths <- c(3, 4, 5, 6)
```

```
49
50 for (width in widths)
51 {
52     d <- main(width)
53 }
54
55 uniqueValues <- c(220, 715, 2002, 5005)
56
57 plot(widths, uniqueValues,
58      log="y",
59      xlab="Width of numeric values",
60      ylab="Log scale of number of unique values")
```

Listing 7: Computing the number of unique values after Kaprekar's first step (see Algorithm 6) (R script: uniqueNumbers.R).

A demonstration program shows the effect of reducing the number of Kaprekar values to consider (see Algorithm 6). In the "classic" Kaprekar routine, a random 4 digit number is selected, and then the individual digits of the are sorted in different ways. This separation and sorting operation reduces the number of 4 digit values considerably. A change in the width of allowable number of 1 (from 2 to 3, or from 4 to 5, etc.), changes the possible number of valid values by a factor of 10. Exploring the Kaprekar world of 2 to 4 digits wide is trivial. Exploring the Kaprekar world of 7 or more digits permits many, many more valid numbers. If we can remove those numeric values that result in the same values after the first Kaprekar step, then we can greatly reduce the number of values to consider. This reduction will help us identify what the "constants" are, what the "head" and "cycle" lengths are, but will not provide us with the frequency of each of these values. To get the frequency of these values, we have to evaluate all possible values (or unmap the uniques strings).

Running the script with the default values (see Listing 7), results in this output:

```
[1] "Width = 3 and generated 220 unique string combinations."
[1] "Vice 1,000 possible values, a reduction of 78.0%."
[1] "The program has ended."
[1] "Width = 4 and generated 715 unique string combinations."
[1] "Vice 10,000 possible values, a reduction of 92.8%."
[1] "The program has ended."
[1] "Width = 5 and generated 2,002 unique string combinations."
[1] "Vice 100,000 possible values, a reduction of 98.0%."
[1] "The program has ended."
[1] "Width = 6 and generated 5,005 unique string combinations."
[1] "Vice 1,000,000 possible values, a reduction of 99.5%."
[1] "The program has ended."
```

Not all of these unique string combinations are valid inputs to the Kaprekar routine. Each width contains strings that are all of the same character, for example: "000", "111", "3333", "777777" So

the number of strings that are valid inputs to Kaprekar is actually 10 less than the number reported. Being able to reduce the number of strings to be considered greatly reduces the script execution time, while sacrificing frequency accuracy.

A variation of this script incorporating the idea of different numeric bases is used in later scripts.

```
source("kaprekar-parallel.R")

getData <- function(fileName)
{
    localGetNumericField <- function(string, field)
    {
        temp <- gsub('"', "", string)
        temp <- gsub(',', "", temp)
        temp <- unlist(lapply(strsplit(temp, " ")[[1]], as.numeric))
        indices <- which(is.na(temp) == TRUE)
        temp <- temp[-indices]
        return(temp[field])
    }


    lines <- readLines(fileName)

    cores <- c()
    widths <- c()
    values <- c()
    prepare <- c()
    process <- c()

    i <- 1

    while (i < length(lines))
    {
        cores <- c(cores, localGetNumericField(lines[i], 1))
        i %+=% 1
        widths <- c(widths, localGetNumericField(lines[i], 1))
        i %+=% 1
        values <- c(values, localGetNumericField(lines[i], 1))
        i %+=% 1
        while (grepl("Time to prepare data", lines[i], fixed=TRUE) == FALSE)
        {
            i %+=% 1
        }
        i %+=% 1
        i %+=% 1
        prepare <- c(prepare, localGetNumericField(lines[i], 3))
        i %+=% 1
        i %+=% 1
        i %+=% 1
        process <- c(process, localGetNumericField(lines[i], 3))
        i %+=% 1
        i %+=% 1
    }

```

```
49      returnValue <- data.frame(cores=cores,
50                                width=widths,
51                                values=values,
52                                prepare=prepare,
53                                process=process)
54
55      return(returnValue)
56  }
57
58  addLegend <- function(d)
59  {
60      legend("topleft",
61             title="Number of\ncores (threads) used",
62             legend=unique(d$cores),
63             pch=unique(d$cores),
64             col=unique(d$cores),
65             )
66  }
67
68  plotPoints <- function(d, col)
69  {
70      for (core in unique(d$cores))
71      {
72          indices <- which(d$cores == core)
73          points(d$width[indices],
74                 d[[col]][indices],
75                 pch=core,
76                 type="b",
77                 col=core
78                 )
79      }
80  }
81
82  normalizeTimes <- function(d, dest, src)
83  {
84      d[[dest]] <- 0
85
86      for (width in unique(d$width))
87      {
88          indices <- which(d$width == width)
89          baseTime <- min(d[[src]][indices])
90          d[[dest]][indices] <- d[[src]][indices]/baseTime
91      }
92
93      return(d)
94  }
95
96  addCoreHighlights <- function(d, col)
97  {
```

```
98      for (core in unique(d$cores))
99      {
100         indicesCore <- which(d$cores == core)
101         minValue <- min(d[[col]][indicesCore])
102         indicesValue <- which(d[[col]] == minValue)
103         index <- intersect(indicesCore, indicesValue)
104         points(d$width[index], c(rep(minValue,length(index))),
105                pch=core,
106                col=core,
107                cex=3)
108     }
109 }
110
111 plotResults <- function(d)
112 {
113     oldPar <- par(no.readonly = TRUE)
114
115     par(mfrow=c(2,2))
116
117     plot(d$width, d$prepare, pch=d$cores,
118          type="n",
119          log="y",
120          xlab="Numeric width",
121          ylab="Seconds to prepare data (log scale)")
122
123     plotPoints(d, "prepare")
124
125     addLegend(d)
126
127     plot(d$width, d$process, pch=d$cores,
128          type="n",
129          log="y",
130          xlab="Numeric width",
131          ylab="Seconds to process data (log scale)")
132
133     plotPoints(d, "process")
134
135     addLegend(d)
136
137     col <- "prepareNormalized"
138
139     plot(d$width, d[[col]],
140          type="n",
141          xlab="Numeric width",
142          ylab="Process preparation time normalized to lowest time per width")
143
144     plotPoints(d, col)
145
146     addCoreHighlights(d, col)
```

```
147
148      addLegend(d)
149
150      col <- "processNormalized"
151      plot(d$width, d[[col]],
152            type="n",
153            xlab="Numeric width",
154            ylab="Process execution time normalized to lowest time per width")
155
156      plotPoints(d, col)
157
158      addCoreHighlights(d, col)
159
160      addLegend(d)
161
162      par(oldPar)
163  }
164
165
166  driveMain <- function(uniqueValuesOnly=TRUE)
167  {
168      captureFile <- "/tmp/capture.txt"
169
170      if(file.exists(captureFile) == TRUE)
171      {
172          file.remove(captureFile)
173      }
174
175      for (width in 2:8)
176      {
177          for (numberOfCores in c(1, 2, 4, 8, 16))
178          {
179              capture.output(
180                  main(width, numberOfCores, uniqueValuesOnly),
181                  file=captureFile,
182                  append=TRUE)
183          }
184      }
185
186      d <- getData(captureFile)
187
188      d <- normalizeTimes(d, "processNormalized", "process")
189      d <- normalizeTimes(d, "prepareNormalized", "prepare")
190
191      plotResults(d)
192
193      invisible(d)
194  }
195
```

