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
initialization;
list ← *Empty* a way to keep track of previously computed values;
repeat
    list ← num is appended to list;
    num ← num with leading zeros to width digits wide;
    digits ← individual digits from num;
    minuend ← sort digits high to low and convert into value;
    subtrahend ← sort digits low to high and convert into value;
    num ← minuend - subtrahend;
until *num* in *list*;
repeatLocation ← location of num in list `/* The location of the repeated value from the front of the list. */`
repeatLength ← length of *list* - repeatLocation`/* The number of operations from the location of the repeated value until the value is repeated. */`
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.

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

```
initialization;
list ← Empty a way to keep track of previously computed values;
repeat
    list ← num is appended to list;
    num ← num with leading zeros to width digits wide;
    digits ← individual digits from num;
    minuend ← sort digits high to low and convert into value;
    subtrahend ← sort digits low to high and convert into value;
    num ← minuend - subtrahend;
until num in list;
repeatLocation ← location of num in list /* The location of the repeated value
    from the front of the list.                                          */
repeatLength ← length of list - repeatLocation/* The number of operations from
    the location of the repeated value until the value is repeated.   */
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 ← maximum number of digits wide for any number
   Data: base ← a numeric base for all operations
   Result: The set of functionally unique Kaprekar values
 1 initialization;
 2 file ← a temporary file to store values;
 3 num = 1;
 4 for num ≤ (base^width − 1) do
 5 |   temp ← num with leading zeros to width digits wide;
 6 |   digits ← individual digits from temp;
 7 |   uniqueDigits ← (sort(digits) → unique() → length());
 8 |   if uniqueDigits ≥ 2 then
 9 |   |   value ← sort digits low to high and convert to string;
10 |   |   write value to file;
11 |   end
12 |   num ++;
13 end
14 read file → sort strings → 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. 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

```
initialization;
values ← either all values, or only unique values;
breaks ← "break" the values to be spread equally across all cores;
num = 1;
Configure cores as a cluster;
for num ≤ cores do
    Distribute values, base, and width to individual cores;
    Begin individual core execution;
    num ++;
end
for num ≤ cores do
    Collect Kaprekar values from each core;
    num ++;
end
return Collected data from all cores;
```