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: $B \Leftarrow [1..k]$ $\triangleright$ Buckets array
  2: **for** $i \leftarrow 1$ to k **do**
  3:     $B[i] \Leftarrow [\ ]$ $\triangleright$ Each bucket contains empty array
  4: **end for**
  5: **for** $j \leftarrow 1$ to k **do**
  6:     *append $A[j]$ to $B[A[j]]$* $\triangleright$ Move item to bucket
  7: **end for**
  8: $c \Leftarrow 1$ $\triangleright$ Count items in A
  9: **for** $i \leftarrow 1$ to k **do**
10:     **for all** *item* in $B[i]$ **do**
11:        $A[c] \Leftarrow item$ $\triangleright$ Move items from bucket to A
12:        $c \Leftarrow c + 1$
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: $B \Leftarrow [1..n]$ $\triangleright$ Temp array
2: $C \Leftarrow [1..k]$ $\triangleright$ Count array
3: **for** $i \leftarrow 1$ to k **do**
4:      $C \Leftarrow 0$
5: **end for**
6: **for** $j \leftarrow 1$ to n **do**
7:      $C[A[j]] \Leftarrow C[A[j]] + 1$ $\triangleright$ Count occurence of value
8: **end for**
9: **for** $i \leftarrow 2$ to k **do**
10:     $C[i] \Leftarrow C[i] + C[i-1]$ $\triangleright$ Accumulate Count array
11: **end for**
12: **for** $j \leftarrow n$ to 1 by -1 **do**
13:     $B[C[A[j]]] \Leftarrow A[J]$ $\triangleright$ Move item from input to temp array
14:     $C[A[j]] \Leftarrow C[A[j]] - 1$ $\triangleright$ 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 $\quad\quad$ ▷ Pigeonhole or Counting Sort
Output: Sorted Array
$\quad$ 1: $d \Leftarrow$ Number of digits of k $\quad\quad$ ▷ Varies based on radix chosen
$\quad$ 2: $A(i) \Leftarrow$ i-th digit of each item in A
$\quad$ 3: **for** $i \leftarrow 1$ to d **do**
$\quad$ 4: $\quad$ $A \Leftarrow f(A(i))$ $\quad\quad$ ▷ Sorts Array using sort function
$\quad$ 5: **end for**
$\quad$ 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 $\quad \triangleright$ start with number of digits of k
Input: Sorting function $f()$ $\quad\quad \triangleright$ Pigeonhole or Counting Sort
Output: Sorted Array
 1: **if** $i == 0$ **then**
 2: $\quad$ **return** A
 3: **end if**
 4: $A(i) \Leftarrow$ i-th digit of each item in A
 5: $A \Leftarrow f(A(i))$ $\quad\quad\quad\quad\quad \triangleright$ Sorts Array using sort function
 6: $s \Leftarrow 1$ $\quad\quad\quad\quad\quad\quad\quad \triangleright$ Starting position of recursive call
 7: **for** $j \leftarrow 1$ to s **do**
 8: $\quad$ **if** $A(i)[j] == A(i)[s]$ **then**
 9: $\quad\quad$ Do Nothing
10: $\quad$ **else**
11: $\quad\quad$ $A[s..j-1] \Leftarrow MSD(A[s..j-1](i), i-1)$
12: $\quad\quad$ $s \Leftarrow j$ $\quad\quad\quad\quad\quad \triangleright$ Reset start position of array slice
13: $\quad$ **end if**
14: **end for**
15: **return** A

---

2 Time complexity

2.1 Integer Sorting

Counting and Pigeonhole sort both require one iteration over the input list which contains n items and one pass over the intermediary buckets array for pigeonhole sort or the count array for counting sort, both of these arrays contain k items where k is the maximum range of input values. The combination of these two passes gives both integer sorting methods a time complexity of

$$\mathcal{O}(n+k)$$

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 $d = log_b k$, giving a time complexity of:

$$\mathcal{O}((n + b) \cdot d)$$

or if b is a constant:

$$\mathcal{O}(n \cdot d)$$

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 list, 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 of any kind, it is necessary to establish the numerical base that will be used, this is the number of unique values that can be displayed in each digit 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 numbers in typical day to day use by humans are typically thought of as 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 to select an appropriate base as well as identify an efficient method for processing each digit.
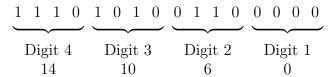
3.2 Explanation

Python stores integers in the format "two's complement", which is a type of binary and therefore uses a base of 2. This means that 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 each digit to be directly accessed from an integer's binary digits.