```
196 d <- driveMain()
```

Listing 8: Comparing the parallel computation times Kaprekar's constant (see Algorithm 8) (R script: kaprekar-parallel-stats.R).

The listing (see Listing 8) can be logically divided into two parts:

1. *driveMain* – to drive the *main* function of *kaprekar-parallel.R*. (*kaprekar-parallel.R* and it's associated library are too long to include as listings here. They are embedded in (see Section 8).) The function has two nested loops, one to vary the width, and the other to vary the number of cores used. The output of *kaprekar-parallel.R::main* is captured to a file.

2. *plotResults* – takes the strings from the capture file and creates a series of plots to support performance analysis.

Running the script results in a data frame that looks like this:

	cores	width	values	prepare	process	processNormalized	prepareNormalized
1	1	2	45	0.181	0.056	1.076923	1.016854
2	2	2	45	0.178	0.052	1.000000	1.000000
3	4	2	45	0.187	0.090	1.730769	1.050562
4	8	2	45	0.238	0.091	1.750000	1.337079
5	16	2	45	0.334	0.119	2.288462	1.876404
6	1	3	210	0.189	0.169	2.485294	1.016129
7	2	3	210	0.186	0.068	1.000000	1.000000
8	4	3	210	0.191	0.079	1.161765	1.026882
9	8	3	210	0.242	0.091	1.338235	1.301075
10	16	3	210	0.335	0.132	1.941176	1.801075
11	1	4	705	0.401	0.274	2.157480	1.382759
12	2	4	705	0.319	0.192	1.511811	1.100000
13	4	4	705	0.302	0.145	1.141732	1.041379
14	8	4	705	0.290	0.127	1.000000	1.000000
15	16	4	705	0.379	0.131	1.031496	1.306897
16	1	5	1992	2.229	1.176	3.818182	3.566400
17	2	5	1992	1.298	0.694	2.253247	2.076800
18	4	5	1992	0.838	0.409	1.327922	1.340800
19	8	5	1992	0.625	0.308	1.000000	1.000000
20	16	5	1992	0.689	0.314	1.019481	1.102400
21	1	6	4995	23.042	11.112	2.802522	5.479667
22	2	6	4995	12.288	6.124	1.544515	2.922235
23	4	6	4995	7.174	4.059	1.023707	1.706064
24	8	6	4995	4.714	3.965	1.000000	1.121046
25	16	6	4995	4.205	4.470	1.127364	1.000000
26	1	7	11430	388.238	334.499	3.733123	7.628665
27	2	7	11430	182.821	171.476	1.913731	3.592333
28	4	7	11430	90.712	101.036	1.127596	1.782441
29	8	7	11430	59.595	89.603	1.000000	1.171009
30	16	7	11430	50.892	98.014	1.093870	1.000000
31	1	8	24300	6378.867	7063.802	3.659876	7.259007
32	2	8	24300	3459.591	3632.498	1.882059	3.936937
33	4	8	24300	1729.753	2118.819	1.097796	1.968420
34	8	8	24300	1088.653	1930.066	1.000000	1.238863
35	16	8	24300	878.752	2005.337	1.038999	1.000000

Where:

- **cores –** the number of cores in the cluster used to search Kaprekar space.
- **width –** the width of acceptable numbers.

- **values** – the number of values evaluated, in this case only unique values were considered vice all possible values.
- **prepare** – the number of seconds used to compute the values to be used for exploration.
- **process** – the number of seconds used to explore the Kaprekar space with the number of cores and values of interest.
- **processNormalized** – for each width block, process times normalized to the shortest time in the block.
- **prepareNormalized** – for each width block, data preparation times normalized to the shortest time in the block.

```r
rm(list=ls())

library(combinat)

main <- function(string)
{
    characters <- string |>
        strsplit("") |>
        unlist()

    permutedSet <- characters |>
        permn() |>
        lapply (FUN=function(x)paste(x,collapse="")) |>
        unlist() |>
        unique()

    invisible(permutedSet)
}

## https://en.wikipedia.org/wiki/6174

value <- "4195"

d <- main(value)
```

Listing 9: Reconstructing original values from a Kaprekar mapped value (see Algorithm 7) (R script: recreateNumbers.R).

The script (see Listing 9) takes in a string, splits the string into its individual characters, permutes that set of characters, collapses those permutations, and returns a sorted list of unique permutations. Running the script results in and evaluating the variable *d* results in:

```
d
 [1] "4195" "4159" "4519" "5419" "5491" "4591" "4951" "4915" "9415" "9451"
[11] "9541" "5941" "5914" "9514" "9154" "9145" "1945" "1954" "1594" "5194"
[21] "5149" "1549" "1459" "1495"
```

In this trivial case, the returned list can be converted to a numeric. If the initial string was non-decimal, then the conversion may be more challenging.

5 Results

Exploration of the Kaprekar world is facilitated by collecting the same type of data based on different numeric "widths" and reporting results in tabular and graphic formats. Driver scripts were created to "drive" the scripts (see Listing 8). The driver script would vary the arguments to a

particular *main* function, collect the resulting output, and format the results in a consistent manner. By default, only unique values were processed. For lower values of *width*, the code was manually modified to process all values.

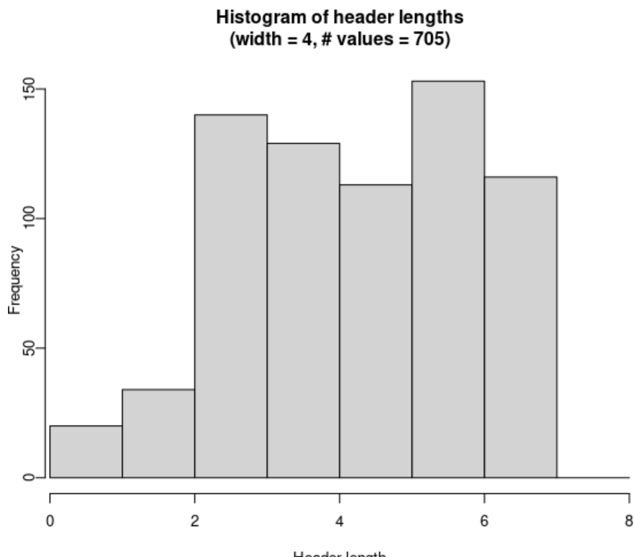
5.1 Overview

These are sample data where $width = 4$ and all input values are evaluated.

