

# Contents

<b>1</b>	<b>Design</b>	<b>3</b>
1.1	Pseudocode . . . . .	3
1.1.1	Integer Sorting . . . . .	3
1.1.2	Radix Sort . . . . .	5
1.2	Time complexity . . . . .	7
1.2.1	Integer Sorting . . . . .	7
1.2.2	Radix Sort . . . . .	7
1.3	Radix Selection . . . . .	8
1.3.1	Explanation . . . . .	8
1.3.2	Implementation . . . . .	9
1.4	Negative Integers . . . . .	9
1.4.1	Two's Complement . . . . .	9
1.4.2	Negatives in RPython . . . . .	9
1.4.3	Solution . . . . .	10
<b>2</b>	<b>Implementation</b>	<b>11</b>
2.1	Extra Methods and Classes . . . . .	11
2.1.1	Function absolute . . . . .	11
2.1.2	Function make_radixsort_class . . . . .	11
2.1.3	Functions setitem and setslice . . . . .	11
2.1.4	Function int_digits . . . . .	12
2.1.5	Class Radixsort . . . . .	13
2.1.6	Function list_abs_max . . . . .	13
2.1.7	Function insertion_sort . . . . .	14
2.1.8	Function reverse_slice . . . . .	14
2.2	Bitwise operations . . . . .	15
2.2.1	Left shift . . . . .	15
2.2.2	Right shift . . . . .	16
2.2.3	And . . . . .	17
2.2.4	Not . . . . .	17
2.2.5	Exclusive Or . . . . .	18
2.3	Optimizations . . . . .	19
2.3.1	Streamlining count lists . . . . .	19
2.3.2	Order checking . . . . .	20
2.3.3	Skipping iterations . . . . .	21
2.3.4	Insertion sort . . . . .	22
2.4	Algorithms . . . . .	23
2.4.1	Shared Variables . . . . .	23
2.4.2	LSD Counting sort . . . . .	24
2.4.3	LSD Pigeonhole sort . . . . .	26
2.4.4	MSD Counting sort . . . . .	27
2.4.5	MSD Pigeonhole sort . . . . .	29

<b>3</b>	<b>Testing</b>	<b>31</b>
3.1	Test Criteria . . . . .	31
3.1.1	Using PyPy for testing . . . . .	31
3.1.2	Methodology . . . . .	32
3.1.3	Preliminary testing . . . . .	33
3.1.4	Test criteria . . . . .	36
3.2	Test results and analysis . . . . .	37
3.2.1	Varying list length . . . . .	37
3.2.2	Varying base . . . . .	39
3.2.3	Categorical tests . . . . .	41
3.2.4	Testing evaluation . . . . .	44

# 1 Design

## 1.1 Pseudocode

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 integer keys.

### 1.1.1 Integer Sorting

Integer sorting is a method of sorting a list of inputs by identifying the integer key of each value of the input and using that to identify its position in the output. Because the sorting is performed based on the integer key and not based on comparisons to other list items, it is non-comparative. Integer sorting is the underlying sorting method that is applied to each digit of the input array.

#### 1.1.1.1 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 integer key 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:  $B \leftarrow [1..k]$  ▷ Buckets array
2: for  $i \leftarrow 1$  to  $k$  do
3:    $B[i] \leftarrow []$  ▷ Each bucket contains empty array
4: end for
5: for  $j \leftarrow 1$  to  $k$  do
6:   append  $A[j]$  to  $B[A[j]]$  ▷ Move item to bucket
7: end for
8:  $c \leftarrow 1$  ▷ Insertion index of items into A
9: for  $i \leftarrow 1$  to  $k$  do
10:  for all item in  $B[i]$  do
11:     $A[c] \leftarrow item$  ▷ Move items from bucket to A
12:     $c \leftarrow c + 1$  ▷ Increment insertion index
13:  end for
14: end for
15: return  $A$ 
```

---

### 1.1.1.2 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 integer key of length  $k$ .  $C$  is instantiated containing 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 integer key of the current item is incremented (7), producing a histogram of the frequency of each key. This count list is then accumulated such that each item is added to the sum of the previous items (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 key of the current item (13), to ensure that values do 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: $B \leftarrow [1..n]$	▷ Output array
2: $C \leftarrow [1..k]$	▷ Count array
3: <b>for</b> $i \leftarrow 1$ to $k$ <b>do</b>	
4: $C \leftarrow 0$	
5: <b>end for</b>	
6: <b>for</b> $j \leftarrow 1$ to $n$ <b>do</b>	
7: $C[A[j]] \leftarrow C[A[j]] + 1$	▷ Count occurrence of value
8: <b>end for</b>	
9: <b>for</b> $i \leftarrow 2$ to $k$ <b>do</b>	
10: $C[i] \leftarrow C[i] + C[i - 1]$	▷ Accumulate Count array
11: <b>end for</b>	
12: <b>for</b> $j \leftarrow n$ to $1$ by $-1$ <b>do</b>	
13: $B[C[A[j]]] \leftarrow A[j]$	▷ Move item from input to output array
14: $C[A[j]] \leftarrow C[A[j]] - 1$	▷ Decrement count array
15: <b>end for</b>	
16: <b>return</b> $B$	

---

### 1.1.2 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).

#### 1.1.2.1 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 digit in the chosen radix of the largest absolute value  $m$  (3).

This sorting process is applied first at the least significant or rightmost digit and progresses 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.

---

**Algorithm 3** LSD( $A, f()$ )

---

**Input:**  $A \Leftarrow$  Unsorted Array of length  $n$  with maximum absolute Value  $m$

**Input:**  $f() \Leftarrow$  Sorting function  $\triangleright$  Pigeonhole or Counting Sort

**Output:** Sorted Array

1:  $d \Leftarrow$  Number of digits of  $m$   $\triangleright$  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:    $A \Leftarrow f(A(i))$   $\triangleright$  Sorts Array using sort function

5: **end for**

6: **return**  $A$

---

### 1.1.2.2 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 integer key 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). Each slice on which  $MSD()$  is called that does not meet the termination condition must wait until all sublists return, which are then placed together in  $A$  and finally returned (15).

---

#### 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  $A$ 

```

---

## 1.2 Time complexity

### 1.2.1 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

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

### 1.2.2 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  $k$  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.

### 1.3 Radix Selection

Radix sort utilises multiple rounds of integer sorting across the digits of the input list, therefore, it is important to create an effective method of separating an integer into its component digits. When expressing a number, requires the selection of a numerical base, this is the number of unique values that can be displayed in each digit. 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.

#### 1.3.1 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 this project as each single  $2^n$  digit utilises a maximum of  $n$  bits and each subsequent digit utilises the next  $n$  bits.

The main consideration when selecting a numerical base for Radix sort is the trade off between iterations of the integer sorting method, minimized by selecting a higher base, and speed of each iteration of integer sorting, minimized by selecting a lower base. There is technically no upper bound for  $n$  when taking a theoretical approach, however in practice the constraints of a physical memory limit and larger overhead costs limit radix sort algorithm to reasonably small values of  $n$ . This project includes values in the range [2..16] as a value of 18 (and presumably anything above) performed so poorly the charts generated were unreadable for any other base.

The reason performance suffers with a higher radix being used is that each item in the range [1.. $n$ ] represents a list item in the integer sorting method (count list items / buckets) and each list item requires a memory address and a certain allocation of memory when being instantiated, each list item requires access to a location in the count list/bucket list during the building stage and the count/bucket list requires iterating over when rebuilding the input list. Whilst a higher value of  $n$  is desirable to reduce iterations over the input, any increase causes an exponential increase in the these overhead costs incentivising a lower  $n$ .

The effect of larger values of  $n$  being more costly is highlighted further when considering the effect of cache misses on performance. A cache is a small piece of high speed memory that can be accessed much faster than main memory, however this memory is much smaller than main memory and so can only store limited amounts of data. The number of successful reads of a count/bucket list item from cache can greatly improve the performance of an algorithm, and when considering radix sort, a higher  $n$  value decreases the ability of the system to be stored in the cache. Lamarca and Ladner use a  $n = 16$  in their implementation of radix sort to minimize cache misses and instruction count [3]



### 1.3.2 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 (see 2.2).

$$\begin{array}{cccc} \underbrace{1 \ 1 \ 1 \ 0} & \underbrace{1 \ 0 \ 1 \ 0} & \underbrace{0 \ 1 \ 1 \ 0} & \underbrace{0 \ 0 \ 0 \ 0} \\ \text{Digit 4} & \text{Digit 3} & \text{Digit 2} & \text{Digit 1} \\ 14 & 10 & 6 & 0 \\ \text{The 4 digits in base } 2^4 \text{ of a 16 bit integer: } 60000 \end{array}$$

## 1.4 Negative Integers

### 1.4.1 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 sort using bitwise methods without any accommodation for negativity is that due to their leading 1, negative values will be incorrectly sorted as though they are larger than positive values. They will also be sorted into the correct order relative to other negative values (increasing from most negative to least negative), as can be seen in Table 1, which is sorted.

