1 Design

This is an implementation in pseudocode of all 4 components of the radix sort variations included in this project: Radix sort starting from the most significant and least significant digit and then the integer sorting methods Counting sort and Pigeonhole sort. Each permutation of Radix sort takes the input array A of integers and treats each item $A[i]$ as a series of digits d within the range $0 \leq d < k$ where k is the radix or the number of unique digits.

1.1 Integer Sorting

Integer sorting is the underlying sorting method that is applied to each digit of the input array, these methods are non-comparitive algorithms that distribute items based on the value of each digit.

1.1.1 Pigeonhole Sort

Pigeonhole sort operates by instantiating an array B of length k (1) containing empty arrays (3). The input array A is iterated over and each item $A[i]$ is appended to the array in B at the position that corresponds to the current digit of the current item. B is then iterated over (10) and each item in each array overwrites the original array in order (11).

Algorithm 1 Pigeonhole Sort(A)

Input: $A \Leftarrow$ Unsorted Array of length n with maximum value k
Output: Sorted Array

```
1:  B ⇐ [1..k]                          ▷ Buckets array
2:  for i ← 1 to k do
3:      B[i] ⇐ [ ]                      ▷ Each bucket contains empty array
4:  end for
5:  for j ← 1 to k do
6:      append A[j] to B[A[j]]          ▷ Move item to bucket
7:  end for
8:  c ⇐ 1                               ▷ Count items in A
9:  for i ← 1 to k do
10:     for all item in B[i] do
11:         A[c] ⇐ item                 ▷ Move items from bucket to A
12:         c ⇐ c + 1
13:     end for
14: end for
15: return A
```

1.1.2 Counting Sort

Counting sort instantiates 2 arrays; B, the output array equal in length to the input array A (1), and C, a count array to store the frequency of each digit which is of length k and is instantiated to contain only integers of value 0 (2, 4). The input array is iterated over and the integer in C at the position equal to the current digit of the current item is incremented (7), producing a histogram of the frequency of each digit. This count list is then accumulated such that each item is added to the sum of the previous values (10). Finally the input array is iterated over in reverse and each item is moved to the output array in the position that is referred to by the value expressed in the count array at the position of the current digit of the current item(13), to ensure that values to not overwrite one another in the output array each item in the count array is decremented when a position in the output array is taken from it (14).

Algorithm 2 Counting Sort(A)

Input: $A \Leftarrow$ Unsorted Array of length n with maximum value k
Output: Sorted Array

1: $B \Leftarrow [1..n]$ ▷ Temp array
2: $C \Leftarrow [1..k]$ ▷ Count array
3: **for** $i \leftarrow 1$ to k **do**
4: $\quad C \Leftarrow 0$
5: **end for**
6: **for** $j \leftarrow 1$ to n **do**
7: $\quad C[A[j]] \Leftarrow C[A[j]] + 1$ ▷ Count occurence of value
8: **end for**
9: **for** $i \leftarrow 2$ to k **do**
10: $\quad C[i] \Leftarrow C[i] + C[i-1]$ ▷ Accumulate Count array
11: **end for**
12: **for** $j \leftarrow n$ to 1 by -1 **do**
13: $\quad B[C[A[j]]] \Leftarrow A[J]$ ▷ Move item from input to temp array
14: $\quad C[A[j]] \Leftarrow C[A[j]] - 1$ ▷ Deincrement count array
15: **end for**
16: **return** B

1.2 Radix Sort

Radix sort is the method of applying the integer sorting method to each digit of the input, there are two approaches to this: Least significant digit first (LSD) or Most significant digit first (MSD).

1.2.1 Least Significant Digit

Least Significant Digit Radix sort (LSD) takes an input array A and sorting function *f()* which can be either of the previous integer sorting methods. The input array is sorted via the chosen method once for each of digits in the chosen radix of the largest value (3) This sorting process is applied first at the least significant or rightmost digit and moves incrementally to the most significant or leftmost digit. Values can have an unequal number of digits provided any digits farther left than their most significant are taken as 0 and the chosen integer sorting method is stable as this means they maintain their relative position with respect to values that have an equal or lesser number of digits.

