The Diverting Fast Radix Algorithm

Stuart Thiel

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

February 2021

© Stuart Thiel, 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Stuart Thiel**
Entitled: **The Diverting Fast Radix Algorithm**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Chun Wang

_____ External Examiner
Dr. Robert Sedgewick

_____ External to Program
Dr. Ferhat Khendek

_____ Examiner
Dr. Todd Eavis

_____ Examiner
Dr. Hovhannes Harutyunyan

_____ Thesis Supervisor
Dr. Gregory Butler

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

March 26, 2021 _____
Dr. Mourad Debbabi, Interim Dean
Gina Cody School of Engineering and Computer Science

Abstract

The Diverting Fast Radix Algorithm

Stuart Thiel, Ph.D.
Concordia University, 2021

Radix sort is a classical algorithm to sort N records with integer keys. The keys are represented using digits of a radix. The classical algorithm executes a pass through the records to count the frequency of each digit to determine the digits bucket size, and then a pass to deal the records to the corresponding bucket. The algorithm may proceed from the most significant digit to the least significant, or from the least significant digit to the most significant.

This thesis presents the Fast Radix and the Diverting Fast Radix algorithms, which replace all but one count pass with estimation of bucket sizes and correction of overflow, and may, in addition, divert to another sorting algorithm such as insertion sort when the subproblem size is below a threshold.

Extensive performance experiments compare the implementations of the two algorithms against the native std::sort in the C++ Standard Library, radix sort, and RADULS2, which is currently considered the fastest radix sort available.

We develop a mathematical model of the average-case performance, when keys have a uniform distribution of digits, that support the observed performance improvements. The modeling demonstrates that the costs of estimation and correction quickly approach zero, falling under 5% of the costs of traditional counting for N as low as 11100. The modeling also provides a formula to calculate the thresholds for diversion that conform to the experimental results.

Our focus is on algorithms, so our implementation is fixed-digit, single-threaded and avoids optimizations such as parallelism, cache, translation lookaside buffer (TLB), or the specific input distribution, which are utilised by RADULS2. Nevertheless, Diverting Fast Radix is more than twice as fast as std::sort on inputs of size up to one billion (and beyond), and approximately 10% faster than RADULS2 for one million records (even faster on smaller inputs). It is notable that the compiled code size of Diverting Fast Radix is two orders of magnitude smaller than RADULS2.

Acknowledgments

I want to acknowledge the incredible patience and support of my wife and daughter. My daughter's entire conscious life has been me "finishing" my Ph.D. During this Ph.D., I have had a life-changing diagnosis of severe sleep apnea, and my wife has had a near-fatal stroke and ensuing terrifying related medical issues. My family supported me in my often meandering efforts to complete this thesis.

I want to thank my father, Larry Thiel, who has become an ever-closer friend as we game together, program together and talk about sorting together. I will always have new programming things to learn from him.

I would like to thank the many friends who have contributed to this journey: Albert Chan, Jeremy Upham, Mia Consalvo, Gina Haraszti, Korosh Koochekian Sabor, Meghan White, Nick Gurkov, Jess Marcotte, Susan Upham, Finn Upham, Bruce Alstrom, Patrice Blais and Allison Cole. I am saddened that Terry Fancott and Viivi Lintula are no longer here.

I would like to thank the Department of Computer Science and Software Engineering, Fine Arts Research Facilities, the Milieux Institute, Concordia University, the Natural Science and Engineering Research Council of Canada(NSERC), and the Canada Foundation for Innovation (CFI) for their support in terms of both finance and equipment.

I would like to thank my committee's local members, whom I have known for a very long time: Dr. Eavis, Dr. Khendrk and Dr. Harutyunyan. You have graciously put up with me taking my sweet time and occasionally bursting into your offices with random questions or hijacking you at Christmas parties to talk about math.

I would very much like to thank Dr. Sedgewick for agreeing to be my external. Since very early on in this degree, I knew I wanted to do something that might catch your attention enough that you would agree to be on my committee. Your research has inspired me, and it has helped shape the research I want to do. I am crushed that I will not be able to take you out for beers after the defence, but your presence turns what would otherwise be a flat zoom-pandemic-end of nearly a quarter of my life into something that feels a little more momentous.

I would lastly like to thank Dr. Greg Butler. He has been a true friend, mentor, boss and, of course, supervisor for this thesis. Aside from answering my occasional thesis-related questions, he has freely shared his experiences on university politics and philosophy, ethics, research, approaches to teaching, tactics to manage graduate students, thoughts on government and enterprise involvement in academia, and there have even been a few wine discussions. Greg let this be my journey and helped me in doing so whenever I asked for it. For this, I will forever be grateful.

Contents

List of Figures

List of Tables

List of Algorithms

Notations

N	the length of a file
$R_1, R_2, \ldots, R_N$	the the records in a file
M	the number of possible values for a digit, which is the number of buckets for storing records that share keys with the same digit
k	the average number of records in a bucket, or the index of a summation relating to such values
$A, B, C,$ D, E, F	components of a partial-fraction decomposition

$N!$	factorial $\prod_{k=1}^{N} k$
$N^{\underline{k}}$	falling factorial notation as used in Knuth, $\frac{N!}{(N-k)!}$
$\binom{N}{k}$	binomial coefficient, $\frac{N^{\underline{k}}}{k!}$
$\lceil \frac{N}{M} \rceil$	the smallest $c \in Z$ where $c \geq \frac{N}{M}$
$\lfloor \frac{N}{M} \rfloor$	the largest $c \in Z$ where $c \leq \frac{N}{M}$

Slice Notation

Much of this thesis considers code that performs some algorithm. There are a variety of lists and arrays as both inputs, intermediary structures and outputs. The management of these can obfuscate pseudo code. To alleviate that, I will make use of the slice notation popularized in languages such as Python. This is strictly a means to communicate more cleanly.

$digits[i]$ represents the access to the i^{th} element in an array/list $digits$

$digits[-i]$ represents accessing the i^{th} element from the end, or the $(length(digits) - i)^{th}$ element

$digits[0 : i]$ represents the subset of values from 0 inclusively to i exclusively

$digits[j : i]$ represents the subset of values from j inclusively to i exclusively where $j < i$

$digits[: i]$ represents the subset of values from 0 inclusively to i exclusively

$digits[j :]$ represents the subset of values from j inclusively to the end

$digits[-i :]$ represents the subset of values from $(length(digits) - i)^{th}$ inclusively to the end

$digits[start : end : step]$ represents the slice of values from position $start$ inclusively to end exclusively stepping by $step$ by default the step size is 1.

$digits[:: 2]$ is new structure representing every element in an even position

$digits[1 :: 2]$ is new structure representing every element in an odd position

$digits \setminus some$ uses set difference notation to represent all entries in $digits$ except those in $some$.

Throughout this thesis we assume a little-endian bit-representation for our data.

Throughout this thesis we refer to the least-significant digit as the 0^{th} digit, using a 0-based indexing; given a number with 8 digits, we would refer to the most significant digit as the 7^{th} digit.

Chapter 1

Introduction

"In the early days of computing, the common wisdom was that up to 30 percent of all computing cycles was spent sorting. If that fraction is lower today, one likely reason is that sorting algorithms are relatively efficient, not that sorting has diminished in relative importance." [Sedgewick and Wayne, 2011, p.243]

It is difficult to say what fraction of sorting could be replaced by radix sorts, the focus of this thesis. The current trend in research on that topic suggests that the promise of linearly scaling performance, advances is competitive cache sensitivity and scalabillity across processing cores is drawing increasing attention.

This thesis presents significant average-case improvements on radix sorts given the most commonly used input distribution used in radix sort literature. An exact mathematical model will be presented for the difference in performance between Fast Radix and traditional least-significant digit (LSD) radix sort. We will also develop an exact mathematical model of how the standard technique of diversion reduces passes of the traditional diverting most-significant digit (MSD) radix sorts, and then show that a diverting LSD radix sort is possible. The model of passes performed for the presented LSD radix sort will be proven to be identical to that of diverting MSD radix sorts. Diverting Fast Radix is presented as the combined algorithm. Results of the implementation of these theoretical improvements will be shown to match the presented models. The approaches considered in this thesis are orthogonal to most recent research and focus on stochastic analysis to show that less work can be performed reliably without needing technical improvements to do so. The goal is

that other researchers apply their future improvements to Diverting Fast Radix as a better starting point than other radix sort variants that currently get attention.

Knuth begins the third volume of *The Art of Computer Programming*, *Sorting and Searching* [Knuth, 1998] with an explanation of the ambiguity around the term sorting in and out of Computer Science. He lays the groundwork for an extensive discussion of sorting, and we will use his terminology with respect to sorting N records that each have an accessible key.

> We are given N items
>
> $$R_1, R_2, \ldots, R_N$$
>
> to be sorted; we shall call them *records*, and the entire collection of N records will be called a *file*. Each record R_j has a *key*, K_j, which governs the sorting process. Additional data, besides the key, is usually also present; this extra "satellite information" has no effect on sorting except that it must be carried along as part of each record. Knuth [1998, p. 4]

In this thesis, we outline our contribution to sorting when “keys are known to be randomly distributed with respect to some continuous numerical distribution” [Knuth, 1998, p.5]. Specifically, we will consider the sorting of records whose keys are accessible and whose keys can be broken into a fixed number of *digits*, where all possible values of a digit obey a total ordering and each key contains the same number of digits. We order digits in terms of *significance*, such that the order of two records is only affected by a digit in their keys when all more significant digits are the same. A simple example of a representative file for our purposes would be an array of positive 64 bit integers, with the key being the records in the array, each key also constituting the record and being composed, in this case, of eight digits, each of one byte. While the implementations developed for this thesis sort unsigned integer types, signed integers and floating-point standards exist that are

known to work well with radix sorts. Using stochastic analysis will identify reliable and measurable improvements which will be demonstrated in our empirical results.

1.1 Sorting

Sorting algorithms are pervasive in Computer Science. According to Chen and Reif [1993], servers perform an estimated ~20% of their work on sorting. Modern *Big Data* processing uses sorting in graph processing, bioinformatics and real-time analytics, often benefiting from non-comparison-based approaches to achieve linear sorting performance[Kokot et al., 2018, Polychroniou et al., 2015, Thiel et al., 2016].

People have developed sorting algorithms for a long time, with the application of sorting using simple machines dating back at least a century to Hollerith's sorting and tabulating machines that were used in the US Census Office[Knuth, 1998, p.384]. How algorithms sort varies wildly, as do the algorithms that sort, each having particular characteristics that render them more or less effective given hardware, software or the characteristics of the input to be sorted. Most sorting algorithms either iteratively or recursively perform on inputs to achieve a sorted result, with each iteration or recursion constituting a pass through some or all of the data. Each pass changes the characteristics of the input data in such a way that after the last pass, the result has the same input records, but they are in sorted order. A simple algorithm can be applied repeatedly in this fashion to sort large amounts of data.