### 1.4.2 Negatives in RPython

Because the purpose of this project was to incorporate this sorting algorithm into the standard library for Pypy, 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 machine word size of the system 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 the sign of the integer and the remaining bits between the sign bit and the most significant bit 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 a value of  $-2^m$  where  $m$  is the machine word length and all following digits are additions of machine where  $n$  is the index of the bit counted from the rightmost bit and in the range  $[(m - 1)..0]$ , the sum of these values is the value of the integer.

This project handles digits with a radix of  $2^n$  with  $n$  being some integer, and in binary representation this means each digit requires  $n$  bits. This causes challenges when handling negatives as the most significant digit of integers that are made up of a number of bits not divisible by  $n$  includes sign bits and integers that are made up of a number of digits divisible by  $n$  do not include a sign bit.

### 1.4.3 Solution

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 use a bitwise exclusive or (2.2.5) to flip all bits to the left of it. The effect of this is that when performing a radix sort across all 64 bits of the integer list is that the order is now corrected as negative values closer to 0 retain their higher position relative to more negative values but are now always sorted below positive values.

In order to reduce unnecessary iterations, radix sort only iterates a number of times such that the number of bits sorted across is at minimum the number of bits needed to express the absolute maximum value of the list including 1 sign bit, rounded up of course to the nearest multiple of  $n$ . In cases where the number of bits needed is divisible by  $n$ , it means the algorithm performs an addition iteration exclusively on sign bits whereas in the inverse case there is no increase in iteration count.

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 interpreted in unsigned binary and two's complement

## 2 Implementation

### 2.1 Extra Methods and Classes

The algorithm relies upon a number of helper methods to operate effectively and must be contained in a specific class structure to allow for it's inclusion into PyPy. This section explains the structure of the class used and contains documentation for all of the helper methods that are used by the algorithms.

#### 2.1.1 Function absolute

Allows for the calculation of absolute values without overflow issues specifically caused by using the minimum possible value as the positive of this value requires an extra bit to express and thus causes an overflow error.

---

```
import sys

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

---

#### 2.1.2 Function make\_radixsort\_class

The container function that instantiates all other functions, this is modelled on the existing function in listsort.py [1] to minimize compatability issues when including this project in the pypy library.

---

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

---

#### 2.1.3 Functions setitem and setslice

These functions have been imported from the existing listsort.py file within rlib [1]. 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.

---

```
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
```

---

#### 2.1.4 Function `int_digits`

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  $2^n$  excluding the sign bit which is not factored in when calculating the return value of this function but is accounted for elsewhere (see 2.4.1).

This function repeatedly rightshifts the input value by a number of bits starting at the chosen base and increasing by the same on each repeat. This is performed until the rightshifted input is equal to 0 or  $-1$  if the input is positive or negative respectively, which indicates that the number of bits the input has been shifted by are equal to or greater than the number of bits needed to express it. The number of repeats ( $l$ ) is returned as this represents the number of digits needed to express the input value.

On each repeat the current value for the rightshifted input is compared to the next value to ensure that the next value is closer to 0, this is done because in RPython rightshifting an input by  $n$  actually performs the input rightshifted by the  $n \bmod s$ , where  $s$  is the machine word size. For inputs occupying a number of bits in the range  $s - b < s$ , the smallest value by which the input must to be shifted that exceeds or equals the number of bits of the input is greater than  $s$ , triggering the modulo operation and thus resulting in a larger value than the previous iteration of the loop and thus the exit condition of the loop is never met. Due to these factors an extra exit condition has been included should the next rightshifted input value be further from 0 than the current one.

Intuitively it would seem the use of logarithms would be a more prudent and concise way of performing this function, as taking  $\log_r i$  (where  $r$  is the chosen radix) and rounding up would give the same result as this function. However the inbuilt logarithm function of RPython does not allow the base to be specified forcing the use of  $\frac{\log i}{\log r}$ . Implementation of this solution caused occasional floating point errors when rounding, these errors occasionally allowed an incorrect calculation of a bit mask resulting in an incorrect sort, namely when the machine word length and number of bits of the input value were divisible by the chosen base without remainder. Whilst this situation is extremely rare and it was possible to hard-code a solution for these cases, including a systematic calculation error in a function was deemed too risky.

---

```
def int_digits(i, base):
    l = 1
    v = 0 if i >= 0 else -1
    prev = i >> (l * base)
    while prev != v:
        l += 1
        new = i >> (l * base)
        if absolute(new) < absolute(prev):
            prev = i >> (l * base)
        else:
            return l
    return l
```

---

### 2.1.5 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 functions *setitem* and *setslice* are 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)
```

---

### 2.1.6 Function list\_abs\_max

This function returns the largest absolute value of the input list as well as identifying whether the list is sorted in ascending 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 the value that will contain the most digits and is thus the largest absolute value.

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.

---

```
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
```

---

### 2.1.7 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 demonstrated in [3.1.3.1](#)

---

```
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)
```

---

### 2.1.8 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).

---

```
def reverse_slice(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
```

---

## 2.2 Bitwise operations

Bitwise operations operate on binary numerals at the level of individual bits, they allow for fast and consistent manipulation that bypasses arithmetic functions. Through the use of bitwise operations, radix sort can be performed using different numeral bases, meaning a different 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 particularly 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 1.3.1) 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  $[-2^{63} .. 2^{63}-1]$ . 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 1.3 and handle negative integers in 1.4.2 but in this section I will cover the implementation of the bitshift operations used in RPython.

### 2.2.1 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:  $n \ll s$  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 accommodations necessary to sort negative integers (further explained in 2.2.5).

Value	7	6	5	4	3	2	1	0
13	0	0	0	0	1	1	0	1
$13 \ll 2$	0	0	1	1	0	1	0	0

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

### 2.2.2 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  $n \gg s$  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 2.2.3). 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
$13 \gg 2$	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
$-2^{61} \gg 2$	1	1	1	1	1	0	0	0

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



### 2.2.3 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 binary 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 2.2.2) 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  $2^4$

### 2.2.4 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 2.2.5

Value	7	6	5	4	3	2	1	0
102	0	1	1	0	0	1	1	0
$\sim 102$	1	0	0	1	1	0	0	1

Table 6: Demonstration of ‘Not’ performed on 102

All bits have been inverted

### 2.2.5 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 1.4.1 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 2.1.4) 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 2.2.4) so that it instead contains 0 in the bits representing the max input and 1 in all other bits. As explained in 1.4.2, 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 1.4.2, 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
$\sim 15$	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  $\sim 15$   
All bits in range [63..8] now equal 0

## 2.3 Optimizations

Many of the algorithms developed for this project differ in substantial ways from the pseudocode outlined in section 1, these differences arise from increasing the performance of the algorithm or adapting them to the purpose of implementation in RPython. In this section I will explain the function and reasoning of these differences.

### 2.3.1 Streamlining count lists

LSD counting sort involves sorting from the least to most significant digit (1.1.2.1) using counting sort (1.1.1.2). This implies iterating twice over the input list for each digit: first whilst counting the occurrence of the values at the digit and a second time when moving the input list items into the output list.

A suggestion by Friend [2] is to combine the counting and moving operations into the same iteration over the input list. This is possible because the relative position of list items does not influence the frequency of the integer keys at each digit, the count list would be identical no matter the order of the list.

The effect this has is that it reduces the number of iterations needed to complete radix sort by half, though 1 initial extra iteration is needed to establish the first count list. Though this requires an extra  $2^n$  space to store the additional count list.

In terms of implementation, an initial pass over the data counts the occurrence of each integer key at the least significant digit, then the algorithm begins a loop over the remaining digits. The first task is the accumulation of the count list, then the input list is iterated over, where the key value at the current digit is used to identify the position of each item in the output list and the next digit it used to create a second count list. Before the next digit is examined, the count list must be overwritten with the second count list, all items of which can be set to 0 during the accumulation stage.