```
              Cycle-length
Header length   1
            1  20
            2  34
            3 140
            4 129
            5 113
            6 153
            7 116
```

This shows that the cycle length was 1, and there were between 1 and 7 header operations to get to the cycle value. The tabular data can also be presented in a graphical format (see Figure 1).

Figure 1: Sample "classic" Kaprekar header length histogram.

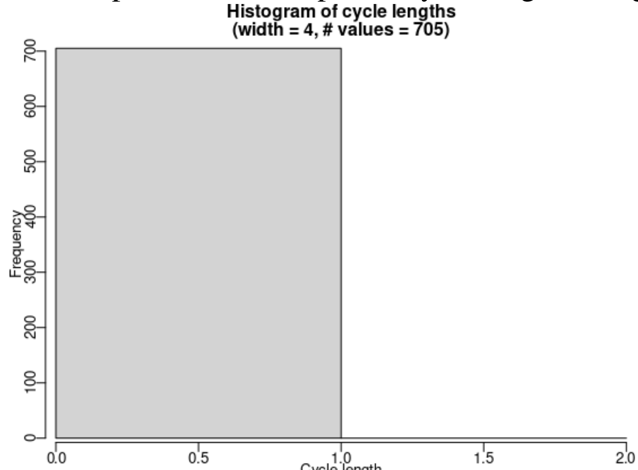


This shows that the cycle length was 1, the only cycle value was 6174, and there were 705 instances where that occurred. The tabular data can also be presented in a graphical format (see Figure 2).

```
              Cycle-length
Cycle-value   1
       6174 705
```

The tabular data can also be presented in a graphical format (see Figure 2). While the histograms

Figure 2: Sample “classic” Kaprekar cycle length histogram.



of the cycle and header lengths can provide some in sight, combing the two related values into on plot may provide a more holistic view (see Figure 3).

The “classic” Kaprekar repeat value is 6174 (see Figure 4). While Kaprekar classic repeats the same value after a maximum of 7 header operations, other width values result in other repeating values.

For ease of presentation, the images referenced in this section (see Figures 1 through 4) may be combined into one image per width.

Figure 3: Sample "classic" Kaprekar cycle and header length frequency plot. The size of the plotted circles represents the relative number of times a particular header and cycle length occurred. The larger the circle, the more often those two values occurred together.

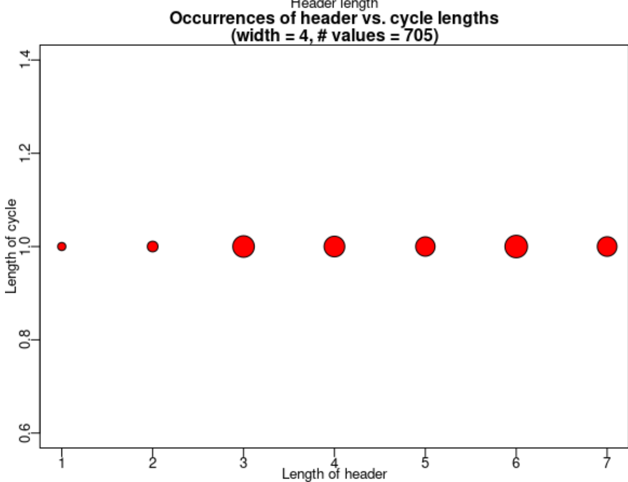
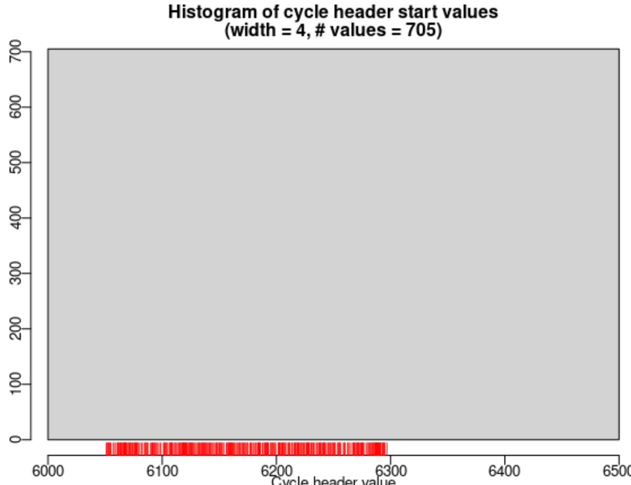


Figure 4: Sample "classic" Kaprekar repeat value. The rug represents the approximate value that was repeated. If the absolute value was used, then repeated values would plot directly on top of each other and it wouldn't be possible to see if there was more than one value at any particular location.



5.2 Width = 2

See the termination statement.

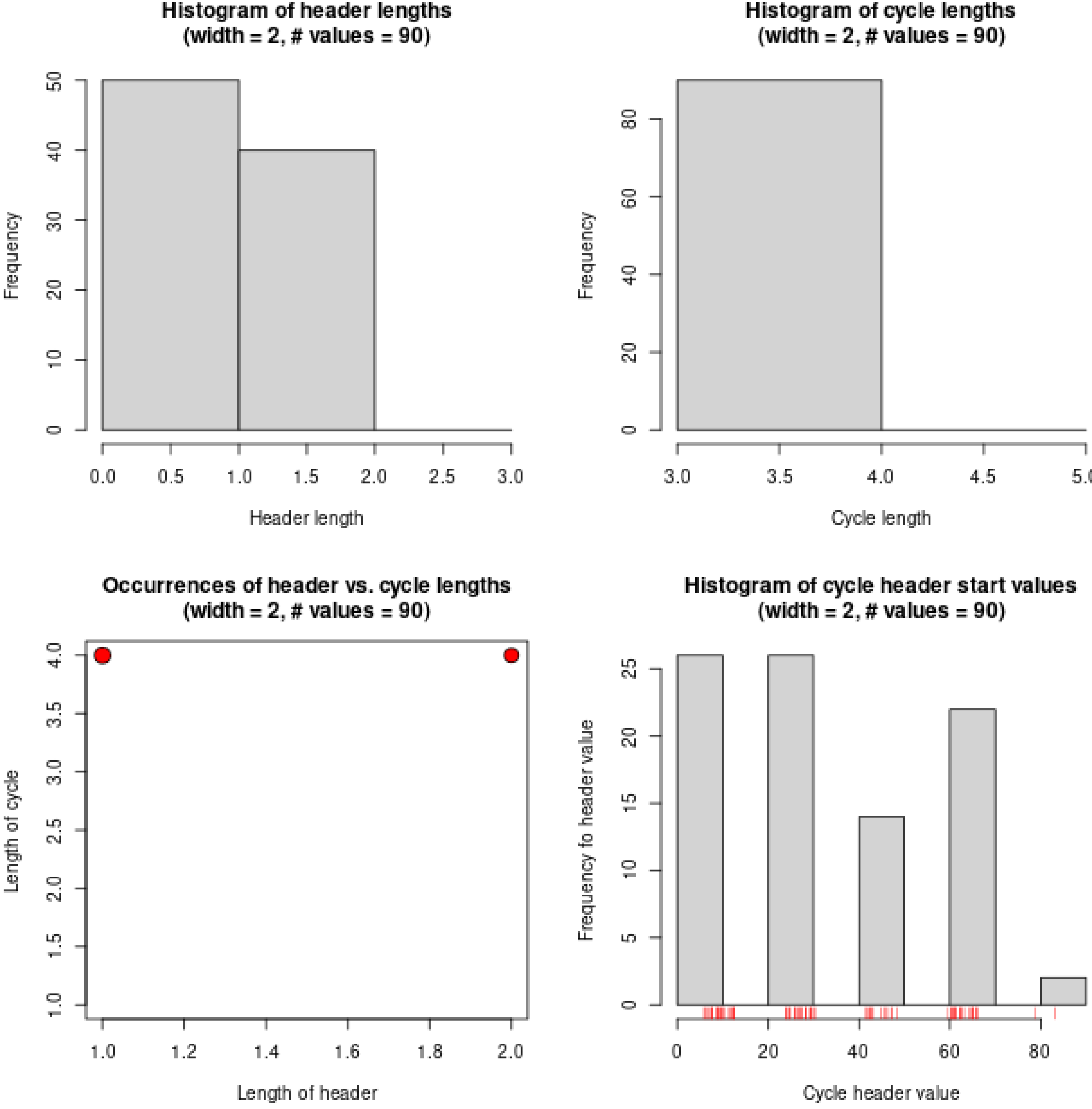
Processing all values was deemed as taking too long (see Section 5.6). Therefore, there are two sets of results for this width. The first is using all values, while the second is using only unique values. It is possible to expand the unique values to be all inclusive and is left as "an exercise for the student."

All possible values allowed by the width value were processed. In all cases the cycle length is 4 operations and the header length is only 1 or 2. So, very short headers, and small cycles. There are several cycle values (values that are computed twice). The combined data are presented in (see Figure 5).