More complex algorithms adapt their approach based on the new characteristics of the data dictated or identified by prior passes. This adaptation can involve using another known algorithm to process the current pass or subset of data. Using another algorithm in this context is called *diverting*, a technique used in this thesis and the subject of the next section.

1.1.1 Diverting

A typical example of a diverting opportunity is when a prior pass identifies small, isolated, contiguous subsets of data, as described in Knuth [1998, p. 115] description of Quicksort. When the characteristics of such a subset are that it can be identified and processed more effectively by another strategy, as in the case where the amount of data in it is small and that the position of that subset of data is known relative to all the data before and after it, then diverting is appropriate. A simple algorithm like Insertion Sort can efficiently finish the sorting on such subsets of data.

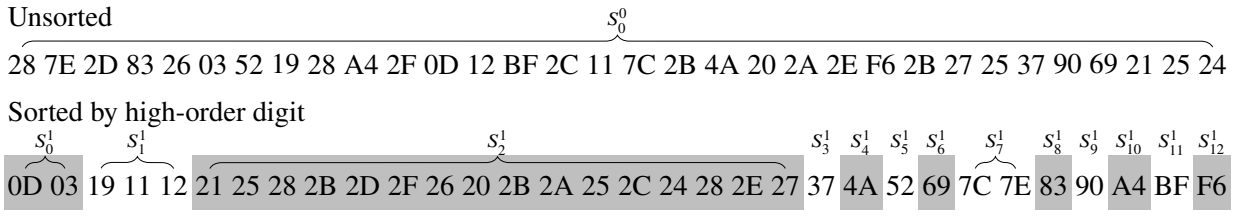

Figure 1: A simple example of showing unsorted data, and then small groups after a first sorting pass.

In Figure 1, we show an unsorted array of two-digit keys and propose to sort them using a two-pass algorithm. In the first pass, values would be ordered by their highest-order digit, as shown in Figure 1. In the second pass, each group of keys with the same high-order digit, represented by alternately shaded backgrounds, would be ordered by their low-order digit. Diversion via Insertion Sort could instead sort those smaller groups. Diversion can be quite flexible; in this example, we identify many small groups, but we would more profitably run Insertion Sort once for each contiguous set of small groups, only running Insertion Sort twice [Knuth, 1998, p. 115].

The choice of diversion can then be thought of as an optimization problem to minimize the cost of sorting an input, the sum of the average amortized operational costs for processing a given group of keys times the number of keys processed. Without going into details on what the analysis would be to yield those costs, imagine that the cost functions were known for this example, either $regular$ or $insort$ for the digit-based ordering cost and the Insertion Sort cost respectively, with each cost

being calculated based on the number of keys being sorted. Each cost would include any setup cost and the cost to perform the sorting-related action on each element given. The cost of the diverting algorithm would also include any cost related to deciding whether to divert.

Let us consider N to be the number of keys being sorted, and let us consider S to be the set of all groups of keys in all passes, and S^j to be the set of groups in a given pass with S_i^j representing the i^{th} group in the j^{th} pass. Lastly, let us consider DT which is the diversion threshold below which we know $insort$ will return a lower cost than $regular$. The overall cost would be:

$$Cost = regular(S_0^0) + \sum_{i}^{|S^1|} \begin{cases} insort(S_i^1), & |S_i^1| <= DT \\ regular(S_i^1), & \text{otherwise} \end{cases} \tag{1.1.1}$$

We know that in general, as larger groups are passed into $regular$, the average cost per key asymptotically approaches c_1 whereas $insort$ asymptotically approaches $c_2 * N$. However, we also know that c_2 will generally be much smaller than c_1. As such, DT is often heuristically or empirically determined to be the value that should minimize the total cost of performing the sort. While Quicksort will not be discussed at any length in this thesis, the canonical example for diversion to Insertion Sort with that algorithm is to choose a DT somewhere between 9 and 15. Knuth provides a detailed analysis for the determination of DT for Quicksort/Insertion Sort written in MIX[Knuth, 1998, p.117-121].

Equation 1.1.1 is just an example to see how cost can be minimized with diversion. The length of a subset is commonly used for diversion, but diversion is not restricted to this approach. There may be more than one diversion algorithm and the cost to determine if one should divert may be more complicated. Anecdotally, more complicated systems cost more and so most examples of diversion keep it simple.

1.1.2 Insertion Sort

Insertion Sort is a simple and effective sorting algorithm in the correct context, and well studied [Knuth, 1998, p.80-105]. While common implementations are usually based off of what Lorin [1975] describes as Sifting, wherein records are swapped into their final place, Straight Insertion Sort, as described by Knuth [1998, p. 80,81], more accurately represents –complete with MIX code– the algorithm we consider wherein the record is copied into place after all the previously in-the-way records are moved over.

If we consider the first eight records from our previous diversion example, shown in Figure 2, we note that every non-empty set of records is a non-empty sorted set of records denoted by S, followed by a potentially empty unsorted set of records denoted by U. When we start our Insertion Sort, we know the first element can be considered a sorted set of records, but we do not know anything about the rest of the records. In each step of the example we identify a cursor, denoted by C underneath the first element in the unsorted list. We will compare this record to each record in the sorted set, from right to left, until we find one that is smaller, and we will denote this insertion point with a < immediately following this smaller record in the sorted set. In the next step, the shaded element is where the previously unsorted element has been placed in the sorted set, which is now one larger. When we are out of unsorted records, we have completed our Insertion Sort. In this example, 23 records are compared (plus two additional boundary comparisons) and 23 records are moved (including moving the record at the cursor out of place, if needed). This is typically fast for small Insertion Sorts.

Modern research focuses on Insertion Sort as a diversion algorithm. As we saw previously, Knuth [1998, p.117-121] gave a direct analysis to determine a size to divert and Sedgewick has discussed delaying the actual running of the diverting algorithm as an optimization, as well as running a variant after the first time to benefit from the creation of a sentinel that allows for fewer boundary checks.

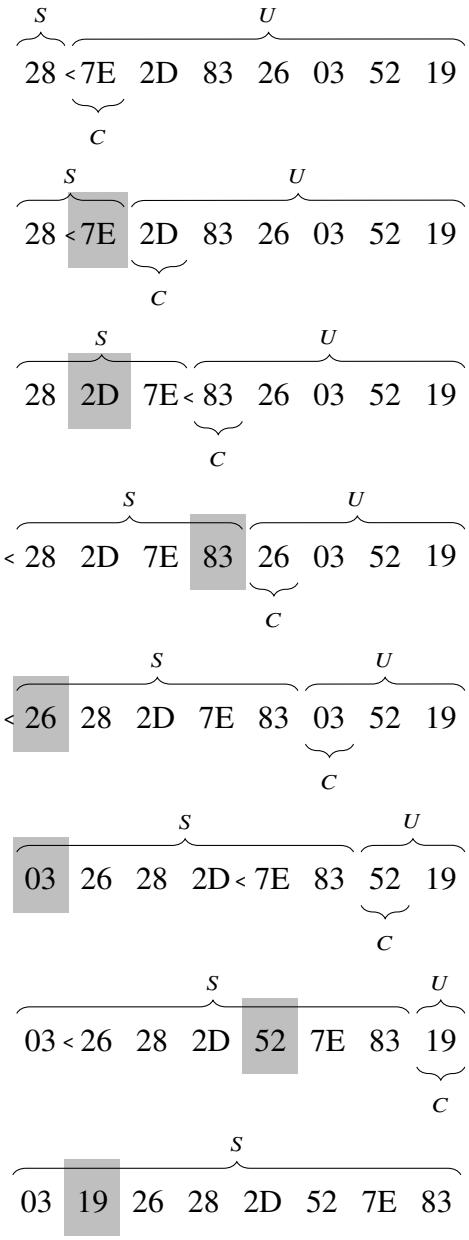

Figure 2: Insertion sorting

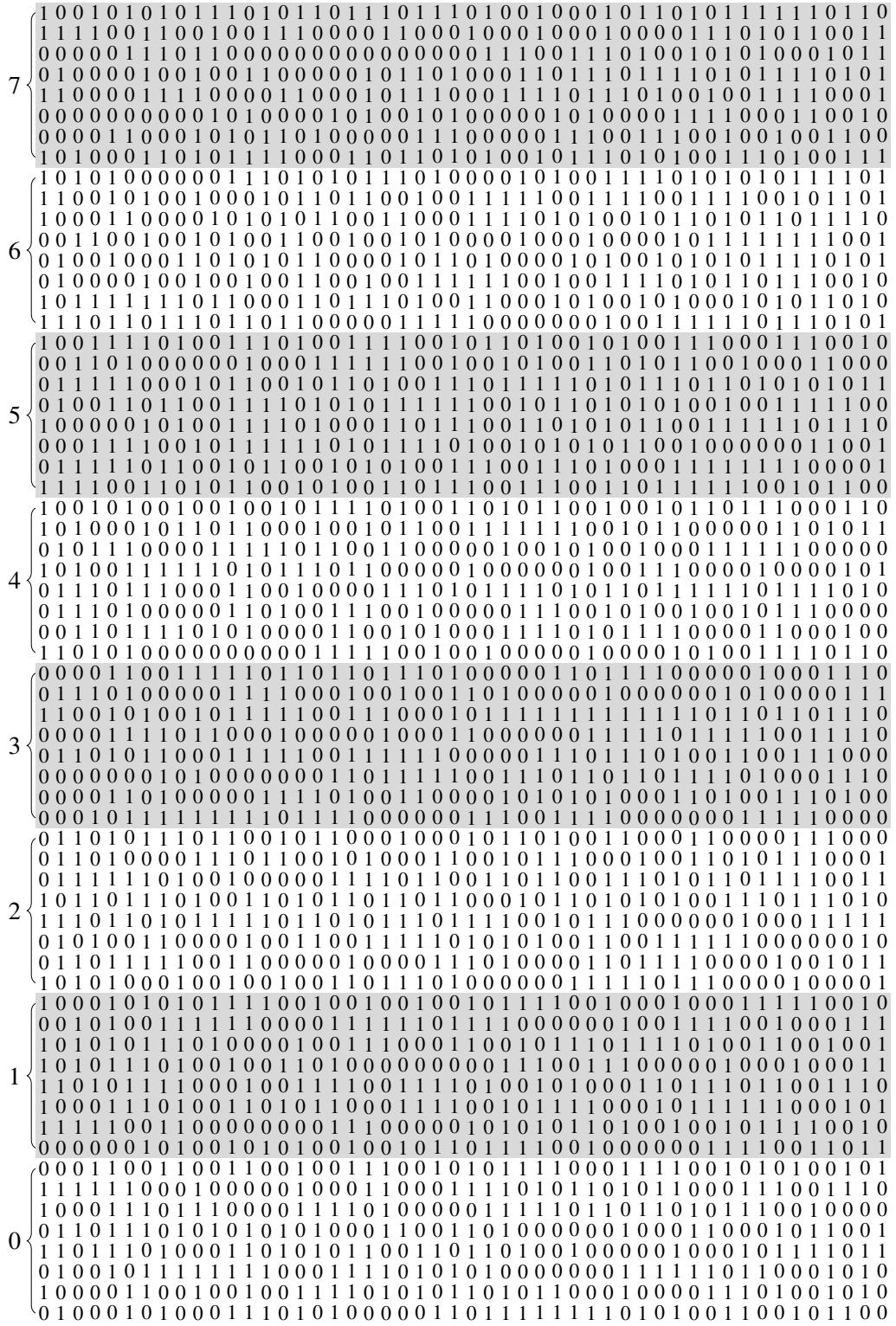

Figure 3: 50 unordered random 64-bit integers, each column representing one integer and each row representing corresponding bits. Most significant bits are at the top. Bits are grouped by byte to form digits, indicated by alternating shading, and each digit is labeled on the left with its index.