---

```
count = [0 for _ in xrange(self.radix)]; nextcount = count[:]
for j in xrange(self.list_length):
    masked_input = self.list[j] ^ bit_mask
    curr_digit = (masked_input) & (self.radix - 1)
    count[curr_digit] += 1
for i in xrange(list_max_digits):
    for j in xrange(1, self.radix):
        count[j] += count[j - 1]
        nextcount[j] = 0
    nextcount[0] = 0

    count[curr_digit] -= 1
    if i < list_max_digits-1:
        nextcount[next_curr_digit] += 1
```

---

Count list accumulation using two count lists  
This code is simplified, see 2.4.2

### 2.3.2 Order checking

Radix sort has no inherent 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 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).

---

```
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 is 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. This type of optimization is not suitable for use in LSD sorts as the chance of the entire list being sorted by a single digit is extremely low.

Testing involving the comparison of the performance of algorithms including and excluding order checking showed that MSD radix sorts do sometimes create sorted sublists but skipping these sublists rarely results in a lower runtime. The testing process is further explored in [3.1.3.3](#)

### 2.3.3 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 algorithm 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

### 2.3.4 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 often highly inefficient when compared to other algorithms to perform radix sort.

This issue becomes 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 which 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.

---

```
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

## 2.4 Algorithms

### 2.4.1 Shared Variables

At the start of each algorithm the following variables are instantiated, they are integral to the a correct radix sort and are identical across all implementations.

**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.

**list\_max\_digits** Represents the total number of iterations over the data (rounds of integer sorting) that will be needed. This is calculated by calling the function *int\_bytes()* (2.1.4) with **list\_max** 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 1.4.2). Identifying if this is the case involves observing whether the quantity of bits needed to express **list\_max** is divisible by the selected base with or without a remainder.

If there is a remainder then the leftmost digit of **list\_max**, 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 **list\_max** 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 **list\_max** 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 this case the additional pass is not needed anyway as the leftmost bit is always the sign bit and so will definitely be included in sorting.

Below are 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

174				26			
0	0	0	0	1	0	1	0
0	0	0	0	1	1	1	0
Sign bits				Sign bits			
Significant digits				Significant digits			
This integer occupies 8 bits or 2 digits in base $2^4$ , but as there is no sign bit in the most significant digit, <b>3</b> 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 <b>2</b> iterations are needed			

**bit\_mask** A bitmask used to invert the sign of the inputs. In binary this value is represented by 0 in all digits needed to express **list\_max** in two's complement excluding the sign bit and a 1 in all other digits. This is calculated using an integer, 1, leftshifted a by a value equal to the quantity of digits in binary of **list\_max** 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 **list\_max** and 0 in every digit to the left. Thus the inverse of this value, calculated by bitwise not (2.2.4), can be used in conjunction with a bitwise exclusive or operation (2.2.5) to invert the sign bit and all digits to the left without altering the magnitude of the input value.

---

```

1  def sort(self):
2      if self.list_length < 2:
3          return
4      list_max = self.list_abs_max()
5      list_max_digits = int_digits(list_max, self.base)
6      if self.ordered:
7          return
8      if self.reverse_ordered:
9          self.reverse_slice()
10         return
11     max_bits = int(int_digits(list_max, 1))
12     bit_mask = ~((1 << max_bits) - 1)
13     if max_bits % self.base == 0 and max_bits != int_digits(
14         (-sys.maxint) - 1, 1
15     ):
16         list_max_digits += 1

```

---

Shared variables

## 2.4.2 LSD Counting sort

This algorithm sorts from least to most significant digit, performing *counting sort* on each digit but differs from 1.1.1.2 as the method is split into the counting and moving processes to reduce iterations over the input list (see 2.3.1)

**Counting integer keys** A 2-dimensional list is instantiated, equal in length to **list\_max\_digits** (the number of digits needed to express **list\_max** in the selected numerical base) where each item is a list equal in length to the radix (the number of possible unique keys for each digit), filled with the integer 0.

The second level of lists store the frequency of occurrence of each possible integer key for each item in the input list across a single digit with the first item in the list storing the rightmost/least significant digit.

The 2 dimensional nature of this list *counts* allows it to store the frequency of each possible integer key at each digit of the input list, meaning the input list needs only be iterated over once to complete the counting process (as stated in 2.3.1).



**Count list accumulation** The counts lists are accumulated, such that the value at each index becomes the sum of all values from 0 up to the index. Before this, each count item is compared to the input list length as the two values being equal indicates sorting is not needed at this digit so the digit index is added to the skip list (see 2.3.3)

**Performing sorting** Finally the items are sorted based on the values in the count lists, but this operation requires  $\mathcal{O}(n)$  extra space, so *temp\_list* is instantiated as a copy of the input list. After sorting the list across a digit into the temp list, the *setslice* method is applied, overwriting the entire input list.

---

```

1  def sort(self):
2      #Shared variables
3      counts = [[0 for _ in xrange(self.radix)] for _ in xrange(
4          list_max_digits)]
5      for j in xrange(self.list_length):
6          masked_input = self.list[j] ^ bit_mask
7          for i in xrange(list_max_digits):
8              shift = (self.base) * i
9              curr_digit = ((masked_input >> shift)) & self.radix - 1
10             counts[i][curr_digit] += 1
11
12     skip = []
13     for i in xrange(list_max_digits):
14         for j in xrange(1, self.radix):
15             if counts[i][j] == self.list_length:
16                 skip.append(i)
17                 counts[i][j] += counts[i][j - 1]
18     temp_list = self.list[:]
19
20     for i in xrange(list_max_digits):
21         if i in skip:
22             continue
23         shift = (self.base) * i
24         for j in xrange(self.list_length - 1, -1, -1):
25             masked_input = (self.list[j]) ^ bit_mask
26             curr_digit = ((masked_input >> shift)) & self.radix - 1
27             temp_list[counts[i][curr_digit] - 1] = self.list[j]
28             counts[i][curr_digit] -= 1
29     self.setslice(temp_list)

```

---

LSD counting sort

### 2.4.3 LSD Pigeonhole sort

**bucket\_list** This two dimensional list contains a quantity of empty lists equal to the radix. Input list items are moved into the bucket with the index equal to the value of the digit at a given position.

Because the frequency of each digit is not known, the sublists must be instantiated empty and the append method is used to insert list items.

**skip iteration check** After moving the list items into the buckets, a list comprehension is used to count the number of non empty buckets (line 27). If this value is 1 this indicates that the list order will not change by unpacking the buckets to overwrite the input list and so the bucket containing the single non empty list is overwritten with an empty list and the next digit is processed.

**Overwriting the input list** The bucket list is iterated over and any non empty buckets are used to overwrite the input list before being themselves overwritten with an empty list. The list length of the buckets is accumulated and used as the start index of the *setslice* method at this refers to the number of inputs already overwritten.

---

```
1  def sort(self):
2      #Shared variables
3      bucket_list = [[] for _ in xrange(self.radix)]
4
5      for i in xrange(list_max_digits):
6          shift = (self.base) * i
7          for num in self.list:
8              masked_input = num ^ bit_mask
9              curr_digit = ((masked_input >> shift)) & self.radix - 1
10             bucket_list[curr_digit].append(num)
11         nonempty = [
12             (bdx, bucket)
13             for bdx, bucket in enumerate(bucket_list)
14             if bucket != []
15         ]
16         if len(nonempty) == 1:
17             bucket_list[nonempty[0][0]] = []
18             continue
19         index = 0
20         for bdx, bucket in enumerate(bucket_list):
21             if len(bucket) == 0: continue
22             self.setslice(bucket, index)
23             index += len(bucket)
24             bucket_list[bdx] = []
```

---

LSD pigeonhole sort

#### 2.4.4 MSD Counting sort

MSD Radix sort, as outlined in 1.1.2.2, sort operates by using integer sorting on each digit to divide the input list into sublists upon which a recursive call to MSD Radix sort is made to sort across the next digit of each sublist.

In order to avoid recursive calls, this project implements a method of storing only the start and end index of the sublists then iterating over this list of indexes when sorting over the next digit and sorting items within each index range only amongst each other.

**Sublist Indexes** This requires two index lists, the first is instantiated on line 4, and contains the indexes 0 and `list.length`, meaning the first sublist contains every item within the input list. The second is `temp_sublist_indexes` on line 7 which is instantiated as empty at the beging of the process of applying integer sorting to each digit.

The algorithm iterates over the digits starting at `list_max_digit` down to 0, meaning the most to least significant. For each digit the sublist indexes are iterated over and for every pair of indexes, the integer sorting method counting sort is applied. When the count list is being accumulated, any items containing a count value greater than 1 indicates that a new sublist must be created, it's start index is calculated by taking the start index of the current sublist added to the previous entry in the accumulated count list i.e. the total number of items within the sublist that preceed the new sublist and the end index is the start index added to the total number of items with the current integer key i.e. the number of items within the new sublist.