```
            Cycle-length
Cycle-value  4
          9  26
         27  26
         45  14
         63  22
         81   2
```

```
              Cycle-length
Header length  4
            1 50
            2 40
```

Figure 5: Width = 2, all values processed.

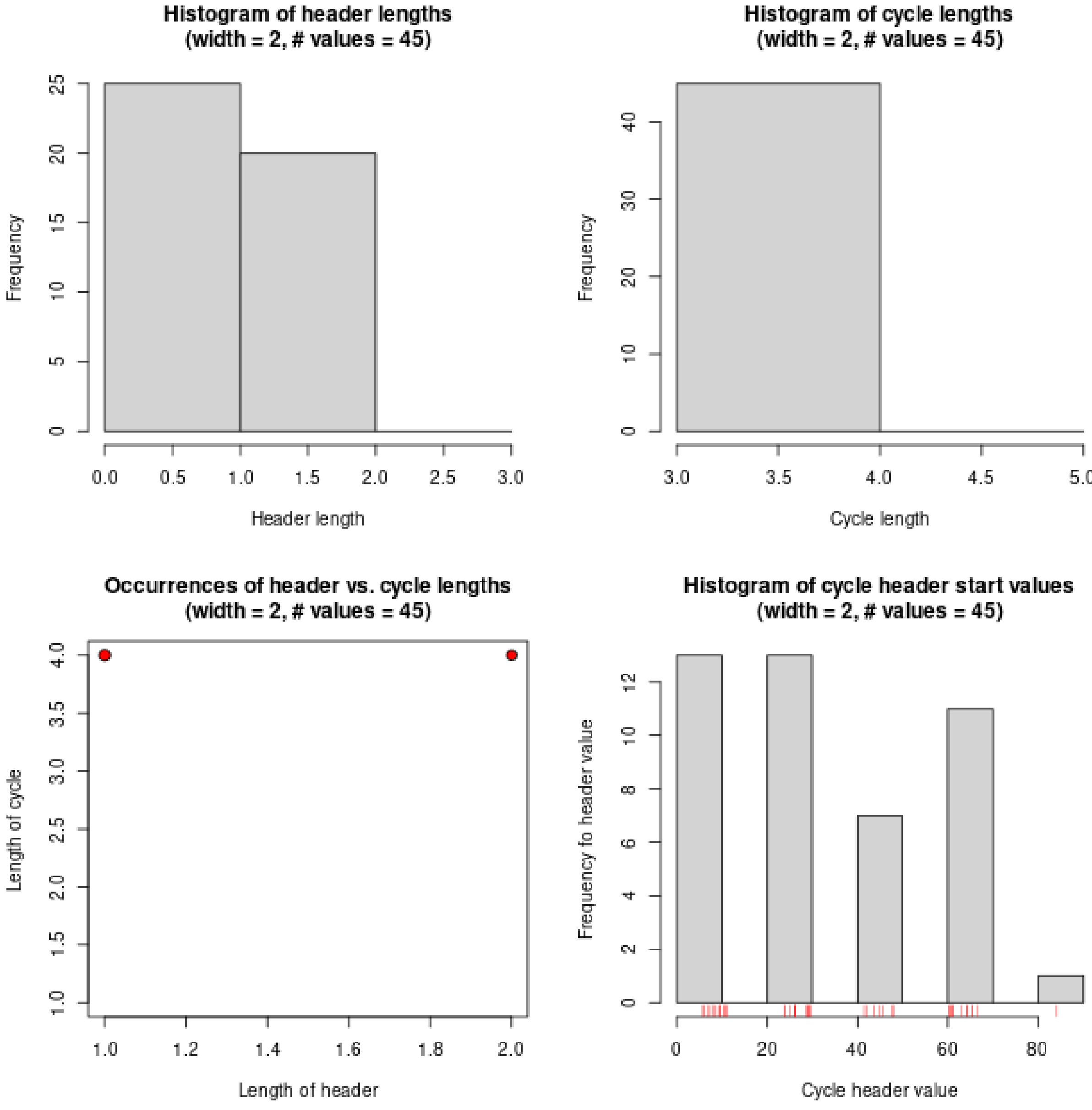


All possible values allowed by the width value were processed. In all cases the cycle length is 4 operations and the header length is only 1 or 2. So, very short headers, and small cycles. There are several cycle values (values that are computed twice). The combined data are presented in (see Figure 6).

	Cycle-length
Cycle-value	4
9	13
27	13
45	7
63	11
81	1

	Cycle-length
Header length	4
1	25
2	20

Figure 6: Width = 2, only unique values processed.



5.3 Width = 3

See the termination statement.

Processing all values was deemed as taking too long (see Section 5.6). Therefore, there are two sets of results for this width. The first is using all values, while the second is using only unique values. It is possible to expand the unique values to be all inclusive and is left as "an exercise for the student."

All possible values allowed by the width value were processed. In all cases the cycle length is 0 operations and the header length is varied from 1 to 6. In all cases, the cycle value was 495. So, short headers, and 0 length cycles. There are several cycle values (values that are computed twice). The combined data are presented in (see Figure 5). All possible values allowed by the width value were processed. In all cases the cycle length is 0 operations and the header length was between 1 and 6. So, very short headers, and small cycles. There was only one cycle value, 495. The combined data are presented in (see Figure 7).