Our approach to Insertion Sort leans on this work. While we consider some empirical analysis to decide on diversion thresholds, and we take advantage of the context to inline and re-use small buffers for quicker multiple-writes; this is strictly a practical extension to what is described in the current literature.

1.2 Radix Sorts

A key subset of sorting algorithms are those that sort based on the radix of data instead of by using comparisons. Each element in an input has a key that is considered in terms of a radix to break it into digits. These sorts achieve linear asymptotic complexity in all cases, and are thus generally more favorable when dealing with data that can be broken conveniently into digits.

Four important terms are used here, and they will be used often in this thesis. *pass*, *bucket*, *deal* and *count*. We *pass* through our input. During a *pass* we process each element, and this process is either a *count* or a *deal*, with both the dealing and the counting based on a specific digit of the key. During a single *pass* we only *deal* based on one digit. The *count* gathers information about the occurrence of a digit in the data in order to build *buckets* that we will *deal* into during a subsequent *pass*. We *deal* records into *buckets*, which means somehow moving them into place. Our initial input can be considered a single *bucket*, but after the first *pass* where we *deal*, the input is organized into *buckets*, and we will process the records in subsequent *passes* based on which *buckets* they are in.

A simple form of radix sort is a counting sort, sometimes referred to as a bucket sort, though Knuth does not make the distinction between radix and bucket (or *digital*) sorts [Knuth, 1998, p.169]. In this algorithm, we consider that there is a single digit and we can pass through the input twice, counting the occurrence of each key in the first pass, and dealing into buckets during the second pass. In between the two passes, we convert the count into indices where records should be placed.

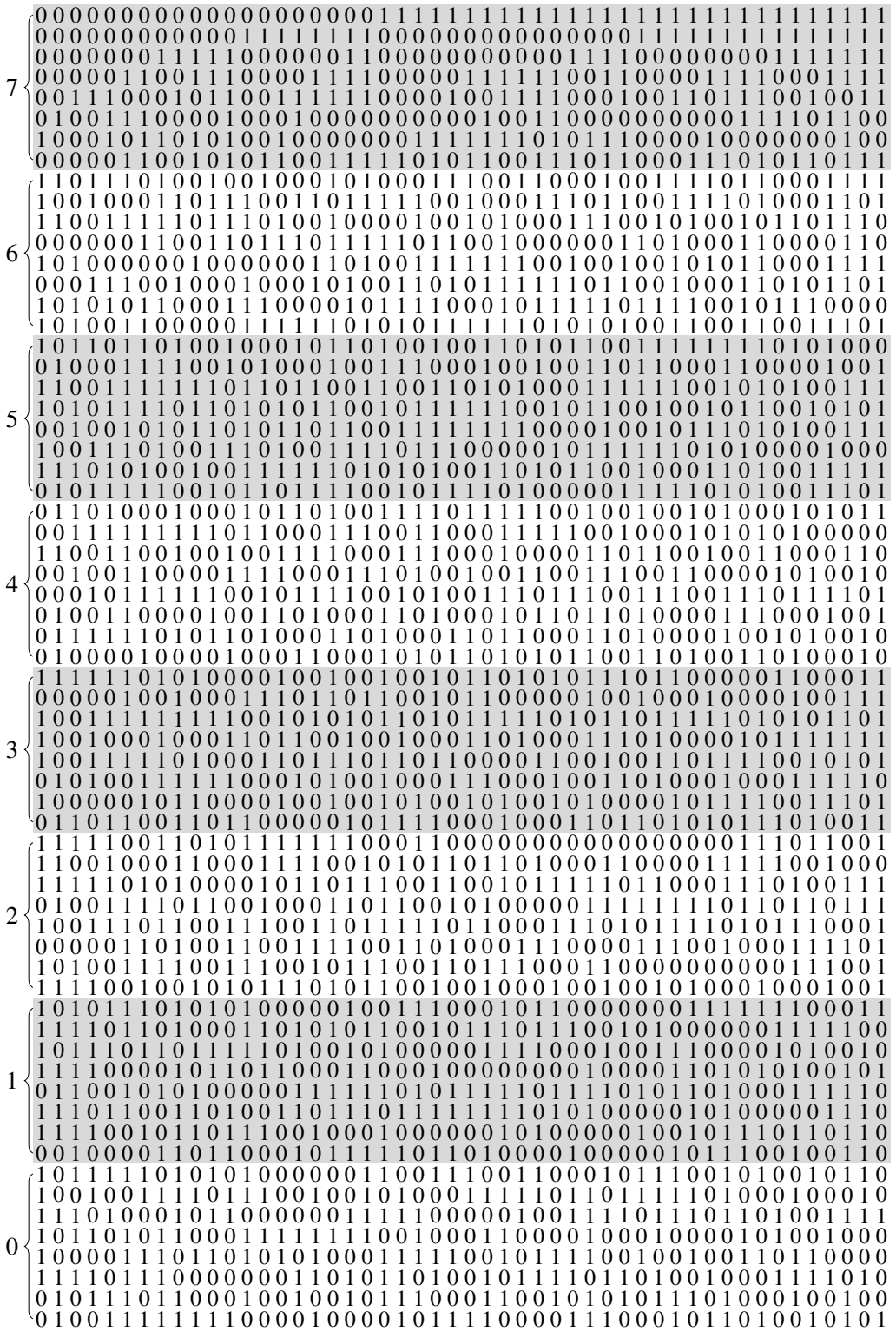

Figure 4: The integers from Figure 3, now ordered. The ordering is visible in the 7^{th} digit, but not elsewhere.

With the indices established, a second pass through the input places everything in order. The areas where records will be placed, where the indices point, are referred to as buckets.

Consider an input whose contents were N records with random keys between 0 and 9. To put this in order one could create an array $count[10]$, initialize its contents to zero, then examine each element in N, increasing the count $count[N[i].key]++$ for each element in N. This set of counts could then be converted into a set of indices where records with the same keys would be directed to the same index. The 0^{th} index would point to enough space to hold all the records that have a 0 for the given digit, the next index would point to enough space to hold all the records that have a 1 and so forth. Passing through the input once more, each record would be placed into the space allocated based on the counts, incrementing the corresponding index so it is pointing to the next free space $buff[index[N[i].key]++] = N$. When complete, the records would be sorted based on their keys, and each index would point to the space immediately following the records that had been placed.

While a simple example, the counting sort above does cover some assumptions that clarify some characteristics with respect to approaches to implementation and application of radix sorts. The characteristics of concern are the digit size of a byte, whether the number of digits is fixed, the use of extra space and the allocation of buckets.

Digit Size of a Byte

Besides the trivial example of the counting sort, in practice the digit size itself is considered fixed, specifically a byte. A byte gives 256 different possible values, which is small enough that processing it is not too costly, and big enough that sufficient data is processed at a time. The size of digit directly affects how many times we pass through the input before achieving a sorted result, so that further encourages the choice of a larger byte, but the cost of processing the counts increases exponentially with the size of the digit. Technical issues around cache also encourage a smaller digit, but in practice there is less wiggle room here than might be expected. Lastly, most computers work very quickly with bytes on all operations one might find during a radix sort. As such, we will use a byte as our digit size, but it is worth understanding that there were options to consider that might become

practical under different computing environments.

Figure 3 shows one-byte digits, with each grouping of eight rows (distinguished by shading) representing corresponding digits from each of 10 64-bit integers shown. In this case, the choice of eight bits, or a byte, creates eight digits for 64-bit integers, thus a radix sort would have eight passes.

A Radix Exchange Sort[Knuth, 1998, p122,123] is a radix sort with an optimization that uses a single bit as the digit allowing that algorithm to skip counting. In the example integers from Figure 3, 64 passes of Radix Exchange Sort would be needed to sort the data. This larger number of passes generally leaves this algorithm as an historical footnote.

Fixed-Digit Radix Sorts

Figure 3 shows 10 64-bit integers with a one-byte digit. Computers regularly specify integer types by their size, either implicitly or explicitly, so sorting a bunch of integers of the same type, as is the common case, yields a fixed number of digits. While radix sorts can be performed on variable digit inputs, this thesis will focus on the the most common usage of radix sorts with fixed digits.

Allocating N Extra Space

The Radix Exchange Sort mentioned previously uses no extra space, but is otherwise of limited practical use. There are in-place radix sorts that also use no extra space, such as Malte Skarupke's Ska Sort[1], that have good performance characteristics. This thesis will focus on the more common radix sorts that allow the use of $\Theta(N)$ extra space and which make up for the extra space usage with faster performance.

While this extra space can be in a variety of forms, either linked-lists or additional contiguous space, depending on the characteristics of inputs, this thesis will focus on the latter, allocating the extra space as arrays. It is worth noting that the usage of linked lists removes the need to count data, but otherwise has poor performance characteristics.

[1]It is well explained, well-optimized, diverts and the author went to the trouble of supporting a variety of non-standard types of input that we know how to support, but we rarely do

Algorithm 1 MSD Radix Sort

Input: input, buf, digits
Output: sorted input on digits

```
1: initialize()
   /* count and deal */
2: c ← d ← digits[−1] // deal/count digit
3: count(input, buf, c)
4: deal(input, buf, d)
   /* recurse for subsequent passes */
5: if length(digits) == 1 then
6:     return
7: end if
8: for j ∈ buckets do
9:     recurse(j, input, digits[: −1])
10: end for
```

Allocating Buckets

Aside from when using linked lists, a radix sort needs to allocate buckets. All implementations reviewed in this research do this by counting values ahead of time to create the buckets that are then dealt into. In this thesis, we will use buckets, but we will demonstrate a more effective approach for their allocation.

When using buckets, records are accessed sequentially within a bucket, which performs more efficiently than accessing values at arbitrary locations in memory in environments with memory hierarchies. While the use of linked lists avoids the need to count, it loses the guarantee of sequential access and incurs additional pointer operations, and thus it is rarely used.