The indexes of the new sublists are appended to `list_max_digit`, meaning all of the sublist indexes for the next digit are calculated whilst a digit is being sorted and not that a sublist is sorted immediately after being identified as could be expected with a typical recursive implementation. At the end of the iteration over all `sublist_indexes` items, it is overwritten by `list_max_digit` which is then overwritten as empty on the next iteration of the digits.

When a sublist does not require sorting, it's indexes are not included in the sublist index list, this occurs when `slice_sorted` returns positive for sorted or reverse sorted, when `insertion_sort` is used or when a sublist contains only 1 item and this just means these items will no longer be included in the sorting of any subsequent digits. A sublist that has all items sorted into the same integer key simply has it's start and end index added to `temp_sublist_indexes` for sorting on the next digit.

**Insertion sort** The limit on line 15 is different for all bases and was identified using the method seen in 3.1.3.1

---

```

1      def sort(self):
2          #Shared variables
3          count = [0 for _ in xrange(self.radix)]
4          sublist_indexes = [(0, self.list_length)]
5          for k in xrange(list_max_digits - 1, -1, -1):
6              shift = k * self.base
7              temp_sublist_indexes = []
8              for start, end in sublist_indexes:
9                  if start + 1 == end: continue
10                 if (end - start) < Limit:
11                     self.insertion_sort(start, end)
12                     continue
13                 for idx in xrange(start, end):
14                     masked_input = (self.list[idx]) ^ bit_mask
15                     curr_digit = ((masked_input >> shift)) & self.radix - 1
16                     count[curr_digit] += 1
17                 if count[-1] == end - start:
18                     temp_sublist_indexes.append((start, end))
19                     count = [0 for _ in count]
20                     continue
21                 if count[0] > 1:
22                     temp_sublist_indexes.append((start, start + count[0]))
23                 for i in xrange(1, self.radix):
24                     if count[i] > 1:
25                         temp_sublist_indexes.append(
26                             (start + count[i-1], start + count[i] + count[i-1])
27                         )
28                     count[i] += count[i - 1]
29                 temp_list = [0 for _ in range(start, end)]
30                 for i in xrange(start, end):
31                     masked_input = (self.list[i]) ^ bit_mask
32                     curr_digit = ((masked_input >> shift)) & self.radix - 1
33                     temp_list[count[curr_digit] - 1] = self.list[i]
34                     count[curr_digit] -= 1
35                 count = [0 for _ in count]
36                 self.setslice(temp_list, start)
37                 sublist_indexes = list(temp_sublist_indexes)
38             if not sublist_indexes: return

```

---

MSD Counting sort

### 2.4.5 MSD Pigeonhole sort

Once again, the method of storing indexes of sublists and sorting within the index ranges is applied.

**Sublist Indexes** This works in largely the same way as MSD Counting sort, sublist indexes are iterated over and items in a sublist have the integer sorting method, pigeonhole sort applied to them wherein further sublist indexes are identified and added to *temp\_sublist\_indexes* to be sorted on the next iteration over the digits.

To calculate the indexes when sorting a sublist, first a cumulative value is instantiated at 0 (*index*, line 31). The buckets are then iterated over, non empty bucket is considered a sublist with the starting index equal to the start index of the current sublist added to *index* with the same for the end index plus the list length of the bucket. The list length of the bucket is added to the cumulative value *index* so that it contains the total quantity of items in all buckets iterated over thus far. This occurs during the loop that overwrites the input list with the buckets as well as clearing the bucket list so it contains only empty lists.

**Insertion sort** The threshold used for insertion sort is different for each base, and is different from those used for counting sort, see [3.1.3.1](#).

**Bucket List** To avoid repeated instantiation of *bucket\_list*, it is created once at the same stage as the shared variables as a list of empty lists with a length equal to the radix. Instead of creating the bucket list again multiple times, each bucket is only overwritten with an empty bucket if it is accessed and identified as non empty, this saves on the number of read/write calls.

---

```

1  def sort(self):
2      #Shared variables
3      sublist_indexes = [(0, self.list_length)]
4      bucket_list = [[] for _ in xrange(self.radix)]
5      for k in xrange(list_max_digits - 1, -1, -1):
6          shift = k * self.base
7          temp_indexes = []
8          for start, end in sublist_indexes:
9              if start + 1 == end:continue
10             if (end - start) < Limit:
11                 self.insertion_sort(start, end)
12                 continue
13             for i in xrange(start, end):
14                 masked_input = (self.list[i]) ^ bit_mask
15                 curr_digit = ((masked_input >> shift)) & self.radix - 1
16                 bucket_list[curr_digit].append(self.list[i])
17             nonempty = [
18                 (bdx, bucket)
19                 for (bdx, bucket) in enumerate(bucket_list)
20                 if bucket != []
21             ]
22             if len(nonempty) == 1:
23                 temp_indexes.append((start, end))
24                 bucket_list[nonempty[0][0]] = []
25                 continue
26             index = 0
27             for bdx, bucket in enumerate(bucket_list):
28                 if len(bucket) >= 1:
29                     temp_indexes.append(
30                         (start + index, start + index + len(bucket))
31                     )
32                     self.setslice(bucket, start + index)
33                     index += len(bucket)
34                     bucket_list[bdx] = []
35             sublist_indexes = temp_indexes
36             if not sublist_indexes: return

```

---

MSD Pigeonhole sort

## 3 Testing

### 3.1 Test Criteria

#### 3.1.1 Using PyPy for testing

When testing the algorithms designed for this project, it was necessary to design tests with the core project goal in mind, to design a more efficient integer sorting method for use within PyPy. It is therefore a criteria of primary importance that tests be performed on a version of PyPy that has been compiled with a custom sorting method that implements the algorithms designed as part of this project.

Testing with this criteria in mind, and testing continuously throughout the development process, ensured that any solutions produced were applicable to the goals of the project and ventures into solutions that may work on other platforms such as Python 3 or even Python 2 but not on RPython were short lived. For example an early solution of this project utilised the inbuilt methods *max()* and *min()*, which are not available in RPython at a later stage the stark difference in RPython handling rightshifting by values greater than the machine word size as the value modulo the machine word size whereas in all other versions of Python the full rightshift is performed, giving very different results to the helper method outlined in 2.1.4.

PyPy source code includes the file *rpython* used to compile and package PyPy, this file has the useful feature of allowing the compilation of a single file into an executable, which is helpful as compiling a single file takes less than 30 seconds whereas compiling a fully functioning version of PyPy can take up to 20 minutes. This feature was regularly utilised during the development process to test the correctness of solutions but is not particularly useful when testing performance for a number of reasons. Firstly it would compromise the applicability of test data to the goals of the project, it is not a version of PyPy after all, merely an executable file compiled using the same tools. Beyond this another problem exists though, it is not possible to design the executable in such a way that it can either generate new data for sorting or read generated data from files in a dynamic way. Whilst it is possible to use a library such as random to generate a random list, or to read from files to assemble a list, this must be performed outside of the code block run by RPython, it is instead run by the version of PyPy that is called to run the file *rpython* and the variables it generates are stored within the executable, meaning the first random list generated is always stored and the values read from a file remain the same even if the file is edited.

### 3.1.2 Methodology

Testing was performed by creating a python script to be run by a version of PyPy using the custom sort method to be evaluated. The testing process in the script is to generate a list within the specified parameters, begin a timer, perform the sort function then stop the timer and record the time taken. The timer used was the *time* library function *perf\_counter*, which uses a system performance counter and when measuring an interval, as in the script, returns highly accurate results.

The decision to use *perf\_counter* was made based on a recommendation that is shown when using the *timeit* module on PyPy, warning against the use of the module and instead presenting *perf\_counter* as an alternative.

The lists to be sorted are generated using various functions in the *numpy* and *random* libraries in the formats listed below and are generated to contain a specified quantity of items and contain values within a specified range.

**Nearly sorted** lists generate a sorted list and repeatedly swap items at adjacent indexes until the number of swaps equals 10% of the list length

**Few unique** generate a random list with a length  $1/10^{th}$  of the target length then randomly populates a list of the target length with the preliminary list.

**Random** generate list of random integers of the target length

Below is a demonstration of 50 items generated in these formats within the range [0..50]. For demonstration purposes the chance of items being swapped in nearly sorted has been increased.

./demographs/demo.png

A demonstration of a list generated in each format.

Each bar represents an integer, the order of the bars left to right represents the order of the items in indexes [0..49]



### 3.1.3 Preliminary testing

#### 3.1.3.1 Insertion sort threshold

MSD Radix sort uses insertion sort on sublists with a length below a certain threshold due to the overhead cost of repeatedly performing radix sort (see 2.3.4). Therefore, in order to implement insertion sort, testing was required to establish the optimal threshold to use.