```
            Cycle-length
Cycle-value    0
        495  990
```

```
              Cycle-length
Header length    0
            1  150
            2  144
            3  270
            4  222
            5  150
            6   54
```

Only unique values allowed by the width value were processed. In all cases the cycle length is 0 operations and the header length was between 1 and 7. So, very short headers, and small cycles. There was only one cycle value, 495. The combined data are presented in (see Figure 8).

```
            Cycle-length
Cycle-value    0
        495  210
```

```
              Cycle-length
Header length  0
            1 30
            2 28
            3 54
            4 46
            5 34
            6 18
```

Figure 7: Width = 3, all values processed.

Figure 8: Width = 3, only unique values processed.

5.4 Width = 4

See the termination statement.

Processing all values was deemed as taking too long (see Section 5.6). Therefore, there are two sets of results for this width. The first is using all values, while the second is using only unique values. It is possible to expand the unique values to be all inclusive and is left as “an exercise for the student.”

All possible values allowed by the width value were processed. In all cases the cycle length is 0 operations and the header length is between 1 and 7 operations. So, very short headers, and small cycles. There was only one cycle value, 6174. The combined data are presented in (see Figure 9).

```
             Cycle-length
Cycle-value     0
       6174  9990
```

```
             Cycle-length
Header length    0
            1  384
            2  576
            3 2400
            4 1272
            5 1518
            6 1656
            7 2184
```

Only unique possible values allowed by the width value were processed. In all cases the cycle length is 0 operations and the header length is between 1 and 7 operations. So, very short headers, and small cycles. There was only one cycle value, 6174. The combined data are presented in (see Figure 10).