Any base of 2^n will utilise all possible combinations of n digits of binary to express a single digit and so the rightmost n digits represent the first digit in 2^n, the next n digits represent the next digit etc.

3.3 Implementation

The method of identifying each digit of an integer from the binary representation first involves manipulating the integer such that the desired digit is in the rightmost position, in base n this means discarding the rightmost $n \cdot d$ binary digits where d is the index (from the right) of the required digit in base 2^n. Following this, isolating the rightmost n binary digits produces the correct digit.

In order to complete this task, bitwise operations have been utilised, and are explained fully in section 5.2.

$$\underbrace{1 \ 1 \ 1 \ 0}_{\substack{\text{Digit 4} \\ 14}} \ \underbrace{1 \ 0 \ 1 \ 0}_{\substack{\text{Digit 3} \\ 10}} \ \underbrace{0 \ 1 \ 1 \ 0}_{\substack{\text{Digit 2} \\ 6}} \ \underbrace{0 \ 0 \ 0 \ 0}_{\substack{\text{Digit 1} \\ 0}}$$

The 4 digits in base 2^4 of a 16 bit integer: 60000

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 bits representing the sign of the largest absolute value of the input list (see 5.2.5 for further explanation). This operation causes the items to retain their relative position amongst elements of the same sign but causes negative values to be sorted to their correct position as less than positive values.

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

Table 1: 3 bit integers and their respective values when interpretted in unsigned binary and two's complement

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

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.

```python
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 quantity of digits needed to express an integer using a specified numerical base. Each digit requires n binary bits to express an integer in the base 2^n. The returned value includes the bit needed to express the sign integer, which is sometimes taken as part of the most significant digit and sometimes is not, but this is handled in either case by the sorting algorithm.

    This function uses a loop to rightshift the absolute value of the input an increasing number of times until it equals or falls below 0. The function begins with rightshifting once to ensure that 0 is never returned so as not to produce errors elsewhere in the algorithm. Finally the number of bits has the modulo of the radix subtracted and is divided by the radix, finding the number of times the radix is divisible by l without producing a fraction. 1 is added to the returned value to include the sign bit if l was divisible by the radix and to include the sign bit and all bits expressed in the remainder otherwise.

```python
def int_bytes(i, radix):
    l = 1
    while (absolute(i) >> l) > 0:
      l += 1
    diff = (((l) - (l % radix)) / radix) + 1
    return diff
```

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 [**listsort**] to minimize compatability issues when including this project in the pypy library.

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

5.1.4 Functions setitem and setslice

These functions have been imported from the existing listsort.py file within rlib. They enable the input list to be edited by functions within the Radixsort class. When the *.sort*() function is called by PyPy, it initialises the Radixsort class with the list as a constructor, this means that the list attribute is not mutable from within the class unless these functions are imported.

These are the only existing functions of the original listsort.py that were not removed.

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

if setslice is None:
    def setslice(list, slice, index):
        list[index : index + len(slice)] = slice
```

5.1.5 Class Radixsort

This is the class that contains all other functions of the radixsort algorithm. It is initialised with the input list which is stored as an attribute for access by other functions as well as the base and radix.

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)
    def setslice(self, slice, index=0):
        setslice(self.list, slice, index)
```

5.1.6 Function list_abs_max

This function returns the largest absolute value of the input list as well as identifing whether the list is sorted in asscending or descending order. These functions are combined as a time saving measure as they both require a pass over the data.

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. Note that neither min() or max() are implemented in RPython [**rpythonobj**], hence why they have not been used.

The secondary purpose of the function is to identify on the first pass over the list whether it is sorted, in which case the algorithm terminates, or if it is reverse sorted in which case the list is reversed and then the algorithm terminates.