Algorithm 3 LSD(A, f())

Input: $A \Leftarrow$ Unsorted Array of length n with maximum absolute Value k
Input: $f() \Leftarrow$Sorting function ▷ Pigeonhole or Counting Sort
Output: Sorted Array
1: $d \Leftarrow$ Number of digits of k ▷ Varies based on radix chosen
2: $A(i) \Leftarrow$ i-th digit of each item in A
3: **for** $i \leftarrow 1$ to d **do**
4: $A \Leftarrow f(A(i))$ ▷ Sorts Array using sort function
5: **end for**
6: **return** A

1.2.2 Most Significant Digit

Most Significant Digit Radix sort (MSD) takes an input array A, sorting function *f()* and current digit i which for the first call of the function should be the number of digits of the largest value in the input. This algorithm operates recursively, and the termination condition occurs when i is equal to 0 meaning that all digits of the input have been sorted (1,2). After ensuring the termination condition has not been met the input array A is then sorted using the chosen function with respect to the current digit (5). The input array is then iterated over and items that have an equal value at the current digit are identified (8) and taken as a slice of the input array. This slice is then sorted recursively using a function call to *MSD()* with a decremented value for i meaning they are sorted over the next least significant digit (11). Once the termination condition has been reached within this slice and all sub-slices created by recursive calls the algorithm resets the start point of the slice and identifies further slices to sort (12).

Algorithm 4 MSD(A, i, f())

Input: $A \Leftarrow$ Unsorted Array of length n with maximum absolute Value k
Input: $i \Leftarrow$ Digit to sort Array on ▷ start with number of digits of k
Input: Sorting function $f()$ ▷ Pigeonhole or Counting Sort
Output: Sorted Array

1: **if** $i == 0$ **then**
2: **return** A
3: **end if**
4: $A(i) \Leftarrow$ i-th digit of each item in A
5: $A \Leftarrow f(A(i))$ ▷ Sorts Array using sort function
6: $s \Leftarrow 1$ ▷ Starting position of recursive call
7: **for** $j \leftarrow 1$ to s **do**
8: **if** $A(i)[j] == A(i)[s]$ **then**
9: Do Nothing
10: **else**
11: $A[s..j-1] \Leftarrow MSD(A[s..j-1](i), i-1)$
12: $s \Leftarrow j$ ▷ Reset start position of array slice
13: **end if**
14: **end for**
15: **return** A

2 Time complexity

2.1 Integer Sorting

Counting and Pigeonhole sort both have a time complexity of $\mathcal{O}(n+k)$ where n is the number of items in the input array and k is the maximum range of input values, this can be seen as one pass over the input data containing n items and one pass over the intermediary buckets array for pigeonhole sort or the count array for counting sort, both of which contain k items.

2.2 Radix Sort

Radix sort repeatedly applies integer sorting to each successive digit of the input array, it differs from counting sort in that it does not sort into k buckets but a constant base b is chosen. It repeats the integer sorting method d times where d is the number of passes necessary to sort each digit of the maximum value in the input with respect to the base b and can be expressed as $d = log_b k$
This gives a time complexity of $\mathcal{O}((n + b) \cdot d)$ or if we select some constant value for b; $\mathcal{O}(n \cdot d)$
However, in the application of this project, radix sort will only be applied to integers represented in 64-bits, meaning there is a defined upper bound to the max value k. This is important because it means that that k is polynomial in b or that k is always able to be represented by b raised to some exponent c where c is some constant value. This can be expressed in the form $k = b^{\mathcal{O}(1)}$ or $\mathcal{O}(1) = log_b k$
$\mathcal{O}(1)$ can be substituted in for d in the time complexity analysis meaning Radix sort will run in $\mathcal{O}(n)$ or linear time in the context of this project.

3 Radix Selection

3.1 Introduction

Radix sort utilises repeated rounds of integer sorting across each digit of the input array, therefore, it is necessary to make a clear definition of a digit so that an effective method of separating an integer into its component digits can be implemented. When expressing a number a base must first be decided, this is the number of unique digits that can be displayed at any position and when a value exceeds the maximum value within this base another digit must be added; i.e. in standard decimal or base 10 numbers 0 through 9 can be expressed in a single digit but increasing beyond this range requires adding a second digit to create the number 10. Though we typically think of numbers in the context of arabic numerals in the decimal system, this is a social convention which this algorithm is not beholden to and as Radix sort fundamentally relies on the processing of the component digits of an integer it is of great importance that an appropriate base is selected as well as an efficient method for processing each digit.