```
             Cycle-length
Cycle-value    0
       6174  705
```

```
             Cycle-length
Header length   0
            1  20
            2  34
            3 140
            4 129
            5 113
            6 153
            7 116
```

Figure 9: Width = 4, all values processed.

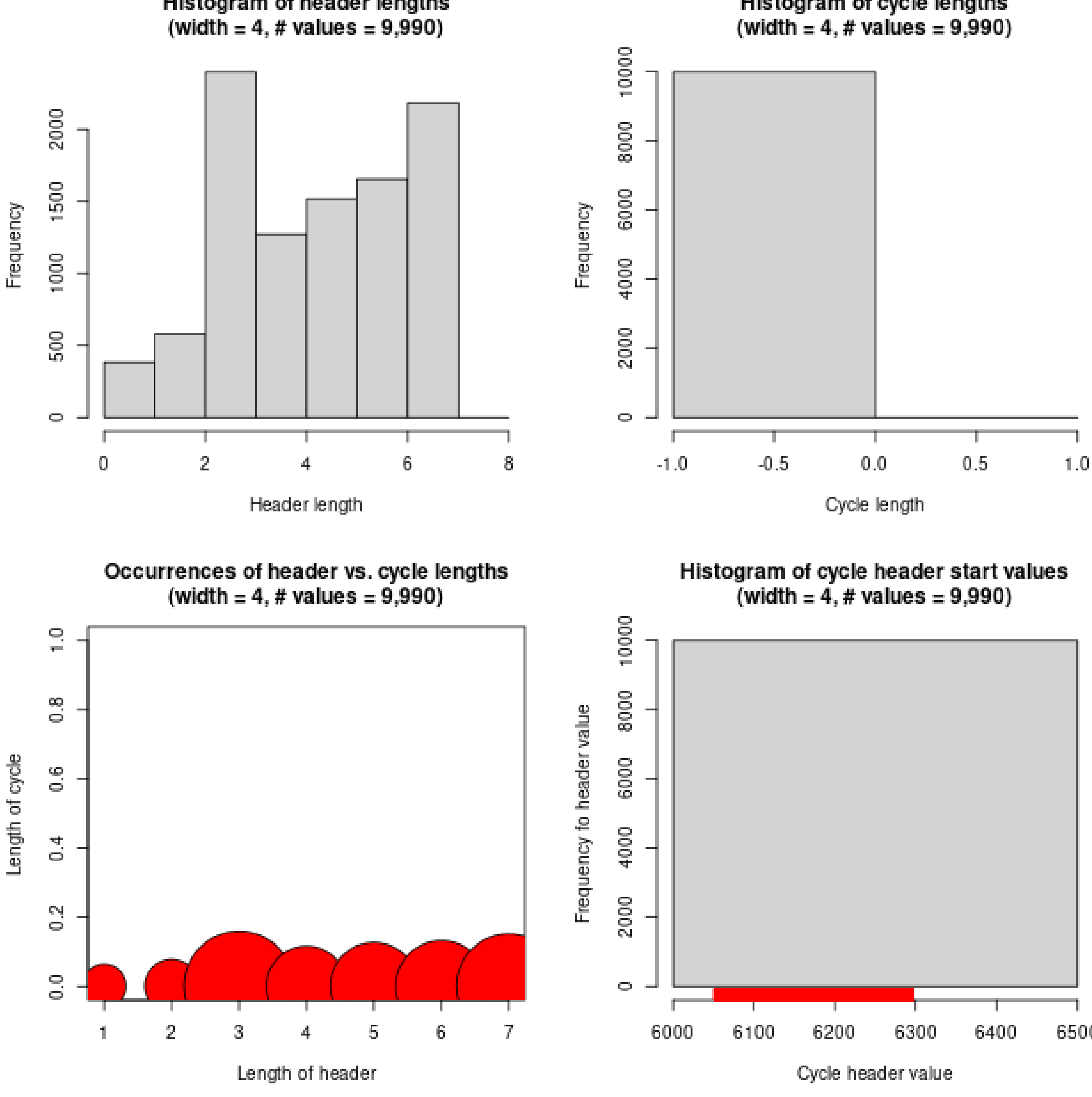


Figure 10: Width = 4, only unique values processed.

5.5 Width = 5

See the termination statement.

Processing all values was deemed as taking too long (see Section 5.6). Therefore, there are two sets of results for this width. The first is using all values, while the second is using only unique values. It is possible to expand the unique values to be all inclusive and is left as "an exercise for the student."

All possible values allowed by the width value were processed. In all cases the cycle length very short (1 or 3) operations and the header length is between 1 and 6 operations. So, very short headers, and small cycles. There are 10 cycle values. The combined data are presented in (see Figure 11).

	Cycle-length	
Cycle-value	1	3
53955	2750	0
59994	440	0
61974	0	5100
62964	0	5050
63954	0	26520
71973	0	6230
74943	0	31700
75933	0	10540
82962	0	6320
83952	0	5340

	Cycle-length	
Header length	1	3
1	2740	28300
2	450	17880
3	0	21000
4	0	17580
5	0	8450
6	0	3590