This testing involved creating two versions of pypy that use MSD Radix sort, one of which never uses insertion sort and another that always uses it, and then recording and comparing the time taken for the two implementations to sort randomized lists of increasing lengths starting at 0.

The goal of this test was to identify a critical list length for which insertion sort is faster than MSD Radix for lists with fewer items and slower for lists with more items, so that small sublists of larger lists can be sorted more efficiently.

Two versions of PyPy were created for each base and integer sorting method, one of which used a threshold of 0 to ensure insertion sort was never used. The other version replaced the threshold check with a test of whether the length was greater than 0 which always returns true and triggers insertion sort, this was done instead of replacing the entire radix sort method with insertion sort as all the code executed prior to sorting (whether that be insertion or radix) is identical and therefore the results can be used to make a valid comparison of the two sorting methods being applied to a sublist.

A sample of testing results are shown in 1. Each point represents the average of 100 sample of the time to sort a list of a given length, also shown are the polynomial regression lines modelled using the method of least squares with the intersect labelled in red which is then taken as a generally acceptable threshold to use.

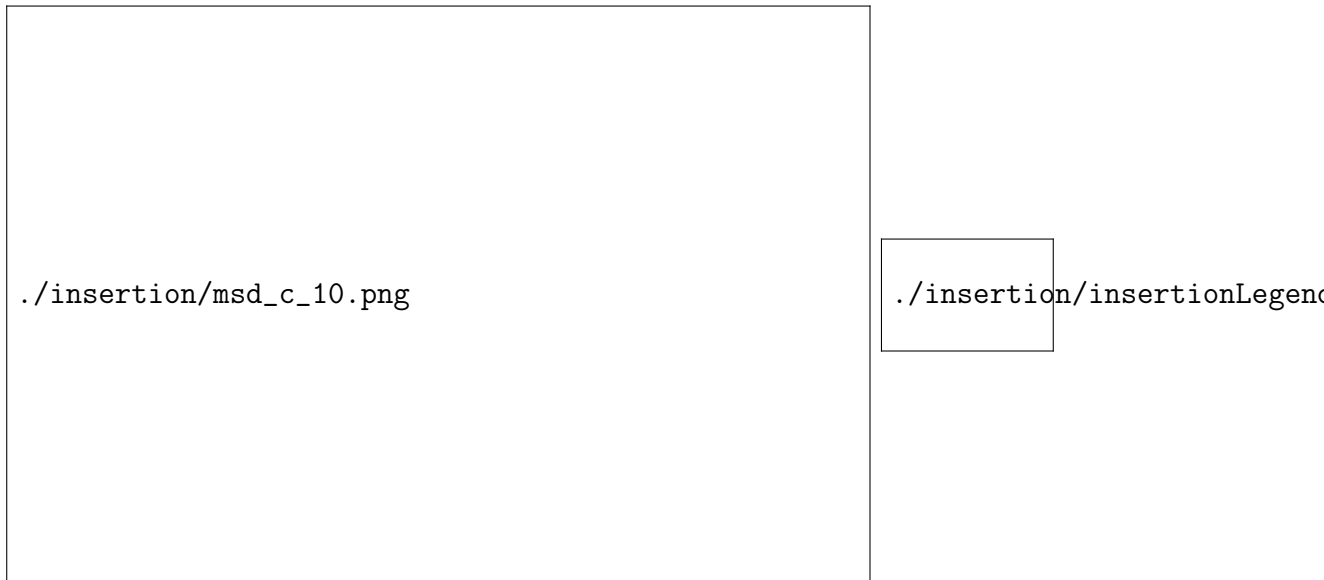


Figure 1: Time comparison for insertion sort and radix sort using counting sort, base  $2^{10}$

### 3.1.3.2 Profiling

A tool utilised to test the performance of solutions was a line profiler for Python, in this project the open source utility `line_profiler` [4] was used. This tool runs a python script and returns information about executed code within a function, namely the execution time of each line per call, the number of calls of each line, the total number of times a line was called and the time taken by calling the line as a percentage of the total execution time of the script.

It is important to note that this tool runs using Python, not PyPy and also that as this project demands a compiled piece of code written in RPython the results regarding time taken must not be taken as too strongly indicative of the performance of a solution. It was also necessary to make minor changes to the code such as replacing `xrange` with the equivalent `range` and `sys.maxint` with `sys.maxsize`.

The data provided by this tool was, however, very useful at identifying the correctness of solutions as the number of calls to a particular line would be identical whether run using `line_profiler` or with a compiled version of PyPy, provided the input list is identical. Using this information allowed me to ensure that the code was efficient and that unnecessary iterations over the input list were not made. As an example this tool allowed me to identify that pigeonhole sort was incorrectly performing an additional iteration by showing that all lists made up of positive integers triggered the condition of a single bucket containing all items.

Initially one of the aims of testing was to complete a profile of the memory usage of the sort methods, however this was deemed to not be feasible as memory profiling in PyPy is highly inaccurate, and the function `sys.getsizeof` is not included in PyPy causing an error for most memory profilers. According to the documentation, the reason for this is due to optimizations performed by the PyPy compiler that cause the size of most objects to be reported incorrectly, meaning any value returned by `sys.getsizeof` would be meaningless [5].

Line #	Hits	Time	Per Hit	% Time	Line Contents
127	5	1.6	0.3	0.0	<code>for i in range(list_max_digits):</code>
128	4	1.6	0.4	0.0	<code>shift = (self.base)* i</code>
129	40004	4158.5	0.1	11.2	<code>for num in self.list:</code>
130	40000	6395.9	0.2	17.2	<code>masked_input = num ^ bit_mask</code>
131	40000	9121.8	0.2	24.5	<code>curr_digit = (masked_input &gt;&gt; shift)&amp; self.radix - 1</code>
132	40000	8222.3	0.2	22.1	<code>bucket_list[curr_digit].append(num)</code>
133	8	117.4	14.7	0.3	<code>nonempty = [</code>
134					<code>(bdx, bucket)</code>
135	4	6.1	1.5	0.0	<code>for bdx, bucket in enumerate(bucket_list)</code>
136					<code>if bucket != []</code>
137					<code>]</code>
138	4	3.6	0.9	0.0	<code>if len(nonempty)== 1:</code>
139					<code>bucket_list[nonempty[0][0]] = []</code>
140					<code>continue</code>
141	4	1.0	0.3	0.0	<code>index = 0</code>
142	260	37.6	0.1	0.1	<code>for bdx, bucket in enumerate(bucket_list):</code>
143	256	41.8	0.2	0.1	<code>if len(bucket)== 0: continue</code>
144	200	257.8	1.3	0.7	<code>self.setslice(bucket, index)</code>
145	200	36.1	0.2	0.1	<code>index += len(bucket)</code>
146	200	37.1	0.2	0.1	<code>bucket_list[bdx] = []</code>

excerpt from line\_profiler output of LSD Counting sort

### 3.1.3.3 Order checking

As an extension of the sort check made at the start of each call to radix sort, in MSD sort, it is possible to check sublists for being sorted or reverse sorted after each iteration of integer sorting to identify lists that can be excluded from sorting on subsequent digits. Whether or not this strategy is effective is a trade off between the time saved from reduced iterations of integer sorting versus the time taken to complete this check. There is no similar optimization for LSD sorts as the list is never broken down into sublists and so the entire list must be checked for randomly becoming sorted which is such a rare occurrence that it is not worth considering.

---

```
def slice_sorted(self, start=0, end=-1):
    end = end if end > -1 else self.list_length
    assert start >= 0
    assert end >= 0
    sublist_sorted, sublist_reverse_sorted = True, True
    for i, el in enumerate(self.list[start + 1 : end]):
        if el < self.list[start + i]:
            sublist_sorted = False
            if sublist_reverse_sorted == False:
                return sublist_sorted, sublist_reverse_sorted
        if el > self.list[start + i]:
            sublist_reverse_sorted = False
            if sublist_sorted == False:
                return sublist_sorted, sublist_reverse_sorted
    return sublist_sorted, sublist_reverse_sorted
```

---

To evaluate the effectiveness of order checking, two versions of PyPy were compiled, one with the above function being used on every sublist and one without. These were then tested on lists of 1,000,000 on lists containing random, nearly sorted and few unique data. The results of the tests showed that in almost all cases the order checking has a detrimental or limited positive effect and therefore this function was not included in this project. The figure below shows two examples of these tests and are representative of the other results.