```python
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 xrange(1, len(self.list)):
        if self.list[i] > m:
            m = self.list[i]
        if self.list[i] < n:
            n = self.list[i]
        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. This value was determined by testing as explored reference testing

```python
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).

```python
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
```

5.1.9 function is_sorted

This function exists to check whether a subsection of the list is sorted. It is called on subsections of the list during MSD sorts. The assert statements exist to ensure that RPython compiles the code as it does not allow for slicing lists with negative slice indexes [**rpythonobj**]

```python
def is_sorted(self, start=0, end=-1):
    end = end if end > -1 else self.listlength
    assert start >= 0
    assert end >= 0
    sortd, rsortd = True, True
    for i, el in enumerate(self.list[start + 1 : end]):
        if el < self.list[start + i]:
            sortd = False
            if rsortd == False:
                return sortd, rsortd
        if el > self.list[start + i]:
            rsortd = False
            if sortd == False:
                return sortd, rsortd
    return sortd, rsortd
```

5.2 Bitwise operations

Bitwise operations operate on binary numerals at the level of individual bits, they allow for fast and consistant manipulation that bypasses arithmetic functions. Through the use of bitwise operations, radix sort can be performed using different numeral bases, meaning a diffent maximum value at each digit and a different total number of digits. By reducing the number of digits the amount of iterations over the data is reduced thereby improving run time, however this comes at the cost of an increase in the amount of time each round of integer sorting takes and an increase in the memory consumption due to the larger count list or quantity of bins.

Bitwise operations are particular useful in this regard as it makes processing integers in bases of the order 2^n where n is some integer very easy (see section 3.2) whereas changing base from the standard base 10 requires time consuming arithmetic methods such as euclidian division.

In order to properly implement bitwise operations it was necessary to properly explore their implementation in RPython as well as the implementation of the Integer type. In RPython each integer is comprised of exactly 64 bits, where 1 bit is assigned to represent the integer's sign, leaving a maximum of 63 bits to express the magnitude of the integer, this gives integers in RPython a range of $[-2^{63} \mathinner{..} 2^{63}-1]$. The PyPy interpreter considers any integer outside of this range to instead be of the 'Long' type, and as such a list containing even a single value outside of this range does not have the Integer specific sorting method applied to it, thus values outside of this range are not within the scope of this project.

I have already covered in general terms how these are used in this project to identify the value of each digit in section 3 and handle negative integers in 4.2 but in this section I will cover the implementation of the bitshift operations used in RPython.

5.2.1 Left shift

A 'Left shift' refers to moving each bit of an integer a number of digits to the left, in RPython this is performed using: $n << s$ 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, this means 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 integers that occupy a specific number of bits and is used for the accomodations necessary to sort negative integers (further explained in 5.2.5).

Value	7	6	5	4	3	2	1	0
13	0	0	0	0	1	1	0	1
13 << 2	0	0	1	1	0	1	0	0

Table 2: Demonstration of shifting 13 left 2 bits
only rightmost 8 bits are shown

5.2.2 Right shift

A 'Right shift' is the inverse of left shift, referring to moving each bit of an integer 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 Left shifting, Right shifting utilises Arithmetic shifting, meaning the rightmost s digits are discarded. Right shift differs from leftshift however as the digits in the the leftmost positions that become empty are overwritten with a value equal to the most significant (leftmost) bit, which would typically represent the sign of the integer. This means that a Right shifted integer retains it's original sign.

Right shifting is used in this project to manipulate integers from the input list in order to move a specific digit into the rightmost position so that the value of this digit can be extracted using Bitwise And (see 5.2.3). 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; Right shifting $n \cdot d$ positions places the digit d from the right in the rightmost position, occupying the rightmost n bits.

Value	7	6	5	4	3	2	1	0
13	0	0	0	0	1	1	0	1
13 >> 2	0	0	0	0	0	0	1	1

Table 3: Shift 13 right 2 bits. Only
rightmost 8 bits are shown

Value	63	62	61	60	59	58	57	56
-2^{61}	1	1	1	0	0	0	0	0
-2^{61} >> 2	1	1	1	1	1	0	0	0

Table 4: Shift -2^{61} right 2 places. Only leftmost
8 bits are shown.

5.2.3 And

The 'And' operator takes two integers and compares each bit across all binary digits, returning 1 if both bits are equal to 1 or returning 0 if either or both are equal to 0.

This is utilised in this project to isolate a specified number of bianry digits of an integer, specifically n digits in binary represent one digit of the integer in base 2^n. This is done by 'Right shifting' $n \cdot d$ positions (as stated in 5.2.2) and then by performing an 'And' operation with 2^n-1, which in binary is represented as a 1 in the rightmost n digits, and 0 in all other digits, which will retrieve the rightmost n bits and 0 for all other bits.