1.2.1 Most Significant Digit (MSD) Radix Sorts

MSD radix sorts process each digit, starting at the most significant digit. For each digit processed, each record will be passed over twice. In the first pass, the digits being considered are counted in order to create buckets of the appropriate size for dealing. In the second pass, the digit being

considered is used to identify the properly sized bucket to deal the record into. After all digits are dealt, each bucket is sorted recursively. After each digit is processed anything in a bucket will be greater than anything in all buckets to its left and less than anything in all buckets to its right. By the time the least significant digit is processed, each bucket for that digit will either be empty or will have one or more records with the same key, and thus is entirely sorted.

A common variant on MSD radix sort is to apply diversion, what we describe as a diverting MSD radix sort. Unless all records are dealt into the same bucket, when each bucket is processed recursively it will be smaller than the original input. Processing smaller buckets is more cache efficient as buckets fit into ever more local levels of cache. Further, if buckets become small enough, diverting to another algorithm can be efficient, with anywhere from 10 to 50% of total time of the sort involving the sorting of these tiny buckets Kokot et al. [2018].

MSD radix sort can be implemented in-place, but will not be stable due to the way records are swapped around. The in-place process swaps records from the free spots in identified buckets, from left to right, into the first free spots in their destination buckets. The next record from the first free spot in the left-most bucket with free spots is then processed. The savings in space come at a potential increase in cost to access records and a guaranteed increase in swaps. Ska Sort, implemented by Skarupke [2017], is an example of a diverting MSD radix sort that makes use of many technical optimizations and ease-of-use measures to demonstrate that this type of sort is still relevant when memory must be conserved. As the performance times of in-place implementations are generally slower than corresponding $\Theta(n)$ space implementations, and we will not consider in-place algorithms any further.

MSD radix sorts can be stable when implemented using an extra buffer of the same size as the input. However, diverting using a non-stable algorithm removes the stability guarantee. Kokot et al. [2018] uses sorting networks to divert in their RADULS2 implementation, and thus give up stability for the performance gains of their diversion algorithm. Using a Merge Sort as a diversion algorithm would be stable, and would take advantage of the already existing buffer space. Using Insertion Sort

Algorithm 2 Diverting MSD Radix Sort

Input: input, buf, digits
Output: sorted input on digits

```
1: if end-start <= DIVERSION then
2:     insertionSort(input)
3:     if isOdd(length(digits)) then
4:         memcopy(input, buf)
5:     end if
6:     return
7: end if
8: initialize()
   /* count and deal */
9: c ← d ← digits[−1] // deal/count digit
10: count(input, buf, c)
11: deal(input, buf, d)
   /* recurse for subsequent passes */
12: if length(digits) == 1 then
13:     return
14: end if
15: for j ∈ buckets do
16:     recurse(j, input, digits[: −1])
17: end for
```

would be stable and would perform very well, but only on the smallest buckets.

Algorithm 1 shows MSD radix sort pseudocode given records in $input$, a suitably sized extra buffer in buf and a set of digits ordered least-to-most significant in $digits$. For the current most-significant digit, the algorithm counts the different values for that digit in the input and then deals into buckets located in buf (lines 3–4). The algorithm is then recursively called on each bucket, using the now available space in $input$ and no longer considering the now counted digit (lines 8–10). The recursion ends all digits have been processed (lines 5–7).

For simplicity sake, the pseudocode in Algorithm 1 hides some details that are less immediately relevant to the general flow of the algorithm. The pseudocode leaves out the creation and management of buckets between the count and deal functions (lines 3–4). The pseudocode also hides that some fiddling with offsets may need to be done when reusing $input$ as the buffer during recursion (line 9). Lastly, the pseudocode of Algorithm 1 also hides the assumption that there are an even number of digits.

In Algorithm 2 we consider a simple diverting MSD radix sort. The algorithm is almost the same as Algorithm 1 but with a check for the input being smaller than the diversion threshold (line 1), in which case it diverts to Insertion Sort (line 2). As we can no longer take for granted an even number of passes, we must track whether an even or odd number of MSD passes have been made.

Any code that calls the MSD radix sort will expect the sorted records to be in the original input when the sorting completes, but during recursion the input and buffer are swapped. If Algorithm 2 diverted after an odd number of passes, the sorted records for that bucket would be in the buffer space instead of in the original input space. This situation is remedied by copying from the buffer holding the sorted records back into the corresponding space in the original input (lines 3–5).

Historically, improvements to MSD radix sorts focus on parallelization, improving cache-locality and effective diversion algorithms, with examples of all these improvements in Kokot et al. [2018], and an example of diversion with an in-place and broadly usable implementation in Skarupke [2017].

Algorithm 3 LSD Radix Sort

Input: input, buf, digits
Output: sorted input on digits

```
1: initialize()
   /* count first digit */
2: c ← digits[0] // counting digit
3: count(input, buf, c)
   /* deal on first digit, and count on second */
4: d ← digits[0] // dealing digit
5: c ← digits[1] // counting digit
6: dealCount(input, buf, d, c)
   /* loop to deal/count-next for each digit save last */
7: for i = 1 to length(digits − 1) do
8:     swapRole(input, buf)
9:     d ← digits[i] // dealing digit
10:    c ← digits[i + 1] // counting digit
11:    dealCount(input, buf, d, c)
12: end for
   /* process last digit, but do not count in deal pass */
13: swapRole(input, buf)
14: deal(input, digits[−1])
```

Parallelization and cache-locality improvements are generally architecture dependent, whereas diversion is more dependent on data distribution.

1.2.2 Least Significant Digit (LSD) Radix Sorts

LSD sorts stably process data from the least significant digit to the most significant digit, performing fewer additional operations per input value, relative to an MSD approach that does not divert.

Historically, improvements to LSD radix sorts focus on improving cache-locality and skipping unneeded passes. Cache-locality improvements are generally architecture dependent, whereas pass-skipping is dependent on data distribution.

We contribute a new improvement to LSD radix sorts which does not rely on architecture and which applies to a broad range of realistic data distributions[Thiel et al., 2016]. We also demonstrate that one can also reliably apply diversion to LSD radix sorts at a trivial cost, demonstrating a further performance boost that was hitherto limited to MSD radix sorts.

1.3 Contributions

The work presented in this thesis will create a new baseline for radix sort algorithm research. The Diverting Fast Radix algorithm presented in this thesis will demonstrate better performance than std::sort on all but the smallest input sizes. Diverting Fast Radix will also be shown to perform competitively against start-of-the-art algorithms in modern literature. Where Diverting Fast Radix does not out perform modern competitors, it will be shown that the cause of the deficiencies is well known and can be overcome by technical optimizations which are well documented, but outside the scope of this thesis.

Previous work in Thiel et al. [2016] will be extended to show that all but one counting pass can be omitted by the Fast Radix algorithm without significant cost. The cost savings from the omission of counting passes via estimation in Fast Radix will be modeled mathematically, in the average case, and a reference implementation of this improvement will be shown to demonstrate the predicted improvement.

Independently of the removal of counting passes, this thesis will demonstrate that diversion can be consistently applied to LSD radix sorts to reduce passes in a manner consistent with how diversion reduces passes in MSD radix sorts. The number of passes performed will be modeled for both diverting MSD and LSD radix sorts. The models of both these algorithms will be shown to be equivalent. Reference implementations of this algorithms will show that they accurately predict actual passes performed. The review of literature in this thesis will highlight that diverting LSD

radix sorts have not previously been seriously explored.

The algorithms will be explained in full, with pseudo-code and reference implementations made available.

1.4 Organization of Thesis

Chapter 2 will begin the discussion of background with a brief history of the terms used in discussing radix sorts. This language will be used to organize a survey of literature on radix sort implementations and their reported improvements. A discussion of trends in how efficiency has been considered over the history of radix sorts will situate these improvements and this work. The background chapter will wrap up with a review of the standard approaches to algorithm analysis applicable in the cases we consider int his thesis.

In Chapter 3, the pseudo-code for both Fast Radix and Diverting Fast Radix will be presented. A general cost model will supplement an explanation of the considerations underlying these algorithms.

In Chapter 4, the performance results of our reference implementations will be shown against optimized std::sort and the current best implementation identified in the literature. A detailed explanation of the techniques applied in the algorithms will given, including both a discussion of chosen approaches and a discussion of some approaches that did not show expected improvement.

In Chapter 5, mathematical models of the presented algorithms will be developed. A model to predict the average cost of overflow from the count estimations in Fast Radix will show that the overflow cost as a percentage of the input size goes to zero quickly as the input size increases. The model of average passes performed of the standard diverting MSD radix sort will be developed. The model of the diverting LSD radix sort used by Diverting Fast Radix will also be developed and shown to be equivalent to that of the MSD radix sort.

The thesis will conclude with a reiteration of contributions and a consideration of future work.

Chapter 2

Background

2.1 Radix Sort

Exactly how Radix Sort came to be is unclear. Knuth cite's Hollerith's sorting and tabulating machines that were used in the US Census Office of the late 1800s and into the early 1900s[Knuth, 1998, p.384] as a solid early example of sorting, but Cormen et al. summarizes it well with the following quote:

"Knuth credits H.H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of mechanical card-sorting machines." [Cormen et al., 2009, p.211]

Its origins as a folk algorithm, often referenced to the 1880s in terms of mechanical use, the fact that there are both most and least significant flavors of the algorithm and even changes in use of language over time lead to some confusion. In Friend [1956], care is taken to distinguish the term *digit* used in the "Internal Digital Sorting" as separate from the *radix*, which need not be 10-based, but they note that it is "...important to recognize that the positions in the control field must be considered in order from "least significant" to "most significant" with each one requring[sic] a separate pass"[Friend, 1956, p.18]. In Rahman and Raman [2001] the standard 8-bit radix is used and the terms least/most-significant-bit (LSB/MSB) first radix sort are considered, which appears to

be easily interpreted as least or most-significant-byte, so that form occurs regularly. After 2000, least/most-significant-digit (LSD/MSD) radix sort, the terms we use in this paper, becomes more common in spite of Friend's careful admonishment on the use of language; in particular Cormen et al., p.211's analysis of the linear time of LSD radix sort (though they do not call it exactly that) refers to "d-digit numbers in which each digit can take on up to k possible values."

There is occasional conflation with counting sort, though when only one pass is involved, they *are* the same. Bucket sort is occasionally conflated with MSD radix sort. In some works, for example Lee et al. [2002], one sees the terms "left-to-right" or "right-to-left" sort that appear to be based on a big-endian interpretation of the actual encoding. Maus [2002] refers to MSD radix sort as also being "top down radix sort", with the "left radix" (again, big-endian) variety being a "bottom up radix sort." Lee et al. [2002] and Al-Badarneh and El-Aker [2004] both refer to LSD radix sort as a "straight radix sort". Lorin [1975] and Andersson and Nilsson [1994] use the term "lexographic sort," though this is implied as being for variable key sized inputs (e.g. variable length strings), and it is generally of the MSD variety, as Andersson and Nilsson [1994] also refer to it as "forward radix sort" and note that it is often best to divert to comparison-based-sorts when buckets become small.