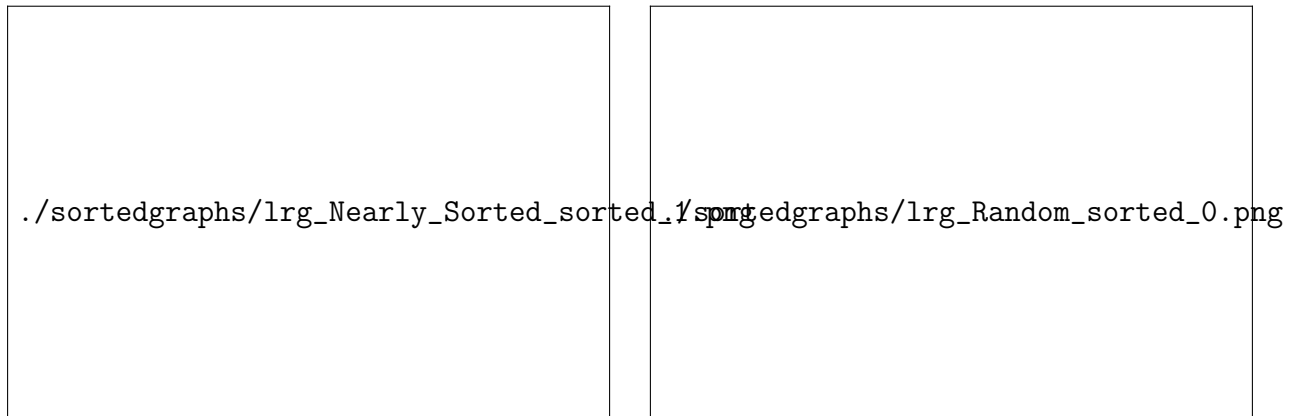


Figure 2: The effects of completing order checking across sublists. Listed below are the number of bases for which sort checking and not checking are the fastest option

### 3.1.4 Test criteria

The purpose of the testing was primarily to compare the efficacy of the solution in this project to Timsort and verify whether the project goal of producing a more optimal sort method had been achieved. Secondary goals of the tests were to identify which of the 4 solutions performs the best and which base as the most effective as well as apply the algorithms to a variety of lists to identify the situations in which the algorithms were strong or weak relative to one another and to Timsort.

As outlined in 1.2, Radix sort has a time complexity that depends on the list length  $n$  and the number of digits  $d$ , therefore a complete testing must evaluate the effect of these two variables on the runtime of the algorithms. Measuring the effect of altering  $n$  and  $d$  was performed in two separate ways, in the first each was isolated and tested as a continuous variable without altering the other, in the second 3 values were chosen for  $n$  as well as 4 ranges for list item values that contain different amounts of digits to test  $d$ .

The continuous testing of  $n$  involved generating lists with a value for  $n$  in the range  $[1..1,000,000]$  and with list values of up to the maximum value for integers to ensure as best as possible that  $d$  was constant. To test  $d$ , lists were generated with values constrained to the range  $[-2^r..2^r-1]$  where  $r$  varies from  $[1..63]$ , thereby increasing the value of  $d$  but keeping  $n$  constant.

The non-continuous testing evaluated the runtime of the algorithms of lists with a length of 10,000, 100,000 and 1,000,000 and containing items with values within the range  $[-2^r .. 2^r-1]$  where  $r$  is 16, 20, 32 and 63. These parameters were selected to provide tests that represent the smallest and largest extremes of the continuous tests as well as some tests in the middle.

Testing had to be split into continuous and non-continuous as performing a continuous evaluation of both  $n$  and  $d$  would require a test of all values of  $d$  performed for all values of  $n$  and the amount of tests necessary makes this unfeasible. Furthermore, the data created by this would span 3 dimensions and would be difficult to extract useful information from. The continuous tests are useful for identifying the differences between each numerical base used for the algorithms, but as they are primarily evaluating the effect of  $n$  and  $d$  on Radix sort, they are not suitable for making a comparison to Timsort, the sort method used in *listsort.py*.

As the non-continuous testing evaluates a range of both  $n$  and  $d$ , the non-continuous testing is also performed on lists generated in different format to evaluate each algorithms performance in the widest range of circumstances. The formats used are identified in 3.1.2 (Random, Nearly sorted and Few Unique). This type of testing was used to make a final comparison to Timsort, which is optimized for efficiency on lists with sorted sub-sections and therefore highly performant on nearly sorted lists.

## 3.2 Test results and analysis

### 3.2.1 Varying list length

This test shows the effect of list size on the performance of radix sort and very clearly highlights the difference between the bases used. Some bases produce results that stretch the size of the graph so much that it prevents meaningful information from being shown, hence some graphs have non-competitive bases omitted for readability.

To prevent the focus of this analysis being unfairly weighted towards lists of lengths closer to 1,000,000 where the difference between run times is larger and therefore more visible on graphs, where the results may be different to lower list length, and which may not represent the most common use case for this algorithm, these tests were also performed with lists of length 100,000.

#### 3.2.1.1 LSD Counting sort

The results show that at all small list lengths, sort times are comparable but above lengths of 65,536 bases 8 and 16, which were slightly faster, become slower and do not recover at any point up to 1,000,000. This critical value likely exists at this point as it is  $2^{16}$  and as 16 is divisible by both values and therefore is the point where another digit is needed for sorting, but all other bases also take slight jumps at this stage and this is the most pronounced increase on either graph which indicates there may be some sort of cache limitation at this value (similar to what is discussed in 1.3.1), alternatively, pypy may be optimizing the storage of lists into chunks of this size.

Base 6 is the slowest shown, performing similarly to 8 and 16 around 100,000 items but falls off as more items are added. These bases seem to be too small to be suitable for this type of radix sort and this result is compounded in the omitted bases 4 and 2 which was the worse performer.

Overall, the results show 10 is the best performing base here, but it is only narrowly better than the 12 and 14 which are slightly faster than bases 8 and 16 with 6 showing a clear deficit particularly at greater list lengths.

./lengthgraphs/lsc\_length.png

LSD Counting sort length test for 100,000 and 1,000,000 items  
bases 2, 4, 6 and 8 omitted

### 3.2.1.2 LSD Pigeonhole sort

The fastest base for LSD Pigeonhole varies a lot amongst the best performers as list length increases. Bases 16, 14, 12 and 10 appear intertwined from 200,000 to 1,000,000 items with 16 ahead at the 1,000,000 point but not by a large margin. Bases 8 and below are omitted from the 1,000,000 graph as they are notably slower in this range.

Though base 16 becomes the most effective at higher list lengths, it is not truthful to call it the best as it is nearly 10 times slower than base 10 for list lengths below 100,000. For this reason bases 10 and 12 would be the most suitable, as base 14 is also noticeably slower for lower lengths, albeit less pronounced than 16.

`./lengthgraphs/lsd_p_length.png`

LSD Pigeonhole sort length test for 100,000 and 1,000,000 items  
bases 2 and 4 omitted

### 3.2.1.3 MSD Counting sort

The results visible here are highly similar to those of LSD counting sort, base 6 becomes more performant than 8 and 16 in the lower range, but again lags behind in the higher range. Once again, base 10 performs best with similar grouping of the other bases.

`./lengthgraphs/msd_c_length.png`

MSD Counting sort length test for 100,000 and 1,000,000 items  
base 2 omitted

#### 3.2.1.4 MSD Pigeonhole sort

This set of results deviates significantly from the rest of this set, it shows that the larger bases are unsuitable for this implementation, with base 16 being the worst. This is the only iteration of this test in which base 2 wasn't the worst performer.

The ideal range appears to be around base 6, with 4 and 8 roughly equivalent in performance, as are 2 and 10, above which a base increase is detrimental.

./lengthgraphs/msd\_p\_length.png

MSD pigeonhole sort length test

#### 3.2.2 Varying base

This test examines lists of 1,000,000 items generated randomly within the range  $[-2^r..2^r-1]$  where  $r$  is the *data\_size* referenced on the x axis. It is designed to test the algorithm's response to an increasing number of digits.

This test is limited when applied to MSD radix sorts due to the algorithms ability to terminate sorting on smaller lists via insertion sort or sorting a list into a sublist of length 1, this means that typically a list of 1,000,000 items is sorted when  $20/base$  digits have been sorted as  $2^{20}$  is the smallest power of 2 greater than the list length. This is visible in the results as the sort times level out after *data\_size* and does restrict the usefulness of the results but does present an interesting benefit of MSD sorting.