3.2 Explanation

Bases that are of the order 2^n where n is some integer are the most suitable for a typical radix sort algorithm, this is because bases that fit this specification allow for easy conversion from binary which is how the computer on which the algorithm runs stores numbers. Any base of 2n will utilise all possible combinations of n digits of binary to express a single digit, meaning, to sort integers across a given digit it is possible to extract a known quantity of binary values at a known position within the binary representation of the integer. Provided it is possible and efficient to extract the binary value of an integer within a programming language, this approach is significantly superior to working in base 10 as it requires no mathematical calculations to be performed on each input item whereas working with base 10 would require some kind of use of the modulo or remainder functions or a type conversion to a string; both of these options are significantly more resource intensive.

3.3 Implementation

Python stores integers in "two's complement" binary and allocates 64 bits, this means that by utilising the in-built python bitwise operations it is possible to quickly manipulate integers and acquire the digit in any position so long as the base is an exponent of 2.
As an example, selecting a base of 2^4 (16) and Radix sort from the least significant digit, utilising the bitwise "and" operation on a chosen integer with 15 (the largest integer that can be expressed in 4 bits (1 1 1 1 in binary) and find the digit in the rightmost position and use that for integer sorting. Then to move on to the next most significant digit we can perform a "right shift" of 4 bits, discarding the bits representing the already sorted digit and moving the bits representing the next digit into the rightmost position and allowing the same "and" operation as used previously to identify the digit.

4 Negative Integers

4.1 Two's Complement

When handling integers in two's complement, positives are represented in binary, but with a leading 0 in the leftmost position to indicate the positive sign of the integer. Negative integers are represented with a leading 1 which represents that the binary value of this digit is to be subtracted i.e. negative and all other binary digits are positive such that the sum of all the digits equals the integer being represented. table reference has been included to illustrate this, as you can see the value +3 has a leading 0 at index 2, representing 2^2 is not subtracted, this is followed by 1 in all remaining digits representing the addition of 2^1 and 2^0 or $2 + 1 = 3$. −3 is represented by a leading 1 to represent -2^2 followed by 0 1 to represent no addition of 2^1 and the addition of 2^0 or $-4 + 1 = -3$.
The effect of performing Radix sorting without any accommodation for negativity is that due to the leading 1, negative values will be incorrectly sorted ahead of positive values as though they were larger. They will also be sorted into the correct order relative to other negative values (increasing from most negative to least negative), as among values that can be represented in the same number of bits the more negative numbers have fewer additional bits in positions to the right, for example −4 is 100 and −3 is 101.
The method of correcting for this problem is to invert the sign bit using a bitwise "exclusive or" operation with a 1 in the position of the sign bit or more accurately performing this operation against 1+ the largest integer possible with the current machine word length. This operation gives 1 if the sign bit of the integer is 0 or 0 if the sign bit is 1. Following this it must be ensured the integer sorting method sorts across the sign bit either at the start for MSD or at the end for LSD, this preserves the relative order amongst positive and negative integers but ensures this pass of the data sorts the items such that negatives come before positives.

4.1.1 Two's complement three bit integers

Bits	Unsigned Value	Signed Value (Two's Complement)
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
101	6	-2
111	7	-1

4.2 Implementation

The purpose of this project was to incorporate this sorting algorithm into the standard library for Pypy, meaning it had to be written in RPython, the language used for Pypy. RPython is a restricted form of python, specifically it is a form of Python 2, which is relevant to the handling of integers as RPython inherits Python2's method of allocation of a quantity of bits for each integer equal to the word size of the machine it is being run on (typically this is 32 or 64 bits), even if the integer can be expressed in fewer bits. The leftmost bit represents whether the integer is positive or negative and the remaining bits represent the magnitude of the integer, all digits between the sign bit and the most significant bit needed to represent the integer are copies of the sign bit. To put this in the context of two's complement , this representation can be understood as the first bit representing -2^m where m is the machine word length and all following digits are additions of 2^n where n is $[m-1..0]$.