The terminology used for the components of a radix sort is more consistent, with most variance being early on or based on the appearance of terms when they are considered important, and their absence otherwise. *Pass* is used consistently going as far back as Friend [1956]. Friend [1956] refers to the "recipient area" for a particular position, but this is synonymous to what is called the *buffer* in most latter papers; they do note that the "sending area" (what is now referred to as *input*, in general) and the "recipient area" alternate with each pass. Cormen et al. [2009] refers to bins. The majority of other papers refer to the specific target area for elements with like digits as *buckets*.

Counting, or frequency counting is quite consistent. Where terminology around counting varies is in consideration of performance impacts of those counts, so papers that concern themselves with these issues include mention of more details related to the counting, such as the initialization of the counts and the "prefix sum" of the counts[Rahman and Raman, 2001], and not just the count itself,

Topic	Source
Count	Friend [1956], Rahman and Raman [2001], Thiel et al. [2016], Thiel [2019], Kumar [2019], Hanel et al. [2020]
Deal	**LSD:** LaMarca and Ladner [1999], Rahman and Raman [2001], Satish et al. [2009], **MSD:** Jiménez-González et al. [2003], Kokot et al. [2017], Kokot et al. [2018], Hanel et al. [2020]
Diversion	**LSD:** Sedgewick [1998], Al-Badarneh and El-Aker [2004], Thiel [2019] **MSD:** Maus [2019], Maus [2002], Jiménez-González et al. [2003], Kokot et al. [2017], Kokot et al. [2018]
Parallelization	Lee et al. [2002], Satish et al. [2009], Delorme et al. [2013]
Variable Radix Size	LaMarca and Ladner [1999], Maus [2002]
Variable Key Size	Paige and Tarjan [1987], Mcllroy et al. [1993], Andersson and Nilsson [1994], Bentley and Sedgewick [1997], Nilsson [2000], Kärkkäinen and Rantala [2009]

Table 1: This table identifies literature where improvements have been made or discussed, organized by parts of the radix sort where improvements have been applied and specific areas of research that affect the radix sort as a whole. Whether the improvement was specific to LSD or MSD radix sorts was specified for clarity.

which is their primary focus.

The term for *dealing* varies significantly across papers. Friend [1956, p.18] uses the term disbursement. [Rahman and Raman, 2001] calls this a permutation phase, what appears to be a nod to in-place MSD radix sort's unstable approach to dealing[1]. There are also some papers that refer to the dealing phase as the distribution phase, as radix sorts fall into the class of distribution sorts; Cormen et al. [2009] describes how mechanical card-sorting machines would "... examine a given column of each card in a deck and distribute the card..."[Cormen et al., 2009, p.197] though they do not use the term *deal* that one might readily associate with *deck* and *card*. It is difficult to say if the term *deal* is the most common term in the literature, but it is in the running.

Table 1 outlines key changes in radix sort over the last several decades to situate this thesis. The most significant improvements outside of this work fall into improvements in the performance of dealing, primarily through improving cache and Translation Lookaside Buffer (TLB) performance

[1]While some papers consider this approach, which is not applicable in this thesis, the well-developed non-academicly published Ska Sort[Skarupke, 2017] gives it an impressive treatment, and it should not be overlooked.

and work on applying diversion to radix sorts, primarily to MSD radix sorts. In addition, there is significant work on parallelization which, while falling outside of the scope of this thesis, identifies important research in the area and highlights opportunities for future work on this thesis material. There are several instances of the use of variable radix size in the literature, and we note some of them here as, while they often lack an immediate performance gain, their regular recurrence in the literature suggests that this approach at least has some niche application, particularly in conjunction with other techniques.

In this section we will introduce the background for key concepts underpinning this thesis. A brief discussion of efficiency will introduce key supporting concepts that will give reason for different approaches used in this work and related work, which otherwise may seem arbitrary. We will then introduce sorting, the concept of diverting (as used in sorting), and the classes of sorting algorithms considered in this thesis, introducing both historical and contemporary work in those areas. We will round out this section by considering relevant analysis techniques in sorting algorithms, including analysis techniques that are more commonly used elsewhere that become relevant for the stochastic approaches introduced in this thesis.

2.2 Efficiency

Efficiency is often used to describe the desirable metric around some process. Often we refer to things as space-efficient or time-efficient. Energy-efficiency is also mentioned to mitigate that simple time-efficiency really just means more energy is being expended to do something mostly in parallel. Compute-time-efficiency, by comparison, is about the time a processor spends doing something and thus is not affected by parallelization and is a better counter-point to energy-efficiency. In this thesis, we will focus on compute-time efficiency when efficiency is discussed, though the general goals of this thesis are to do fewer things than we used to, not just to do the same things faster or more energy efficiently.

Often we see reference to the number of records that can be sorted in a given amount of time or with a given amount of energy. If an algorithm can sort more records in the same amount of time or energy, then that algorithm is more efficient in that respect. If an algorithm sorts more records in a given amount of time and energy, but takes more space, is it more efficient? All things scaling in the same way, it is really about limiting factors in the environment where the sorting is taking placel usually that is tied to money.

These days, space is the least relevant limiting factor. In practical terms, we will consider sorting things that can be held entirely in memory, though, particularly with the algorithms we will consider, this is not so relevant a restriction as it once was, both because memory is cheap and plentiful, and because cache-efficient writing to disk does not have as significant an impact is it might in other times or with different algorithms.

Compute-time and energy are where money comes in. Considering parallel computing, a business is not concerned with how fast they can sort a single large input, they are concerned with sorting many large inputs all the time. In that sense, it is more efficient to sort 10 inputs on 10 processors by having each processor sort one input instead of having each input be sorted in parallel by 10 processors. That is not to say that there is no application to being able to parallelize an algorithm, only that this is just a special case worth noting, not the general target.

Where compute-time and energy are not exactly the same are when individual operations performed on a single processor have differing costs and can be interchanged. This can either relate to low-level parallelism or actual differences in performance. On a specific hardware, does the energy cost of accessing a certain level of cache scale linearly with the time cost? Most of the accessible research in this area deals with access to cache and the varying flavors of cache misses that can impact compute-time and energy, specifically there is a lot of research around cache-sensitive algorithms and minimizing TLB misses. Discussion of low-level parallelism is generally relegated to low-level programming heuristics, but there is some research on surveys of algorithms and variants as well as results for specific implementations of pieces of algorithms that are practical to consider.

In practical terms, this generally translates to taking care to access data in a cache-sensitive fashion, for both reading and writing. It also impacts balancing the cost of recording and processing meta-information such as counts (or even sampling data) against other approaches to achieve the algorithms goals.

An example of the issue of cache-sensitivity was an implementation of a sorting algorithm that processed pointers that were contiguous in memory, so accessing these pointers was cache friendly, but it then looked up information at a position in memory unrelated to the relative position of the pointer. Since the pointers were initially just the relative positions of the records in memory, the first pass was cache friendly and efficient when interacting with the underlying data. When the amount of data was not very large, it was somewhat slower, but not too much so. However, when many millions or billions of records were involved, as the pointers were shuffled around during the sorting process, subsequent access to the underlying data appeared to lead to cache misses on virtually ever access, with the possibility that main memory was accessed most of them, taking orders of magnitude more time for those operations than had been taken in the initial pass. We had no immediate measure of the energy efficiency in this example, but the compute-time efficiency was clearly impacted.

In this thesis, we focus on compute-time efficiency, and indirectly energy-efficiency, though this is not our direct goal. The primary goal is to just do less, but we consider compute-time-efficiency when weighing the cost of doing less, and use this consideration to balance against making the algorithms we propose small, simple and easy to use. In general, the approaches we use for our implementation are relatively standard practices and are not contentious, and more importantly we stress that the benefits of our algorithm are still present when contemporary alternatives to implementation details are used.

2.3 Analysis of Sorting Algorithms

2.3.1 Concepts

Certain concepts from the analysis of algorithms covered in this thesis come up often.

Occupancy Distribution

Given the standard ball-and-urn model where $N, M \in Z_{\geq 1}$, we can use Sedgewick and Flajolet's description:

> The average number of urns with k balls, when N balls are randomly distributed in M urns, is:
>
> $$M\binom{N}{k}(\frac{1}{M})^k(1-\frac{1}{M})^{N-k}$$
>
> [Sedgewick and Flajolet, 2013, p. 503, Theorem 9.5]

Further, Sedgewick and Flajolet's specify the average number of balls per urn is $\frac{N}{M}$ and the standard deviation is $\sqrt{\frac{N}{M}-\frac{N}{M^2}}$ [Sedgewick and Flajolet, 2013, p. 503, Second Corolooary].

So the sum of all average possible values for k must therefore be M:

$$M = \sum_{k=0}^{N} M\binom{N}{k}(\frac{1}{M})^k(1-\frac{1}{M})^{N-k} \tag{2.3.1}$$

2.3.2 Transformations

Throughout this thesis, many results are mathematically established. The equations, in context, can appear complicated. In order to simplify, we note here the transformations that we rely on, and we will refer back to the transformations identified here to clarify what we are doing later.

Pascal's Rule or Addition/Induction

Given the nature of binomial coefficients, and for $N, k \in \mathbb{R}_{>0}$, we can always decompose a binomial coefficient into a sum of two other binomial coefficients:

$$\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1} \tag{2.3.2}$$

[Graham et al., 1994, p.174, Table 174, 4^{th} identity]

This is useful when we need to increase or decrease the number of terms. For the analysis in this thesis, when intuition suggests a solution of a certain form, but what we have does not seem to fit, this is one of the first things we attempt.

Trinomial Revision

$$\binom{N}{j}\binom{j}{k} = \binom{N}{k}\binom{N-k}{j-k} \tag{2.3.3}$$

[Graham et al., 1994, p.174, Table 174, 6^{th} identity]

Equation 2.3.3 is the number of ways to pick the j men from the N mortals times the number of ways to pick the k Socrates from those j men is the same as the number of ways to pick the same k Socrates from those N mortals times the number of ways to pick the $j-k$ men who are not Socrates

from the $N-k$ mortals who are not Socrates[2]. It is a way of getting a $\binom{N}{k}$ factor when you have something in the form of $\binom{N}{j}\binom{j}{k}$.