In pypy, the And operation is performed using the x & y operator resulting in an 'And' operation between x and y.

Value	7	6	5	4	3	2	1	0
59	0	0	1	1	1	0	1	1
15	0	0	0	0	1	1	1	1
59 & 15	0	0	0	0	1	0	1	1

Table 5: Demonstration of 'And' on 15 and 59

This has isolated the rightmost 4 digits which is one digit in 2^4

5.2.4 Not

The bitwise not operator produces the inverse of all binary digits in an integer, giving a 1 if an integer contains a 0 and vice versa. It is unique in the sense that it is the only operation that is performed on a solitary input. This is used in the process of inverting the sign of integers detailed in 5.2.5

Value	7	6	5	4	3	2	1	0
102	0	1	1	0	0	1	1	0
~102	1	0	0	1	1	0	0	1

Table 6: Demonstration of 'Not' performed on 102

All bits have been inverted

5.2.5 Exclusive Or

'Exclusive Or' compares the bits of two integers and returns a value of 1 for each digit where the two bits are different and 0 if they are the same. It is performed in pypy using $x\,\hat{}\,y$ which results in an 'Exclusive Or' performed across all 64 bits of the two integers x and y.

The use case for this in this project occurs when inverting the sign bit to ensure negative and positive values are sorted correctly (see 4.1 for details).This is done by identifying the quantity of binary digits needed to express the largest absolute value of the input using the function 'int_bytes' (shown in 5.1.2) with a radix of 2, this quantity is given as the integer d. Left shifting an integer of value 1 d times and then subtracting 1 produces the maximum integer expressable in the same number of bits as the largest absolute input value, this is given as integer v. In binary v contains 1 in all digits up to the most significant digit of the largest absolute input value excluding any sign bits and a 0 in all other digits.

This value v is then inverted using bitwise 'Not' (see 5.2.4) so that it instead contains 0 in the bits representing the max input and 1 in all other bits. As explained in 4.2, in two's complement binary all digits to the left of the most significant are equal and represent the sign of an integer, therefore v contains 1 in all the sign bits of the largest max input. Performing the Bitwise 'Exclusive Or' operation with any value in the list against the value v inverts all values in these sign bits. As shown in 4.2, negative integers are originally stored in the correct order but placed higher than positive values, performing these operations ensures that the relative order is not altered but the negatives are positioned correctly.

Depending on the base selected and the number of digits of the largest absolute value, multiple sign bits may be contained in the leftmost digit of some integers being sorted, for example 300 contains 3 digits in base 2^4, but only 9 digits in binary, therefore the leftmost digit includes three sign bits. Therefore all sign bits must be flipped to ensure that sorting an integer across digits containing sign bits does not alter their position in the sorted list.

Value	7	6	5	4	3	2	1	0
-13	1	1	1	1	0	0	1	1
~ 15	1	1	1	1	0	0	0	0
-13$\,\hat{}\,\sim 15$	0	0	0	0	0	0	1	1

Table 7: Performing 'Exclusive Or' on -13 with ~ 15
All bits in range [63..8] now equal 0

5.2.6   Optimizations Incomplete

Some of the code outlined in section Reference code explanation differ in relatively substantial ways from the pseudocode outlined in section 1, many of these differences arise from increasing the performance of the algorithm. In this section I will explain the function and reasoning of these differences.

5.2.6.1   LSD counting sort - creating all count lists in one pass

LSD counting sort is a combination of LSD sort (1.2.1) which calls counting sort (1.1.2) on every digit. It iterates over the input list for each digit: once whilst counting the occurence of the values at the digit and a second time when moving the input list items into the output list based on these count lists.

An optimization I have used is to make a single pass over the input list whilst creating the count lists for every digit simultaneously instead of once for each digit. The reason this is possible is that the occurence of the values for each digit do not change based on the order of the input list, items are moved from the input list to their new position based on the value stored in the count list representing the value of the current digit of the item, the order of the input list has no bearing on the order or values of the count list.

The values stored in the count list represent the position of the items relative to one another after being sorted across a given digit; due to MSD sort creating sublists in a specific order and then sorting the sublists' items amongst one another, a count list created for a diffent digit was therefore created for a different list and so does not represent the position of the items relative only to other items in the sublist.

The way that this has been implemented in this project is seen below. It involves creating a 2 dimensional array with an outer list of d sublists each of length r where d is the number of digits and r is the radix. Following this the input list is iterated over and for each item the count list of each digit is incremented i.e. the sublist at position $[0..d]$ represents the count of the digit at the same position. After this, list of count lists is iterated over and each list is incremented, then when moving the input items based on the count lists the relevant count list can be accesed by accessing the outer list at the index equal to the index of the current digit.