The true sign bit, meaning the bit that always and exclusively represents the sign of the integer, is the one in the leftmost position, however it is not necessarily the most optimal solution to this problem of negatives being sorted into the incorrect position to flip this bit. Firstly, when sorting integers that take up a small number of bits it would require some additional logical test to identify if the algorithm is performing the first (for MSD) or last (for LSD) pass of the data to then instruct the algorithm to jump to the leftmost bit. Additionally, it is necessary to identify the machine word length of the machine on which this code is running as flipping the 64th bit on a 32 bit machine is not possible, this eliminates the possibility of a hardcoded value and essentially forces the use of the "sys.maxint" function from the Python 2 standard library. I decided that using this function was not in the best interest of this project for posterity as it was removed from Python 3 and replaced with "sys.maxsize" which means if RPython ever moved on to Python 3 this function would stop working; whilst this is a minor issue that would likely never cause problems there is an alternative solution without any issues.

The approach used in this project was to identify the number of digits needed to express the largest absolute value of the input in the chosen base and then flip the leftmost bit of the digit preceding this digit. So for example the integer 246 when expressed in base 4 (equal to 2^2 or 8 bits) requires 4 digits [3312], so in RPython the preceding bits are all 0 to indicate the sign being positive, this method then identifies the bits that would represent the next digit (the bits 9th and 10th from the right) and flips the leftmost digit giving it the binary value of [10] and a digit value of 2 in base 4. This dynamic approach ensures that the relative correct order within the positive and negative integers is preserved and is suitable for use with any integers being sorted in any base.

5 Implementation

5.1 Extra Methods and Class

Explanation of all functions and classes that exist outside of the radixsort algorithms. Each function is documented in isolation in the order they appear in the source files, for a full view of each of the files refer to reference

5.1.1 Function absolute

The purpose of this helper method was to allow for the calculation of absolute values without overflow issues specifically caused by using the minimum possible value. This value's positive counterpart lies outside of the range of the 64 bits assigned for each integer in RPython, this causes unpredictable errors and even though there is only one specific case that necessitates the use of this method over the regular abs() inbuilt function, the runtime impacts are low so it's a worthwhile helper method to include.

```
import sys

def absolute(num):
if num == (-sys.maxint) - 1:
return sys.maxint
return -num if num < 0 else num
```

5.1.2 Function int_bytes

Returns the minimum quantity of bytes in a given radix necessary to express an integer, including a sign bit. This function is used to calculate the number of iterations that the sorting algorithm must make in order to sort all values up to and including the given input i.
The function calculates $\log_{radix}(i)$, i.e. to what power i must be raised to get $radix$, uses ceil to get the next highest integer and adds 1 to the value to allow for the sorting of negatives. It is necessary to return an integer in this case because this value is used to determine the number of bits used to identify each digit and the number of iterations over the list that must be made, both of these are discreet values. $ceil()$ has been used instead of $int()$ or $floor()$ as a value lower than needed would result in incorrect sorting of values the larger values.

```
from math import ceil, log

def int_bytes(i, radix):
return ceil(log(absolute(i),radix)) + 1
```

5.1.3 Function make_radixsort_class

The container function that instantiates all other funtions, this is modelled on the existing function in listsort.py reference to minimize compatability issues when including this project in the pypy library.
Almost all functions have been removed or altered in a major way, particularly any functions that pertain to comparative sorting, it is the general structure that has been emulated.

```
def make_radixsort_class(
    setitem=None,
):
```

5.1.4 Function setitem

This function has been imported from the existing listsort.py file within rlib Reference? I'm not actually 100% on why this is necessary. It exists outside of the Radixsort class and is then is imported by the class and used to alter the input list outside of the scope of the class.
This is the only existing constructor of the original listsort.py that was not removed.

```
if setitem is None:
  def setitem(list, item, value):
      list[item] = value
```

5.1.5 Class Radixsort

This is the class that contains all other functions of the radixsort algorithm, it creates an object that is used to ease the manipulation of the input list and store attributes that are referenced and edited from multiple different functions.
in particular the list attribute contains the input list, which is used by every other funtion of the object.
The function *setitem* is imported here as mentioned above so that the list can be edited outside of the scope of the class.

```
    class Radixsort(object):
        def __init__(self, list, listlength=None):
            self.list = list
            self.base = 0
            self.listlength = len(self.list)
            self.radix = 0

        def setitem(self, item, value):
            setitem(self.list, item, value)
```