Partial-Fraction Decomposition

Sometimes we have sums of fractional terms that are not of the form that we want. Often we have too many, or too few terms; more significantly, the denominators or numerators may not be in an obviously useful or familiar form. When we need an extra term and we know what denominators we will need, then we have guidance to attempt such a transformation. When we have too many terms, this can be trickier, but if we know the denominator we want, then at least we can look for two terms with denominators that have the desired denominator as their product. The numerators are often less obvious, but in walking through the process of decomposing or recomposing partial fractions things often come together.

$$\frac{E}{F} = \frac{A}{C} + \frac{B}{D} \tag{2.3.4}$$

where

$$E = AD + BC \tag{2.3.5}$$

and

$$F = CD \tag{2.3.6}$$

Graham et al. refer to this more generally as *partial fraction expansion*[Graham et al., 1994, p.298-299] (in the other direction, of course) and show a "slick trick"[Graham et al., 1994, p.299] to solve for $\frac{E}{F}$, even when terms are not simple.

[2]My mnemonic for this is *Binomial Syllogism*

The Ceiling of Rational Numbers

In this thesis, we will often refer to the ceiling of a very specific fraction, $\lceil \frac{N}{M} \rceil$, which we will discuss later. For our purposes, positive integer N is of interest. Given that, we can consider N in two parts, some multiple of M and some integer between 1 and M. We represent this as $N = cM + a$ for $c \in \mathbb{Z}_{\geq 0}, N \in \mathbb{Z}_{>0}, M \in \mathbb{Z}_{>0}, a \in \mathbb{Z} : 1 \leq a \leq M$. Of further interest is the relationship between $\frac{N}{M}$ and $\lceil \frac{N}{M} \rceil$:

$$\frac{N}{M} = \qquad \frac{a+cM}{M} = \qquad c + \frac{a}{M} = \qquad c + 1 - \frac{M-a}{M} \tag{2.3.7}$$

$$\lceil \frac{N}{M} \rceil = \qquad \lceil \frac{a+cM}{M} \rceil = \qquad c + \lceil \frac{a}{M} \rceil = \qquad c + 1 \tag{2.3.8}$$

This leads to the fact that $\frac{N}{M}$ is just $\lceil \frac{N}{M} \rceil$ taking away $\frac{M-a}{M}$ parts that were rounded up by in the process of performing the ceiling operation. When $a = M$, N is $cM + M$, therefore $\frac{N}{M} = \lceil \frac{N}{M} \rceil = c + 1$

We can expand further:

$$\frac{N}{M} = \lceil \frac{N}{M} \rceil - \frac{M-a}{M} \tag{2.3.9}$$

$$\frac{N}{M} = (\frac{a}{M} + \frac{M-a}{M}) \lceil \frac{N}{M} \rceil - \frac{M-a}{M} \tag{2.3.10}$$

$$\frac{N}{M} = \frac{a}{M} \lceil \frac{N}{M} \rceil + \frac{M-a}{M} (\lceil \frac{N}{M} \rceil - 1) \tag{2.3.11}$$

Equation 2.3.11 splits $\frac{N}{M}$ in terms of $\frac{M-a}{M}$ and its complement with respect to $\lceil \frac{N}{M} \rceil$, which provides a practical representation that we will use in analyzing Fast Radix later.

2.3.3 Classification of Input

While sorting algorithms themselves cover the operations involved in going about sorting, the sorting itself is applied to some input. Understanding of that input and what it means is potentially quite complex. However, such considerations can yield improved performance, and more specifically, a better understanding of what operations are appropriate – and when – for potential algorithms.

To tie it all together in a simple way that will hopefully make clear why it is useful in this research, we present the following: An input for a sorting algorithm is a List L containing n elements E_i, where i is from 0 to n. All elements E_i are drawn from the Set S. L is a permutation of the ordered List L_{ord}. In these terms, we identify "Types of Input" as per Stevens' Scales, and suggest that this applies a classification of what relations are appropriate to apply to any List of elements drawn from S within the context of L. Similarly, we identify "Distributions" as a classification describing how elements E_i can be drawn from S, indicating how many will be drawn, and the probability that any given element will be drawn, but not at all pertaining to the order in which the List L is filled; treating S as continuous, then this applies equally well. Lastly, we identify "disorder" as a classification of the specific permutation of elements in L.

The likelihood of presortedness is not explicitly identified by these classifications, but as noted by Estivill-Castro, "nearly sorted sequences are common in practice" Estivill-Castro and Wood [1992]. Thus a fourth indirect classification appears to be "Presortedness" of several input Lists, which is to say an expected measure of disorder that is consistent over a set of actual or expected practical inputs. In most cases, authors appear to imply this by identifying the specific measure of disorder and a numeric range of expected values (or a loose textual qualification) while further qualifying the source of the sets with natural language e.g. "... ordered by reversed spelling, which mixes the data well." Mcllroy et al. [1993], which identifies a high value of disorder for some implied yet ambiguous measure of disorder, qualified by natural language indicating that this set of Lists is coming from the reverse ordering of dictionaries. However, when the term "presortedness"

Scale	Basic Empirical Operations	Mathematical Group Structure	Permissible Statistics (invariantive)
NOMINAL	Determination of equality	*Permutation group* $x' = f(x)$ $f(x)$ means any one-to-one substitution	Number of cases Mode Contingency correlation
ORDINAL	Determination of greater or less	*Isotonic group* $x' = f(x)$ $f(x)$ means any monotonic increasing function	Median Percentiles
INTERVAL	Determination of equality of intervals or differences	*General linear group* $x' = ax + b$	Mean Standard deviation Rank-order correlation Product-moment correlation
RATIO	Determination of equality of ratios	*Similarity group* $x' = ax$	Coefficient of variation

Figure 5: Stevens Scales of Measurement taken directly from "On the Theory of Scales of Measurement" Stevens [1946]

Class	example	allows	Worst case Big O
nominal	shapes	=	$\Theta(n^2)$
ordinal	favorite foods	$\leq$	$\Theta(n \log n)$
interval	date	-	$\Theta(n)$

Table 2: How Stevens' Scales of Measurement impacts the complexity of sorting. Sorting of nominal things would just be grouping like things.

is applied to a specific List, it appears that it is always meant as a measure of disorder.

Types of Input

Around the 40s Stevens Stevens [1939, 1946] proposed a scale of measurement based around the idea that such scales are the assignment of numerals to things by a set of rules, and that these rules defined appropriate interaction on the things being measured. Our interest in this scale lies in its application to the input needing sorting and whether the specification of scale has implications on the type of sorting that is optimal to use.

The nominal scale is of little interest for this purpose, as it is concerned only with identifying whether two elements are the same, but offers no insight on any rank relation. On the other hand, the ratio scale may be more than is immediately useful when analyzing these inputs, at least given the types of algorithms and inputs being considered thus far.

That leaves ordinal and interval scales to consider. The former supports identifying equality or rank between elements, exactly what traditional comparison sorting does, and the latter supports those tests, as well as identifying equality or difference of intervals between elements. This additional property can allow the encoding to a radix-based system that one can see corresponds to radix sorting.

While this sounds trivial, it offers terminology to distinguish between two known input types, as well as introducing Stevens' cautions with regards to their applicability. To paraphrase, you can number ordinal things and apply statistics, pretending that the intervals are the same, and maybe even get good results, but that does not mean that this is a good idea and that you will always get good results. Conversely, just because one has inputs that appear numeric or to have distance does not guarantee that the intervals are relevant and that the encoding allows for anything more useful than tests of equality or rank ordering.

Distributions

Input distributions are well known in terms of sorting algorithm input, particularly uniform distributions and normal distributions. Binomial distributions and Cauchy distributions are brought up by Chakraborty, Singh and Sourabh in their various papers, but they appear not to be generally considered in sorting algorithms [Chakraborty et al., 2007, Singh, 2012, Singh and Chakraborty, 2012a,b, 2011, Sourabh and Chakraborty, 2008].

Additional ambiguity in distributions stems from the fact that distributions have different meaning when applied to ordinal scales than when applied to interval scales. With ordinal scales, you have

the occurrence of duplicate keys discussed in detail by Sedgewick in "Quicksort With Equal Keys" Sedgewick [1977], where multisets feature prominently.

When dealing with interval scales and radix sorting, distributions affect how many things will get assigned to buckets, which can directly affect the structure of an algorithm, or can have just as significant an impact based on properties of the underlying computational model/memory model that it is implemented on. Even under cases of comparison-based sorts on interval data, such as with meansort, a distribution based on the frequency of duplication of keys would have a different performance impact than a continuous distribution around a certain integer value, though both might be described as normal distributions.

Fortunately, such ambiguity appears as rare as the overlap between such approaches. When dealing with comparison based-sorts, distribution usually implies some sort of histogram of frequency of occurrence of equal keys, and when dealing with radix sort distributions are continuous, or at least are impacted by proximity vs. equality.

Measures of Disorder and Presortedness

Several measures of disorder are outlined in Estivill-Castro's "A Survey of Adaptive Sorting Algorithms" [Estivill-Castro and Wood, 1992]. (Inv) Inversions is the measure of how frequently a given element occurs before an element that should precede it, summing these counts for each element. Thus, an ordered list has no inversions and a reverse-ordered list would have the maximum inversion count for its length and distribution. (Dis) Distance is a measure of the degree of separation between inversions. (Max) Max is a measure of the maximal distance between elements and their correct sorted position. (Exc) Exchanges is a count of the minimal number of exchanges (or other unit-operations) required to put a sequence back in order. (Rem) Removed measures the minimum number of elements needed to be removed in order to be left with an ordered sub-sequence. (Runs) Runs measure the number of ordered runs (vs. reverse-ordered runs) in a sequence, as measured by

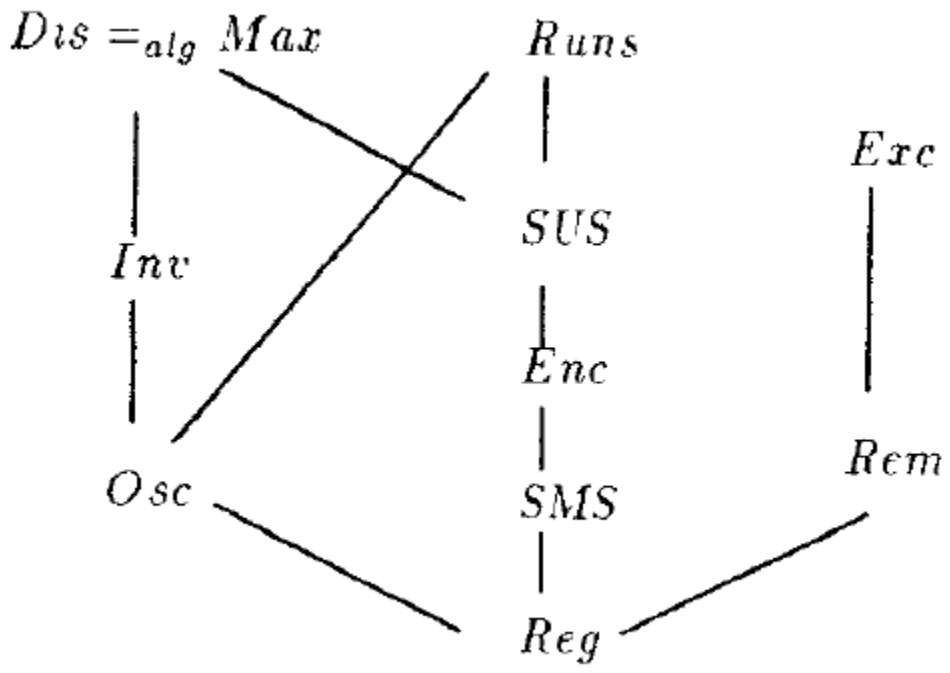

Figure 6: Estivill-Castro's partial ordering of measures of disorder [Estivill-Castro and Wood, 1992].