```
#Note: this code has been simplified
counts = [[0 for _ in range(radix)] for _ in range(min_bytes + 1)]

for num in self.list:
    disc = 0
    for i in xrange(min_bytes):
        val = get_digit(num, i)
        counts[i][val] += 1

for i in range(min_bytes):
    for j in xrange(1, self.radix):
        counts[i][j] += counts[i][j - 1]
```

5.2.7 Order checking This needs testing further, it might be incorrect

Radix sort has no advantage when called on already sorted lists compared to unsorted due to the non-comparative nature of the algorithm. Simple comparative algorithms such as bubble sort will actually out perform radix sort when called on sorted lists and as Timsort is highly efficient at sorting them there is substantial motivation for improvements to be made with regards to handling sorted data.

Identifying sorted lists is not computationally expensive, it involves iterating over and comparing the list items in order and ensuring that each value is larger or equal to the previous value, essentially only involving $n-1$ comparisons where n is the input list length giving a time complexity of $\mathcal{O}(n)$. As this algorithm already involves iterating over the data, the run time cost of adding a comparison function to check for sortedness is extremely low, thus a check to identify whether a list is sorted as well as whether a list is reverse sorted is included in the first iteration over the data, which is the point at which the absolute maximum value of the list is identified (a necessary step for this algorithm).

```python
def list_abs_max(self, checkorder=False):

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

Checking whether the list is ordered or reverse ordered whilst
iterating over the list to identify the absolute maximum value

An extension of this idea is to perform this order checking every time the input list is rearranged (once for each digit). Unlike the initial order checking which has a minor footprint on the run time of the algorithm, checking order on each digit has a moderately significant impact, therefore it should only be implemented on the conditions that it reasonably likely that sorted lists will occur and that when the test does identify a sorted list the time saved from early termination of the algorithm is greater than the time added by performing the test.

Testing involving the comparison of the performance of algorithms including and excluding order checking showed that MSD radix sorts can benefit from order checking performed on sublists which trigger positive ordered tests frequently and early termination of the sorting of these sublists improved performance Reference testing that proves this. In contrast, LSD sorts rarely trigger positive results with the frequency of these events dropping to 0 when negative integers are included. The lack

of occurences of sorted lists in this circumstance is likely because the least significant digits have weaker influence on the size of integers meaning aside from random events and specific circumstances the input list is simply less likely to become sorted earlier than expected; the inclusion of negatives procludes the possibility of a list being sorted because of the leading sign bits causing negatives (and particularly negatives closer to 0) to be sorted as though they are the largest integers i.e. in the wrong order.

```python
def is_sorted(self, start=0, end=-1):
    end = end if end > -1 else self.listlength
    assert start >= 0; assert end >= 0
    sortd, rsortd = True, True
    for i, el in enumerate(self.list[start + 1 : end]):
        if el < self.list[start + i]:
            sortd = False
            if rsortd == False: return sortd, rsortd
        if el > self.list[start + i]:
            rsortd = False
            if sortd == False: return sortd, rsortd
    return sortd, rsortd
```