5.1.6 Function list_abs_max

This function is an implementation of a combination of min() and max() functions (neither of which are implemented in RPython reference) as well as an optional module to identify whether the list is sorted in asscending or descending order. The function also compares the maximum and minimum values and returns the value that is farthest away from 0.
The primary purpose of this function is to return the value that is used to identify the number of passes over the input necessary for sorting, this is why it is necessary to identify the farthest value form 0 as this is the value that will contain the most digits.
The secondary purpose of the funtion is to update the class attributes *ordered* and *reverseOrdered*, which are used to identify whether the sort function has been called on an already sorted list in which case there is no purpose in running the algorithm. The reason that these actions are performed at the same time is because iterating over the list is necessary for both purposes, and both must be performed before sorting can take place (or not take place if the list is sorted), therefore it is more efficient to perform both tasks simultaneously.

```
        self.ordered = True
        self.reverseOrdered = True

        def list_abs_max(self, checkorder=False):

            assert len(self.list) != 0
            m = self.list[0]
            n = self.list[0]
            prev = self.list[0]
            for i in range(1, len(self.list)):
                if self.list[i] > m:
                    m = self.list[i]
                if self.list[i] < n:
                    n = self.list[i]
                if checkorder:
                    self.ordered &= self.list[i] >= prev
                    self.reverseordered &= self.list[i] <= prev
                    prev = self.list[i]
            return m if absolute(m) > absolute(n) else n
```

5.1.7 Function insertion_sort

This is an implementation of insertion sort, it is used in MSD sorts when the length of a sublist falls below the critical value. reference

```
    def insertion_sort(self, start, end):
        for step in xrange(start, end):
            key = self.list[step]
            j = step - 1
            while j >= 0 and key < self.list[j]:
                self.setitem(j + 1, self.list[j])
                j = j - 1
            self.setitem(j + 1, key)
```

5.1.8 function reverse_slice

This reverses the items of the input inbetween the provided start and stop indexes. By default is reverses te entire list, but includes provisions to reverse a smaller slice of the list (this was from an earlier version of the algorithm, this feature is currently unused). I should probably just remove that at this point

```
    def reverseSlice(self, start=0, stop=0):
        if stop == 0:
            stop = self.listlength - 1
        while start < stop:
            i = self.list[start]
            j = self.list[stop]
            self.setitem(start, j)
            self.setitem(stop, i)
            start += 1
            stop -= 1
```

5.2 Radix sort preamble

In this section I will explain the four different implementations of Radix sort, but as they all operate with the same general principal I will first explore the features and tools they have in common with particular attention to how these tools are applied within RPython.

5.2.1 Bitwise operations

Bitwise operations are the heart of this project, I have already covered in general terms how they are implemented in this project to identify the value of each digit in 3.3 and handle negative integers in 4.2 but in this section I will cover specifically which bitshift operations are used and their exact effects in RPython

5.2.1.1 Leftshift

A left shift refers to moving each bit of an integer a number of digits to the left, in RPython this is performed with the operator $n << s$ where n is the integer to be shifted and s is the number of digits the integer should be shifted to the left.
RPython utilises Arithmetic Shifting reference, this causes when a shift is performed and the rightmost bit is no longer in the rightmost position, all digits in the range $[0..s]$ are overwritten with the value 0 and an equal quantity of digits in the leftmost positions are discarded.

In this project, left shifting is used to create integer masks reference used to manipulate the sign of integers and discard irrelevant data when sorting inputs across a specific digit (explained further in reference).
For example

5.2.1.2 Rightshift A Right shift is thenverse of left shift, it refers to moving each bit of integer digits to the right, using the operator $n << s$ where n is the integer to be shifted and s is the number of digits the integer should be shifted to the right.
Like leftshift, rightshift utilises Arithmetic shifting, meaning the rightmost s digits are discarded. Rightshift differs from leftshift however as the digits in the the leftmost positions are overwritten with a value equal to the sign bit, i.e. 1 or 0 if the number is negative or positive respectively.

Rightshifting is used in this project to manipulate integers from the input list by moving discarding a number of digits. The bases selected for this project are of the order 2^n where n is some integer, meaning a single digit of the integer occupies n bits and therefore; rightshifting $n * d$ positions places the digit d from the right in the rightmost position, occupying the rightmost n bits.