the boundary between such runs. (SUS) Shuffled Up-Sequences measures the minimum number of ascending sub-sequences that an input can be partitioned into, the "shuffled" suggesting the interleaved nature of these sequences in the original input. (SMS) Shuffled Monotone Subsequences is similar to SUS, but considered both ascending and descending order for valid runs. (Enc) Encroaching Lists measures the number of such lists that can be generated by melsort [Skiena, 1988], and is identified as one of a variety of such measures that have been specifically designed to accommodate the identifications of good permutations for specific algorithms or research. (Osc) measures the oscillations between large and small elements in heapsort. (Reg) Regional is a measure proposed by Petersson and Moffat which is designed to cover all the previously mentioned measures [Petersson and Moffat, 1995]. [Estivill-Castro and Wood, 1992]

Petersson and Moffat identify a partial ordering of these measures and Estivill-Castro appear to adjust this slightly [Estivill-Castro and Wood, 1992, Petersson and Moffat, 1995]; Estivill-Castro's partial ordering can be seen in Figure 6. It is unclear how Estivill-Castro adjusted the partial

ordering, but Moffat introduces it as follows:

> We introduce a natural relation, denoted $\supseteq$, which defines a partial order on the set of measures. If M_1 and M_2 are two measures of presortedness, and $M_1 \supseteq M_2$, every sorting algorithm that optimally adapts to M_1 optimally adapts to M_2 as well. [Petersson and Moffat, 1995]

In either case, these measures are used to define an algorithm as adaptive, that is an algorithm which sorts faster when some measure of disorder is small [Estivill-Castro and Wood, 1992, Mannila, 1985]. Such adaptive algorithms are considered useful to study as "...nearly sorted sequences are common in practice, and adaptive sorting algorithms are more efficient [Knuth 1973, p. 339; Mehlhorn 1984, p. 54]." [Estivill-Castro and Wood, 1992, as cited by]. The approaches and considerations identified here can guide the creation of new adaptive algorithm, or offer insight and alternatives approaches that might add adaptive qualities to a new or existing algorithm at little or no cost.

While empirical analysis of sorting algorithms is most common in recent literature, one can also mathematically model algorithms, or the differences between algorithms to more exactly understand them. A comparison of the models of different classes of algorithms can demonstrate the differences in their asymptotic analysis, e.g. comparing Quicksort to a radix sort. One can also compare the models of very similar algorithms, or their differences, to model costs and benefits.

One of the more commonly considered models for visualizing many classes of algorithm analysis are the Ball and Urn Models[Berg, 2014]. Stochastic analysis lends itself to such models, and thus one can more readily show comparisons between algorithms and approaches in varying fields by using these familiar models. Further, an extensive body of research exists showing how the study of characteristics of Ball and Urn Models apply to a variety of problems, both within Computer Science and elsewhere.

In this thesis, we focus on the variant of the Ball and Urn Models where there is a fixed number M urns and N balls randomly distributed among them, evaluating how many urns contain balls beyond a fixed threshold τ and what percentage of balls overflow on average as N increases. This is a specific configuration of the Occupancy Problem. Considering a brief history of Ball and Urn Models, the Occupancy Problem and their related problems will situate our contribution within the literature.

The Occupancy Problem

The general occupancy problem is set up as N balls placed randomly into M urns with the probability p_f that the ball will remain (instead of falling through) and the probability p_m that the next ball will end up in any urn m. The problem is to determine some characteristic of such a system, or how that characteristic changes as one of the parameters changes. If we consider the special case where $p_f = 1$, M is fixed and all values of p_m are the same, then we can model various characteristics relevant to this research.

Williamson et al. have examined a random variable X, effectively describing the average number of urns that will contain at least one ball after N balls have been placed [Williamson et al., 2009]. This is the same form of the Occupancy Problem described above, and a simple solution can be shown based on the occupancy function $u(k, N, M)$, "[t]he average number of urns with k balls, when N balls are randomly distributed in M urns"[Sedgewick and Flajolet, 2013, p.503].

$$u(k, N, M) = M\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \tag{2.3.12}$$

This is derived from the probability function $p(k, N, P)$, that "an event will occur exactly k times out of $[N]$"[Freedman et al., 2007, p.259] where "[P] is the probability that the event will occur on

any particular trial":

$$p(k, N, P) = \binom{N}{k} P^k (1 - P)^{N-k} \tag{2.3.13}$$

Specifically, we consider the case where the possible outcomes are equally probable, i.e. $P = \frac{1}{M}$, as an event k represents a ball being placed in an urn and we have M equally probably urns for each ball to be placed into. In that case, we have $u(k, N, M) = p(k, N, \frac{1}{M})M$

The variable X described by Williamson et al., which we denote as the function $X(N, M)$, is simply $M - u(0, N, M)$:

$$X(N, M) = M(1 - (\frac{M - 1}{M})^N) \tag{2.3.14}$$

Chapter 3

Fast Radix Algorithms

In previous work, Thiel et al. [2016], we have demonstrated the improvements of Simple Fast Radix (referred to as "Fast Radix" in that work) over radix sorts, showing that omitting the first counting pass leads to faster performance. This chapter will discuss the Simple Fast Radix algorithm, our improved Fast Radix algorithm, our diverting LSD radix sort algorithm and our Diverting Fast Radix algorithm. The chapter will complete with a summary of the discussed algorithms.

Each section will provide a brief introduction to the corresponding algorithm that highlights the introduced improvements. The algorithm's details will then be discussed, describing variables, structures and functions used in the pseudocode. The section will then present high-level pseudocode for its respective algorithm and then describe in natural language that references the pseudocode. An example that refers to the pseudocode will be presented, including additional lower-level pseudocode as warranted. Each section will finish with a proof of correctness and discussion of the areas, often glossed over by the pseudocode, where there are opportunities for future work. Exceptionally, Diverting Fast Radix will only include pseudocode and a brief discussion of how it is just a direct combination of Fast Radix and diverting LSD Radix Sort.

The Simple Fast Radix section will highlight how that algorithm compares to the LSD radix sort algorithm with a review of the LSD radix sort pseudocode from Chapter 1. Similarly, we will briefly revisit the diverting MSD radix sort pseudocode from Chapter 1 when comparing diverting MSD radix sort and diverting LSD radix sort.

Throughout this chapter, there are commonalities in how we refer to the records passed to the sorting algorithms and the notations we use. The input to radix sorts is an array of N records to be sorted identified as `input` and an array of the digits on which to perform the radix sort identified as `digits`. We omit the case of a single digit, and in general, the case of an odd number of digits. With a one-byte digit (i.e. where M=256) and a key of 8 bytes, `digits` would contain the values $[0, 1, 2, 3, 4, 5, 6, 7]$, with 0 being the least significant digit. Specific records in `input` are accessed via array notation, e.g. if `input` contained $[65539, 27]$ we would access the 0^{th} record, 65539, using the notation `input[0]`. The algorithms similarly will access the digits within the key of a record via array notation, e.g. If we had a one-byte digit and wanted to access the least significant digit of the 0^{th} record, 65539, which would be 3, we would use `input[0][0]`. The length of an array can be accessed using the `.length` notation, e.g. the length of `input` would be `input.length`. The pseudocode will use an array notation `[]`, even when no index is specified, as a reminder that a variable is an array. The pseudocode will similarly use a brace notation `{}` to specify that a variable is a structure with dot notation access to internals.

3.1 Simple Fast Radix

Simple Fast Radix performs its first dealing pass using estimations instead of counting. This first pass estimates bucket sizes, deals into buckets or the overflow and then processes overflow. The subsequent pass deals from each predicted bucket and its overflow counterpart if necessary, such that the algorithm will still perform the pass in stable, bucket order. All other passes deal from input contiguously into exact buckets, as would an LSD Radix Sort.

Simple Fast Radix differs from LSD radix sort in that it can omit the initial standalone counting pass. Instead, Simple Fast Radix estimates bucket sizes for the first pass and processes any overflow afterward. The second pass deals from the estimated buckets and corresponding overflow buckets but otherwise, deals into the same buckets as LSD radix sort would, and all subsequent passes are

Algorithm 4 LSD Radix Sort Revisited

Input: input, buf, digits
Output: sorted input on digits

```
1: initialize()
   /* count first digit */
2: c ← digits[0] // counting digit
3: count(input, buf, c)
   /* deal on first digit, and count on second */
4: d ← digits[0] // dealing digit
5: c ← digits[1] // counting digit
6: dealCount(input, buf, d, c)
   /* loop to deal/count-next for each digit save last */
7: for i = 1 to length(digits − 1) do
8:     swapRole(input, buf)
9:     d ← digits[i] // dealing digit
10:    c ← digits[i + 1] // counting digit
11:    dealCount(input, buf, d, c)
12: end for
   /* process last digit, but do not count in deal pass */
13: swapRole(input, buf)
14: deal(input, digits[−1])
```

the same as for LSD radix sort.

3.1.1 Algorithm Details

Simple Fast Radix uses some labeled placeholders for specific digits of interest and arrays that act as buffers for dealing from input or holding counts and offsets for specific buckets. For clarity, we refer to `d` as the current digit being dealt to, `c` as the next digit to count exactly all but the last pass and `overflowCount` as the number of records that have overflown in the first pass. In Simple Fast Radix, the buffer `buf` is the same size as `input`, with alternating passes first dealing from `input` into `buf` and then swapping roles, via `swapRole`, to deal from `buf` into `input`.

In Simple Fast Radix, various bucket arrays store information about the M buckets used in each pass, representing each bucket's count or index for their corresponding digit. Simple Fast Radix uses four bucket arrays: `cur`, `next`, `over` and `exact`. `cur` stores the current buckets that the algorithm deals from during the second pass. The algorithm stores the estimated buckets in `next` for the first pass and then uses `next` to store exact buckets for all other passes. `over` stores the counts for overflow buckets, and after the overflow is processed, it stores the overflow buckets that extend the buckets stored in `cur` during the second pass. `exact` stores the counts for buckets in the next pass, which are converted to exact buckets at the end of each dealing pass after the first one, and then assigned to `next`. On the first pass, `cur` is unused as `input` is the single source bucket. Neither `cur` nor `over` are used after the second pass as no more overflow is tracked. After the second pass, the input can be treated as a single bucket as the deal from the previous pass has ordered all its buckets contiguously. As with `buf` and `input`, the `next` and `exact` bucket arrays swap roles after each pass using `swapRole`. In the last pass, `next` refers to the indices generated from the exact counts in `exact` in the previous pass, but `exact` is not used as there are no additional passes.