##### 3.2.2.1 LSD Counting

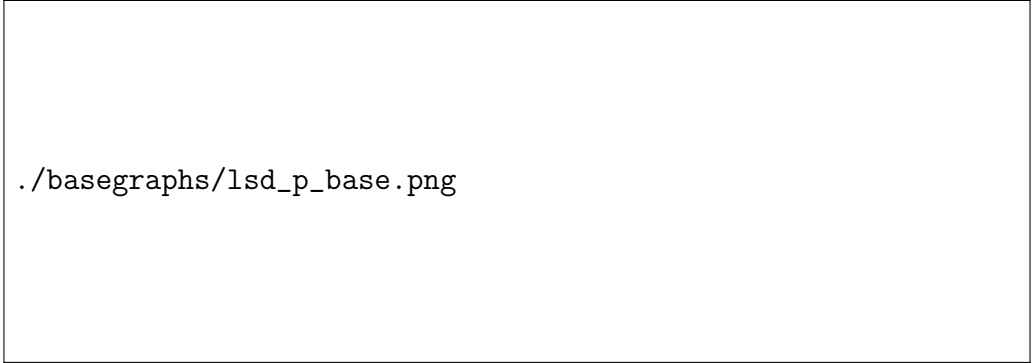
The results shown on this graph are so similar that any difference between them was completely unreadable if any of the other bases were not omitted, showing that any of the bases 8,10,12,16 are suitable. As is typical for this type of testing, the jump in runtime as *data\_size* reaches values divisible by each base is easily identifiable, showing the effect of each additional digit.

./basegraphs/lsc\_c\_base.png

LSD counting sort input limit test

### 3.2.2.2 LSD Pigeonhole

The results for LSD Pigeonhole sort show higher bases performing better than lower bases, with 16 and 14 performing the best and almost identically which indicates that the results for the list length results for base 14 (in which it was slower than base 12 and 10 at 1,000,000 items) may be anomalous.

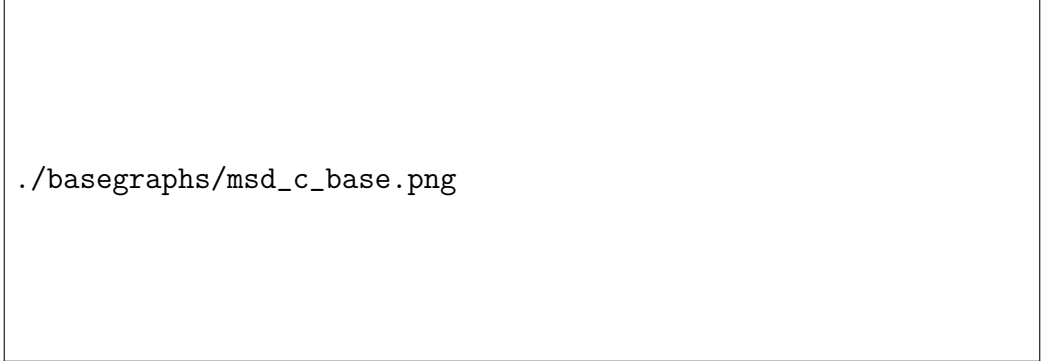


`./basegraphs/lsd_p_base.png`

LSD pigeonhole sort input limit test

### 3.2.2.3 MSD Counting

As already stated, the usefulness of results of this type of test for this algorithm is limited, but of the most performant algorithms, 10 and 12, the results for base 10 show a smaller delta between the maximum and minimum times with very similar times overall



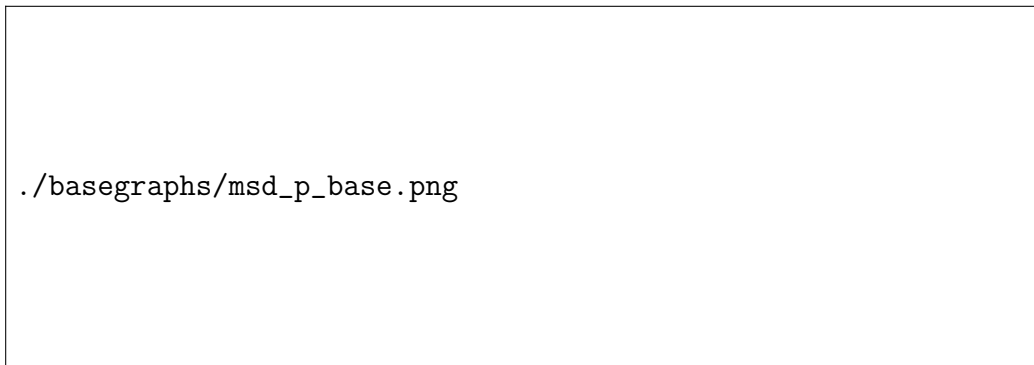
`./basegraphs/msd_c_base.png`

MSD counting sort input limit test



### 3.2.2.4 MSD Pigeonhole

MSD pigeonhole again has results quite dissimilar from the other algorithms, the increase *data\_size* has the smallest effect on the growth of run time and as the increases causes an additional digit's worth of sorting, the jump in run time is followed by a huge spike before falling again all in the range of values fitting onto the same digit. Drawing conclusions about any of these algorithms based on this data is almost impossible, but there isn't anything immediately wrong with bases 8 or 10, where base 8 is slightly slower but has smaller spikes in runtime.



MSD pigeonhole sort input limit test

### 3.2.3 Categorical tests

These tests were completed to provide an opportunity to make comparisons between the 4 algorithms, the 8 bases used and give insightn into their effectiveness relative to Timsort. Each image shows the results for a list length and data type identified in the title, a set of axes for each data size listed as a subtitle and on each axis is the 4 algorithms developed for this project split into a bar for each base alongside timsort farthest on the right which has a line stretching across the axis to aid comparison to each implementation.

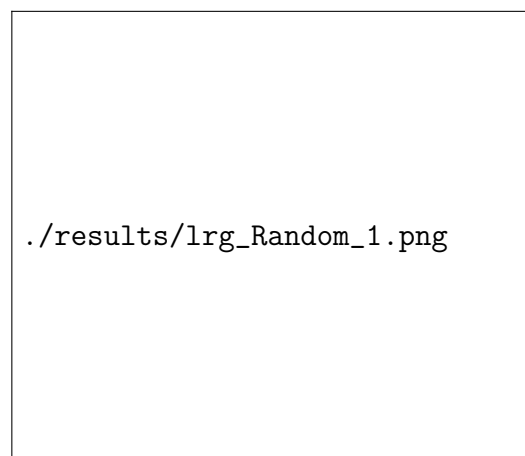
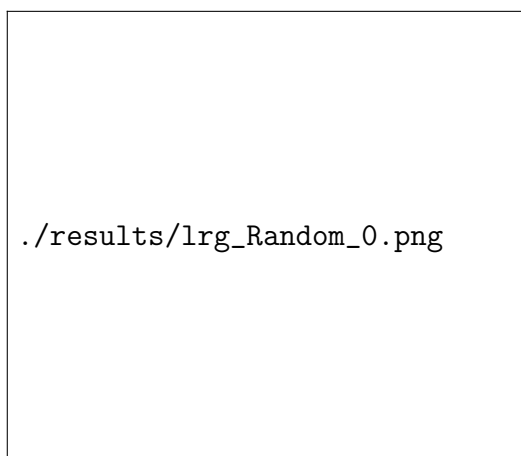


Figure 3: Sort times for categorical tests, List length, data type and maximum integer identified above, Sort method and base (2-16) listed below

### 3.2.3.1 Random

Random data also was used for the continuous tests and as expected, the results show similar results to the previous continuous tests for comparisons between the bases of each algorithm.

When comparing the algorithms against one another and timsort, the random data shows that all implementations of counting sort perform faster than timsort, as well as at least one base of each iteration of pigeonhole sort. The difference between the comparative nature of timsort and non-comparative radix sort is highlighted by observing LSD\_C for list length 1,000,000 (shown in 3); Timsort completes sorting in roughly 0.17s regardless of integer limit whereas LSD\_C sees smaller runtimes for smaller integer limits, showing the benefits of the fewer digits requiring counting sort in contrast with the lack of any reduction in the number of comparisons timsort must make.

### 3.2.3.2 Few Unique

Data containing few unique values was intended to simulate sorting a series of discrete values, a potential real world usage of the algorithm. 5 shows the percentage difference in sort time between sorting random and few unique data where each item is the average difference of all bases and a positive value indicates Few unique was faster. It shows there is typically a negligible difference between the two with the exception of certain higher data limits in longer lists for MSD sorts, particular counting sort. This may be due to the large number of identical items preventing insertion sort and only causes 10% difference from random data.