Checking whether a slice of the list is ordered or reverse ordered after integer sorting has been applied to each digit

5.2.8   skipping iterations

The time complexity of radix sort is $\mathcal{O}(N \cdot k)$ where k is the number of digits that must be sorted, therefore it is possible (and effective) to have a meaningful effect on the runtime of the algothim by reducing k, this is why the use of different bases has been focussed on as this also reduces k (the number of iterations over the input list that needs to be made).

Skipping iterations over the data can be made by identifying digits on which no sorting is necessary, the specific method of identification differs based on the implementation of radix sort.

For counting sort, if every single list item contain an identical value for a digit there will be no change in the order of the list when sorting across that digit. This circumstance is identified by a single count list item being equal to the list length and can be identified by iterating over the count list, a positive result means the process of moving the list items can be skipped. As the count list is already iterated over to accumulate the values, this test is computationally inexpensive.

```
skip = []
for i in xrange(min_bytes):
  for j in xrange(self.radix):
    if counts[i][j] == self.listlength:
      skip.append(i)
      break
    if j==1: continue
    counts[i][j] += counts[i][j - 1]
```

Identifying if all items have an identical value for each digit in LSD counting sort.
The same process is used for MSD counting sort but the sublist length is used instead of total list length

In pigeonhole sort a similar opportunity arises in the form of identifying whether all items have been placed into any of the buckets which would mean overwriting the input list with the bucket wouldn't change the order. Testing for this condition involves iterating over the buckets list and counting the number of non-empty buckets, if the count increases past 1 bucket it signals the condition has not been met and the test iteration can be terminated. This test takes up no space and at worst $\mathcal{O}(k)$ time to iterate over the bucket list but negative resulting tests take far less time typically.

```
count = 0
for b in bucket:
  if b!=[]:
    count+=1
    if count>1:
      break
if count==1:continue
```

Identifying if more than 1 bucket has been used

5.2.9 Insertion sort

Radix sort has a significant overhead cost due to the establishment of the count lists and the list of bins that must be instantiated or cleared on each digit, this cost has little to no relation to the length of the list and so it is oveten highly inefficient when compared to other algorithms to perform radix sort.

This issue beomes exponentially worse when radix sort is implemented in MSD fashion due to the nature of establishing sub-lists and thus performing sorting on an ever increasing number of lists containing progressively fewer and fewer items (sorting 150000 items in base 2^{16} can create over 65000 lists with an average length of 2 for which the count list must be established separately).

Using insertion sort in situations where the input list is reasonably small can be much more efficient than performing continuous rounds of radix sort, this effect is compounded on small lists of large items. Identifying the exact point at which it becomes more efficient to use insertion sort was done by testing the time taken to sort lists containing an increasing number of items by the MSD radix sort methods compared to insertion sort. Starting at 2 items where insertion sort universally out-performs each sorting method, an increase in the initial list length by 1 item is tested repeatedly and eventually all radix sort methods perform better than insertion sort above a certain list length with is different for pigeonhole sort and counting sort and different for each numerical base used. The value of list length at which each sorting method out-performs insertion sort is taken as the critical value.

```python
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)
```

Insertion sort algorithm used in this project

5.3 Algorithms

5.3.1 LSD Counting sort

In lines 2-18 the algorithm is instantiating the variables it needs to perform calculations:

listmax contains the largest absolute value of the list and is used to calculate the passes over the data needed to complete sorting as well as the bitmask used to invert the sign bit.