Only unique possible values allowed by the width value were processed. In all cases the cycle length very short (1 or 3) operations and the header length is between 1 and 6 operations. So, very short headers, and small cycles. There are 10 cycle values. The combined data are presented in (see Figure 12).

	Cycle-length	
Cycle-value	1	3
53955	80	0
59994	28	0
61974	0	118
62964	0	118
63954	0	454
71973	0	148
74943	0	646
75933	0	206
82962	0	102
83952	0	92

	Cycle-length	
Header length	1	3
1	78	432
2	30	436
3	0	354
4	0	342
5	0	242
6	0	78

Figure 11: Width = 5, all values processed.

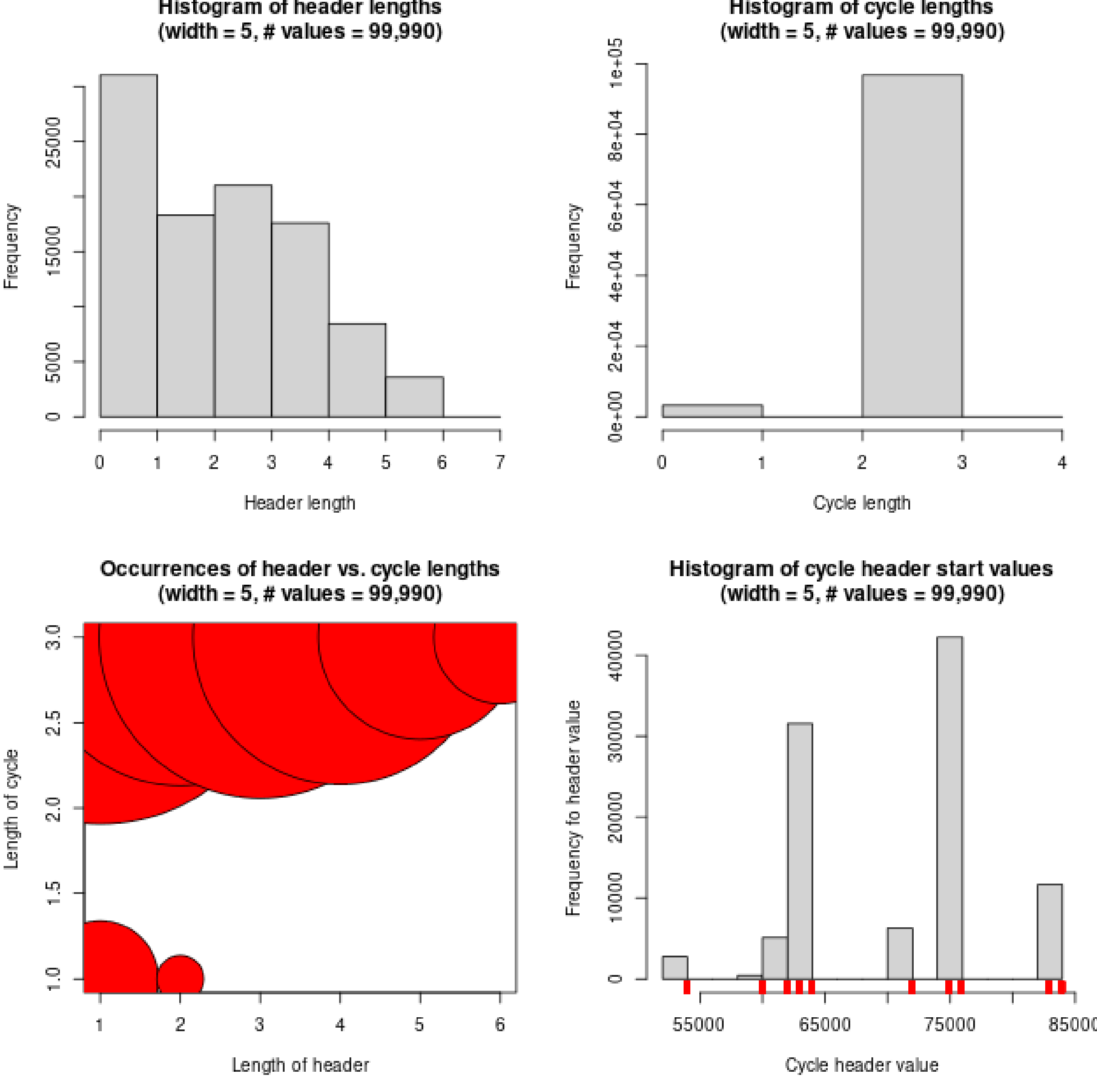
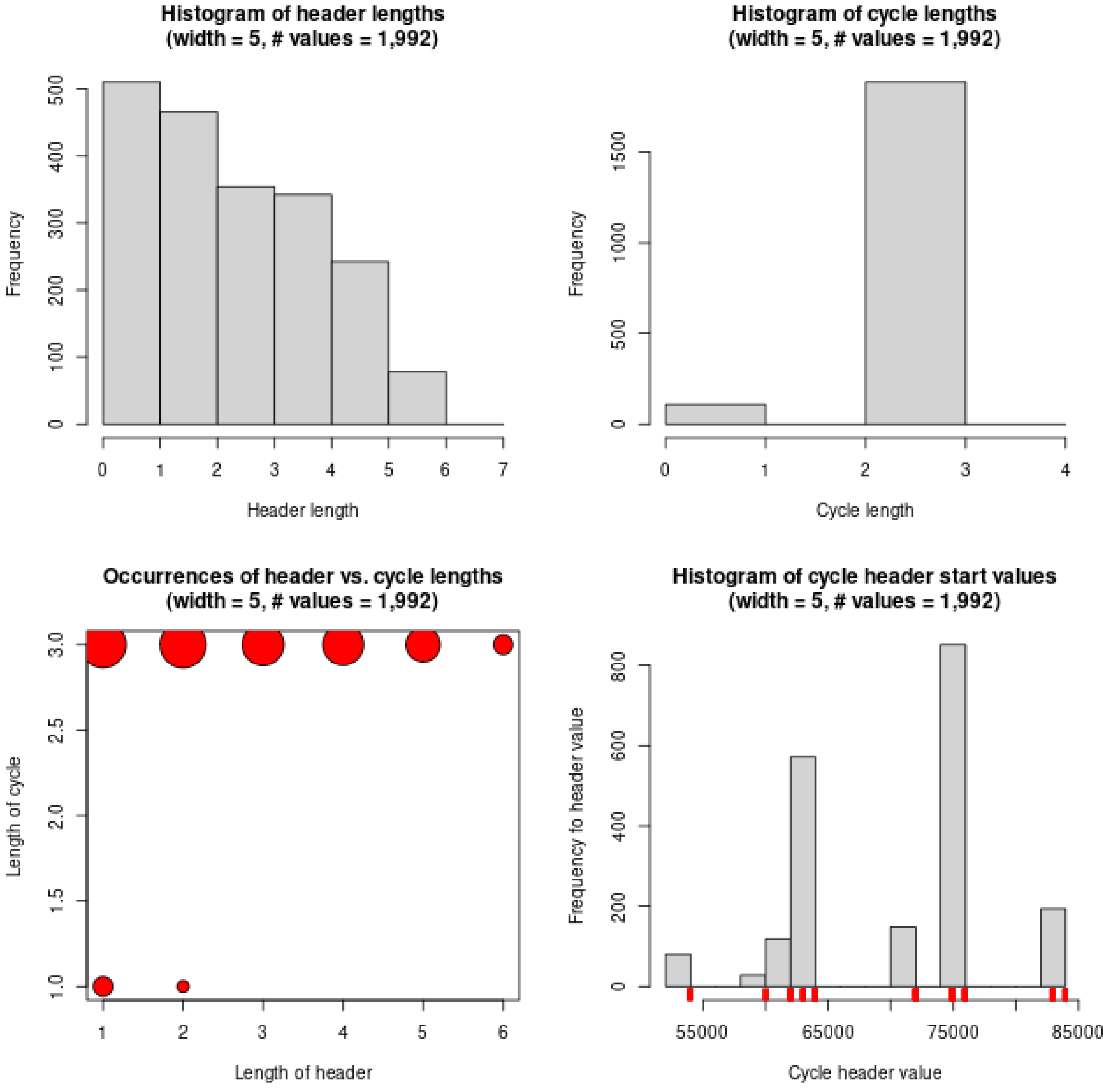


Figure 12: Width = 5, only unique values processed.



5.6 Width = 6

Program termination.

The R script has been running for approximately 60 hours processing the width equal 6 on a:

- AMD Ryzen 7 5800X 8-Core Processor
- 8 cores
- CPU clock speed 3800 MHz
- 32 GByte RAM
- Ubuntu 22.04.1

attempting to process all values. Results presented for widths 2 through 5 are based on all possible values, and unique values. Results for widths greater than 5 are based on unique values.

Only unique possible values allowed by the width value were processed. In all cases the cycle length very short (0 or 6) operations and the header length is between 1 and 13 operations. So, very short headers, and moderate length cycles. There are 9 cycle values. The combined data are presented in (see Figure 13).