./results/med\_Few\_Unique\_1.png

items	int size	lsd c	lsd p	msd c	msd p	timsort
10000	large	-9.5%	0.3%	2.3%	3.3%	6.9%
	med	-7.2%	6.1%	3.0%	3.6%	5.3%
	small	-3.4%	7.6%	6.3%	1.8%	1.1%
	tiny	-1.8%	-0.3%	0.8%	6.5%	1.8%
100000	large	4.7%	-23.1%	18.2%	7.1%	5.2%
	med	7.2%	-10.4%	10.1%	9.0%	5.1%
	small	1.7%	2.1%	2.0%	16.4%	3.1%
	tiny	0.9%	-5.6%	11.8%	1.0%	4.8%
1000000	large	8.8%	-1.0%	-124.9%	10.4%	7.0%
	med	8.5%	-6.3%	-20.1%	-48.6%	3.9%
	small	0.9%	7.5%	0.0%	51.2%	2.5%
	tiny	2.9%	23.9%	-33.7%	20.2%	4.6%
mean		1.1%	0.1%	-10.4%	6.8%	4.3%

Figure 5: Comparison of few unique and random data, using average between equivalent bases

Figure 4: categorical tests for few unique data

### 3.2.3.3 Nearly sorted

This data type was selected to make a comparison between algorithms in a situation where timsort is at an advantage. As stated in the source code for timsort [1], the algorithm iterates over the input, continuously identifying sorted sequences and merging them with the already sorted items and as such it is highly efficient at sorting this data which is essentially a series of sorted sequences within a larger list.

The graphs shown in 6 and 8 have been pruned to omit all of LSD\_P as well as bases 2, 14 and 16 as the results for these algorithms are much slower than timsort and stretch the axis, making the graph unreadable. The remaining data is representative of all tests of this data type, with timsort performing exceptionally well compared to radix sort and being faster in every situation except on lists of 10,000 items where certain bases of MSD\_P and LSD\_C are a fraction faster. 7 Contains the difference in runtime between Random and nearly sorted lists, it shows Whilst timsort benefits from a 94% decrease in runtime when performed on nearly sorted lists compared to random, radix sort does not see the same benefit, though there is variation base on the implementation used. MSD sorts benefit from nearly sorted lists due to their use of insertion sort which completes in  $\mathcal{O}(n)$  time on sorted lists. LSD sorts do not employ this method and as such don't see much benefit. The discrepancy between pigeonhole and counting sort may be due to optimizations in PyPy involving moving items from one list into another in the same order, and there is some evidence for this such as LSD\_P having larger speed gains at smaller integer limits.

./results/lrg\_Nearly\_Sorted\_nolp\_0.png

Figure 6: Nearly sorted data test with LSD\_P, bases 2, 14, 16 pruned

items	int size	lsd c	lsd p	msd c	msd p	timsort
10000	large	-3.1%	-8.4%	-1.3%	32.3%	91.9%
	med	-3.6%	-9.7%	17.5%	52.0%	92.0%
	small	-4.5%	-5.1%	0.9%	43.5%	92.2%
	tiny	-1.8%	-3.9%	15.1%	39.6%	92.0%
100000	large	-0.2%	6.7%	8.6%	27.5%	94.4%
	med	-1.6%	10.8%	19.6%	43.1%	94.3%
	small	-3.7%	20.0%	16.4%	39.0%	94.3%
	tiny	-7.2%	32.4%	14.0%	31.1%	95.0%
1000000	large	0.3%	6.1%	35.4%	54.7%	95.3%
	med	1.4%	0.4%	23.0%	47.0%	95.1%
	small	-1.4%	22.6%	14.7%	40.5%	95.2%
	tiny	11.9%	32.9%	28.7%	41.8%	98.6%
mean		-1.1%	8.7%	16.0%	41.0%	94.3%

Figure 7: Comparison of Nearly sorted and random data, using average between equivalent bases. Positives indicate nearly sorted is faster

./results/sml\_Nearly\_Sorted\_nolp\_2.png

./results/sml\_Nearly\_Sorted\_nolp\_3.png

Figure 8: Nearly sorted data test showing list length 10,000

### 3.2.4 Testing evaluation

Making an empirical choice for the best algorithm is not possible in this situation, the variables identified during time complexity analysis,  $n$  and  $d$ , are independant of one another meaning an exhaustive performance test of both is not possible. Additionally, the categorical testing of different data types proves that the nature of the input list also affects sort times, meaning only sorting radom data would not paint the full picture.

Based on the idea that there are so many factors in play, and the uncertainty as to the relative importance of each factor, the best algorithm out of the choices presented in this project should be one that performs poorly in the fewest number of situations, that is to say it is unimportant to identify the best algorithm in any one of the tests and instead find an algorithm that performs at an acceptable level in as many as possible.

Based on this principle, the data from the categorical tests has been used to identify a suitable list of bases for each algorithm by analysing the bases in groups containing a single list length, data type and integer range. For each group, the bases that are deemed 'acceptable' are identified by performing k-means clustering of the sort time with 2 clusters, and selecting all items within the same cluster as the fastest base within the group, resulting in a smaller group that excludes the slowest base and bases significantly slower than the fastest. Counting the number of occurances of each base across all groups then shows how often the base performed at an acceptable level, with bases that show higher counts being good performers in a wider range of circumstances.

lsd c			lsd p		msd c		msd p	
rank	base	count	base	count	base	count	base	count
1	10	36	10	36	4	36	4	34
2	12	36	12	36	10	35	6	34
3	14	36	14	36	8	35	8	33
4	4	36	8	34	12	31	10	32
5	6	36	6	31	6	29	12	31
6	8	36	4	30	16	24	2	28
7	16	23	16	16	14	22	14	25
8	2	9	2	13	2	17	16	10

Combining this information with the suitable bases identified in the continuous testing, the bases that have been selected for use are listed overleaf, 36 situations were examined overall.

**MSD Pigeonhole - 6** Performed well in 34 situations, tied with base 4 but showed greater performance in the variable test of length, particularly at smaller list lengths. Admittedly performed poorly for small integer sizes on large list lengths in the category few unique, but this is anomalous as this base performed very well in all other situations.

**MSD Counting - 10** Deemed acceptable in 35 situations, with the exception being Few unique, medium int limit, length 1,000,000. This was selected over the best performer, base 4, which showed confusingly good results in categorical tests despite the poor performance in the continuous tests, for which it was rejected.

**LSD Pigeonhole - 12** Bases 10, 12 and 14 were acceptable in all situations, and performed very similarly on continuous tests so this decision was made arbitrarily so that the selection is more in line with the other methods.

**LSD Counting - 10** Bases 6, 4, 14, 12 and 10 were all acceptable in all situations, but of those base 10 was the best performer in the continuous tests.

Now these 4 final methods have been identified, comparing them is simple; omitting all results not for the four stated methods and timsort, across the 36 categories LSD counting sort was the best 20 times, MSD counting 5 and Timsort 11. Selecting only random lists produces the results of 9 and 3 for LSD C and MSD C respectively with 0 for Timsort. This indicates that LSD counting sort is the best option, that pigeonhole sort is generally inferior for this purpose and within the tests specified performed better than timsort.

## References

- [1] Carl Friedrich Bolz-Tereick. *listsort.py*. The PyPy Project. Apr. 1, 2022. URL: <https://foss.heptapod.net/pypy/pypy/-/blob/ab3f173b52ba0b51b155c372af40de3d85a855a7/rpython/rlib/listsort.py>.
- [2] Edward H. Friend. “Sorting on Electronic Computer Systems”. In: *J. ACM* 3.3 (July 1956), pp. 134–168. ISSN: 0004-5411. DOI: [10.1145/320831.320833](https://doi.org/10.1145/320831.320833). URL: <https://doi.org/10.1145/320831.320833>.
- [3] Anthony LaMarca and Richard E Ladner. “The Influence of Caches on the Performance of Sorting”. In: *Journal of Algorithms* 31.1 (1999), pp. 66–104. ISSN: 0196-6774. DOI: <https://doi.org/10.1006/jagm.1998.0985>. URL: <https://www.sciencedirect.com/science/article/pii/S0196677498909853>.
- [4] *line\_profiler*. Pyutils. Nov. 2, 2023. URL: [https://github.com/pyutils/line\\_profiler](https://github.com/pyutils/line_profiler).
- [5] *PyPy Documentation*. The PyPy Project. Mar. 26, 2024. URL: [https://doc.pypy.org/en/latest/cpython\\_differences.html](https://doc.pypy.org/en/latest/cpython_differences.html).
- [6] *RPython Language - RPython Documentation*. The PyPy Project. Jan. 4, 2024. URL: <https://rpython.readthedocs.io/en/latest/rpython.html#object-restrictions>.