min_bytes (1) Represents the total number of iterations over the data (rounds of integer sorting) that wil be needed. This is calculated by calling the function *int_bytes()* (5.1.2) with **listmax** and the selected base.
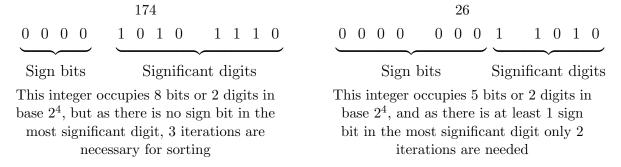
Ensuring that this algorithm properly sorts integers with respect to their sign requires that the most significant digit sorted includes at least one sign bit (see 4.2). Asserting this involves observing whether the quantity of bits needed to express **listmax** is divisible by the selected base with or without a remainder.

If there is a remainder then the leftmost digit of **listmax**, when expressed in the selected radix (2^{base}), includes at least 1 sign bit and thus all negative/positive integers will be properly sorted when this digit is iterated over.

If there is no remainder then the sign bit is not included in the leftmost digit of **listmax** as all bits in the leftmost digit are needed to express the magnitude of the digit. This signals that an extra iteration needs to be added to ensure negative values are properly sorted.

This extra iteration does not get added on if the quantity of digits in binary of **listmax** and the system maximum integer are equal as this would result in a rightshift of more bits than there are in an integer resulting in an error in RPython go into detail? it's kind of pointless. In this case the additional pass is not needed anyway as the leftmost bit is always the sign bit and so will definititely be included in sorting.

Two integers expressed in binary, the sign bits are highlighted and the binary digits are split into groups of 4 illustrating the number of iterations that would be needed if they were the maximum absolute value of an input list

This integer occupies 8 bits or 2 digits in base 2^4, but as there is no sign bit in the most significant digit, 3 iterations are necessary for sorting

This integer occupies 5 bits or 2 digits in base 2^4, and as there is at least 1 sign bit in the most significant digit only 2 iterations are needed

uint_63 This is a bitmask used to invert the sign of the inputs. In binary this value is represented by 0 in all digits needed to express **listmax** in two's complement

23

excluding the sign bit and a 1 in all other digits. This is calculated using an integer, 1, lefshifted a by a value equal to the quantity of digits in binary of **listmax** minus 1 (to account for 1 already containing a digit). Subtracting 1 from this value produces an integer that, represented in binary, contains a 1 in all digits needed to express **listmax** and 0 in every digit to the left. Thus the inverse of this value, calculated by bitwise not (5.2.4), can be used in conjuction with a bitwise exclusive or operation (5.2.5) to invert the sign bit and all digits to the left without altering the magnitude of .

```python
def sort(self):
    if self.listlength < 2:
        return
    listmax = self.list_abs_max(checkorder=True)
    min_bytes = int_bytes(listmax, self.base)
    if self.ordered == True:
        return
    if self.reverseOrdered == True:
        self.reverseSlice()
        return

    bitno = int(int_bytes(listmax, 1))
    uint_63 = ~((1 << bitno - 1) - 1)

    if bitno % self.base == 0 and bitno != int_bytes((-sys.maxint) -
        1, 1):
        min_bytes += 1

    counts = [[0 for _ in xrange(self.radix)] for _ in xrange(
        min_bytes)]

    for num in self.list:
        for i in xrange(min_bytes):
            shift = (self.base) * i
            sortkey = (num) ^ uint_63
            val = (sortkey >> shift) & self.radix - 1
            counts[i][val] += 1

    skip = []
    for i in xrange(min_bytes):
        for j in xrange(1, self.radix):
            if counts[i][j] == self.listlength:
                skip.append(i)
            counts[i][j] += counts[i][j - 1]
    temp_list = [0 for _ in xrange(self.listlength)]
    for i in xrange(min_bytes):
        if i in skip:
            continue
        shift = (self.base) * i
        for j in xrange(self.listlength - 1, -1, -1):
            num = self.list[j]
            sortkey = (num) ^ uint_63
            val = (sortkey >> shift) & self.radix - 1
            temp_list[counts[i][val] - 1] = self.list[j]
            counts[i][val] -= 1
        self.setslice(temp_list, 0)
```

LSD counting sort