```
             Cycle-length
Cycle-value     0    6
     420876     0 1839
     549945    30    0
     631764   352    0
     642654     0  126
     750843     0  135
     840852     0  225
     851742     0 1482
     860832     0  712
     862632     0   94
```

```
             Cycle-length
Header length    0    6
            1   62  237
            2   68  463
            3  228  610
            4   24  428
            5    0  521
            6    0  468
            7    0  398
            8    0  472
            9    0  379
           10    0  378
           11    0   88
           12    0  166
           13    0    5
```

Figure 13: Width = 6, only unique values processed.

5.7 Width = 7

Only unique possible values allowed by the width value were processed. In all cases the cycle length was 7 operations and the header length is between 1 and 13 operations. So, very short headers, and moderate length cycles. There are 8 cycle values. The combined data are presented in (see Figure 14).

Cycle-value	Cycle-length 7
7509843	1316
7519743	108
7619733	462
8429652	7450
8439552	418
8649432	1012
8719722	204
9529641	460

Header length	Cycle-length 7
1	622
2	958
3	1136
4	1234
5	956
6	1516
7	2088
8	1050
9	930
10	394
11	320
12	168
13	58

Figure 14: Width = 7, only unique values processed.

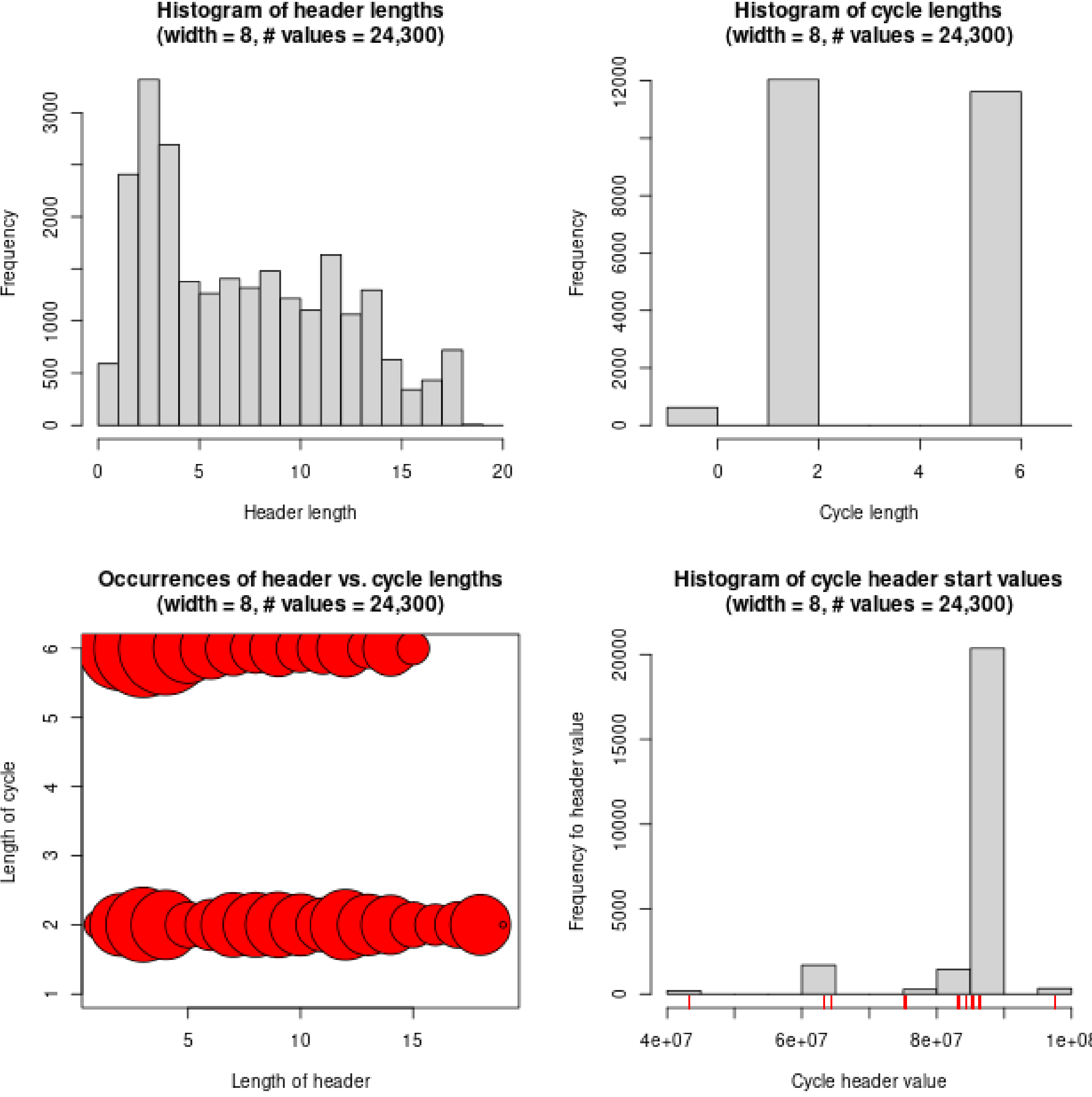
5.8 Width = 8

Only unique possible values allowed by the width value were processed. In all cases the cycle length was 1, or 3, or 7 operations and the header length is between 1 and 19 operations. So, very short headers, and lengthy cycles. There are 12 cycle values. The combined data are presented in (see Figure 15).

	Cycle-length		
Cycle-value	1	3	7
43208766	0	0	198
63317664	300	0	0
64308654	0	1075	0
64326654	0	0	318
75308643	0	0	271
83208762	0	784	0
84308652	0	0	665
85317642	0	0	818
86308632	0	0	6043
86326632	0	0	3315
86526432	0	10192	0
97508421	321	0	0

	Cycle-length		
Header length	1	3	7
1	77	156	357
2	224	770	1411
3	308	1103	1906
4	12	948	1731
5	0	418	960
6	0	519	746
7	0	815	592
8	0	821	496
9	0	843	635
10	0	755	462
11	0	575	530
12	0	980	656
13	0	738	327
14	0	690	607
15	0	417	212
16	0	343	0
17	0	430	0
18	0	722	0
19	0	8	0

Figure 15: Width = 8, only unique values processed.



6 Future Work

As demonstrated (see Listing 6), it appears that Kaprekar's routine works for different numerical bases (at least to limited testing). The same exploration should be done with different numeric bases, and different widths as was done with the base 10 values. There is probably a practical upper limit on both the base and the width due to CPU and numerical limitations. Currently, bases 2 through 26 are supported simply because it was easy to deal with numeric and upper case letter base value representations. A trivial extension would be to add lower case alphabetic representations. Extending the bases beyond single byte/character representations would more of a challenge, and may necessitate a programming language other than R. The internal storage of values may also present a problem not solvable using R. For instance, a base 26 number that is 10 digits wide is a large decimal value ($\approx$ 3,670,344,486,987,776), and operations with numbers of this magnitude require special processing (again possibly beyond the scope of R).

The number of unique values based on numeric width is another possible area of investigation. Data generated by the embedded program (see Listing 7) could be expanded to include greater widths and a curve fitted to the resulting data. If such a curve could be found, then it could be used to predict how many unique values are possible for a given width, then therefore a decision as to the practicality of exploring that width determined.

7 Conclusion

Investigating Kaprekar's Constant was a delight. The routine is almost trivial in the extreme[3]:

1. Take any four-digit number, using at least two different digits (leading zeros are allowed).

2. Arrange the digits in descending and then in ascending order to get two four-digit numbers, adding leading zeros if necessary.

3. Subtract the smaller number from the bigger number.

4. Go back to step 2 and repeat until the number is 6174.

and cries for testing at many different levels. These include:

1. Do all 4 digit numbers terminate at 6174? Yes. This includes all values from 1 to 9999. Most people when asked to write a 4 digit number will write a value that has 4 characters. In reality 1, 2, and 3 digit numbers are really 4 digit numbers, with some number of leading zeros that are generally not written.

2. Do all values terminate at 6174? No. Only numbers that are 4 digits wide terminate at 6174. All values investigated terminate, but at values other than 6174 (see Table 1).

Table 1: Kaprekar terminating decimal values based on numeric width. The precise terminating value is determined by the initial value.

Numeric Width	**Terminating Value(s)**
2	$\{9,27,45,63,81\}$
3	495
4	6174
5	{53955, 59994, 61974, 62964, 63954, 71973, 74943, 75933, 82962, 83952}
6	{420876, 549945, 631764, 642654, 750843, 840852, 851742, 860832, 862632}
7	{7509843, 7519743, 7619733, 8429652, 8439552, 8649432, 8719722, 9529641}
8	{43208766, 63317664, 64308654, 64326654, 75308643, 83208762, 84308652, 85317642, 86308632, 86326632, 86526432, 97508421}

3. Does the routine terminate? The routine terminates as far as the numeric width of decimal numbers was tested (see Table 1). Testing was terminated because testing width 8 values was taking too long.

4. Does the routine terminate with numeric bases other than decimal? Very limited testing shows that it does. The embedded R scripts could be modified to extensively explore other bases and other widths.

5. Do all numeric width values need to be explored? No. The first step in the Kaprekar routine is to separate the numeric value under consideration into it's individual characters and then sort those characters. This separating, sorting, and then combining the characters maps many different values to a single value. Only this single value needs to be given to the Kaprekar routine to arrive at the terminating value.

6. Will all values, regardless of base or width terminate? It is expected so. The essence of Kaprekar's routine is to subtract a smaller number from a larger number, and repeat until your result is a number you've encountered before. These operations can be envisioned as a directed graph, where the previously computed value forms a cycle.

Exploring Kaprekar's routine opened a small world of questions and numeric investigations. It was a true delight to wander through this space.

8 Embedded files

Embedded files may not be extractable using conventional browsers. They are extractable using Adobe Acrobat and Reader software by right-clicking on the paperclip and saving the file to a local directory.

- basesLibrary.R – A support library used when computing numeric bases other than 10. This file is needed when exploring with Kaprekar with different bases.
- kaprekar-bases.R – The Kaprekar routine using different numeric bases.
- kaprekar-classic.R – The "classic" Kaprekar routine.
- kaprekar-unknown.R – The Kaprekar routine without knowing about 6174.
- kaprekar-where.R – The Kaprekar routine that lists when a value is first repeated, and how many operations does it take to repeat the value.
- kaprekar-width.R – The Kaprekar routine where the width of digits can be varied.
- uniqueNumbers.R – Demonstrating the reduction in unique values after the first step in the Kaprekar routine.
- kaprekar-parallel.R – Spreading Kaprekar processing across available cores in your local machine.
- library.R – Functions required by *kaprekar-parallel.R*.

References

[1] Allan Cameron, *How can I convert between numeral systems in R?*, `https://stackoverflow.com/questions/64378066/how-can-i-convert-between-numeral-systems-in-r`, 2020.

[2] Yutaka Nishiyama, *The Mysterious Number 6174*, Gedai Sugakusha, Co., 2013.

[3] Wikipedia Staff, *6174*, `https://en.wikipedia.org/wiki/6174`, 2024.