To avoid clutter, we specify two structures as variables for the input data and the overflow data which are cumbersome to pass around. The variable `inputData` uses `input`, `cur`, `overflow`

and over. The variable overflowData uses as fields the variables input, overflow, over and overflowCount. This structure allows us to pass inputData and overflowData as needed, and then refer to the components with the dot notation, e.g. oD.overflowCount if overflowData were passed as oD.

Some high level operations are used in the description of Simple Fast Radix, summarized here. estimate() populates the next array with estimates for the locations of buckets in buf. estimatedDealCount() deals records, based on their d-digit, into the buckets in buf which are specified in next or into overflow space if an estimated bucket is full, also counting the c-digit into exact for use in the next pass. processOverflow() sorts any overflow into buckets (specified in over) into the newly created overflow array. dealCount() deals like estimatedDealCount() save that it uses exact buckets so there is no need to check for overflow. *deal*() deals like dealCount() save that there is no counting of the next pass.

Aside from arithmetic operations, the assignment operation ← and the array notation, we use some operations without further definition. initialize() manages the creation of arrays and zeroing the count arrays. prefixSum() converts counts in an array into corresponding indices. swapRole() swaps the roles of arrays, making for a simpler notation, but not creating new arrays or materially changing contents. swapRoleClear() acts like swapRole, but zeroes the second array, re-initializing it for subsequent use. est() yields the exclusive end of an estimated bucket given a digit from a key, the size of the input and the number of buckets.

In this context getNext() encapsulates the logic behind accessing the next value from either buckets specified by cur on input, buckets specified by over on overflow or from input all but the second pass. How getNext() processes inputData will be omitted for clarity. The important thing is that getNext() ensures that estimated and overflow buckets are processed in order.

Algorithm 5 Simple Fast Radix

Input: input[], digits[]
Output: sorted input on digits

```
1:  initialize()
    /* estimate first digit */
2:  d ← digits[0] // estimate/deal digit
3:  c ← digits[1] // counting digit
4:  estimate(next[], N)
    /* deal first and count second digit */
5:  estimatedDealCount(inputData{}, buf[], overflowData{}, next[], exact[], d, c)
    /* process overflow */
6:  processOverflow(overflowData{}, d)
7:  prefixSum(exact[])
8:  swapRole(input[], buf[])
9:  swapRoleClear(next[], exact[])
    /* deal for remaining digits save last, counting for next digit*/
10: for i = 1 to length(digits) − 1 do
11:     d ← digits[i] // dealing digit
12:     c ← digits[i + 1] // counting digit
13:     dealCount(inputData{}, buf[], next[], exact[], d, c)
14:     prefixSum(exact[])
15:     swapRole(input[], buf[])
16:     swapRoleClear(next[], exact[])
17: end for
18: d ← digits[−1] // dealing digit
    /* deal last digit */
19: deal(inputData{}, buf[], next[], d)
```

3.1.2 Pseudo-code

We can break Algorithm 5 into four parts. The first part is the initialization on line 1 where variables and structures are initialized (`overflowCount`, `inputData` and `overflowData`), internal arrays are initialized, creating new space and zeroing the bucket arrays. The second part is from lines 2 to 9 and is where the estimation is performed and overflow from that initial pass is processed. The third part is from lines 10 to 17 where all pass but the last are performed, dealing from the estimated buckets and any overflow in the first of those passes but then dealing from input as in LSD

radix sort. In lines 18 to 19 the final exact deal is performed without additional counting, leaving the array sorted.

The loop making up the second part of Algorithm 5 processes the first pass on the first, least significant, digit in `digits`, counting on the second digit. On lines 2 and 3 the current digit for dealing and the next digit for counting are stored in `d` and `c` respectively. Line 4 calls `estimate()` to fill `next` with the estimated indices of the buckets that will be subsequently dealt into given N (the length of `input`). Line 5 calls `estimatedDealCount()` to deal all records (access to all records coming from `inputData`), based on the current digit `d`, into either their estimated buckets in `buf`, as specified in `next` or into overflow space when the estimated buckets are full. Overflow space will be the beginning of `input`. During this dealing step, the bucket array `exact` will be updated with counts on digit `c`, the bucket array `over` will be updated with exact counts for the overflown records, given digit `d`, and the number of records overflown will be recorded in `overflowCount`. Line 6 calls `processOverflow()`, which will create an `overflow` as a buffer of size `overflowCount`. With enough space assured in `overflow`, the processing of overflow will convert the bucket array `over` from counts into indices for the exact overflow buckets in `overflow` and will then deal into that space from the beginning of `input` where overflow is stored. Finally, in lines 8 and 9, the roles of arrays are swapped for`input` and `buf` as well as for `next` and `exact`, with `exact` being cleared for subsequent use.

The third part of Algorithm 5 deals the middle passes, dealing and performing counting for each pass. Lines 11 and 12 store the digit to deal on in `d` and the digit to count on in `c`. In line 13, `dealCount()` deals exactly as `estimatedDealCount()` on line 5, save that there is no overflow as `next` refers to exact buckets. Line 14 calls `prefixSum` to convert the bucket array `exact` into exact bucket indices for the next round of dealing. As in the previous part, the pass completes with the roles of arrays being swapped in lines 15 and 16.

The final part of Algorithm 5 deals the last pass into place, leaving `input` sorted. Line 18 assigns the last digit to `d` and line 19 calls `deal()` to deal exactly into `buf`, which is just the original

Algorithm 5.1 estimatedDealCount

Input: iD, buf[], oD, next, exact, d, c

```
 1: estimatedDealCount  /* Θ(N) */
 2: for i = 0 to N − 1 do
 3:    record ← getNext()
 4:    exact[record[c]]++
       /* If we have not overflowed */
 5:    if next[record[d]] < est(record[d], N, next.length) then
 6:       buf[next[record[d]]] ← record /* Deal into buf based on next */
 7:       next[record[d]]++ /* Increment next for the bucket that was dealt into */
          /* Else we have overflowed */
 8:    else
 9:       oD.input[oD.overflowCount − oD.size] ← record
10:       oD.overflowCount++
11:       oD.over[record[d]]++ /* Increment over for the digit that was dealt into overflow*/
12:    end if
13: end for
```

`input` array after an even number of passes, but now it is guaranteed sorted.

3.1.3 Simple Fast Radix Example

In order to examine the details of Algorithm 6 we will consider the example used inThiel et al. [2016] that uses an `input` of $N = 16$ records [$A1$, $D7$, $5D$, 89, $B6$, 18, $A3$, 39, 98, $F3$, 52, $F9$, $F6$, 67, $9A$, 05]. Each key is represented by four hexadecimal digits, so $M = 16$ and `digits` are [0, 1]. A more thorough example of estimation and processing is given for Fast Radix later.

The initial estimation pass is shown in Figure 7 and is performed by `estimatedDealCount` as shown in Algorithm 5.1. The most significant digit is counted and stored in `exact` on line 4, identified as the underlining of the most significant digit (`c` from line 3 of Algorithm 5) of records dealt into `buf` or the overflow area of `input`. The overflow area also uses an underscore on the current digit (`d` from line 2 of Algorithm 5) to prepare the overflow counts for the subsequent overflow processing pass. The arrows are labeled in the order that records are processed.

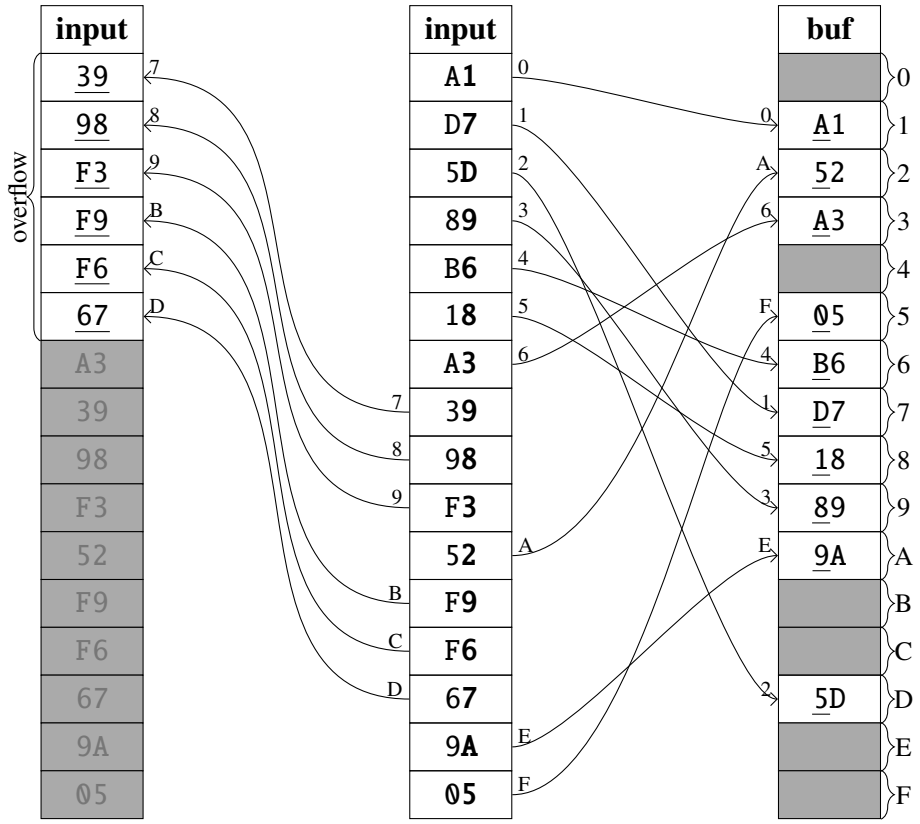

Figure 7: Dealing and counting pass. Dealing from top to bottom in the second *input* is shown with arrows, with dealing into estimated buckets going to *buf* and overflow going to the beginning of the first *input*. Based on the example from Thiel et al. [2016]

Records 0 through 6 are dealt into empty estimated buckets as the check for a bucket having room on line 5 passes, with the dealing happening on line 6 and the index of the bucket dealt into being incremented on line 7. Record 7 fails the check for space in the estimated bucket as record 3 was already placed there, and so it is placed into the beginning of `input` on line 9, the overflow count is increased on line 10 and the overflow count for the bucket in digit `d` in `over` is incremented on line 11. All other records are processed in the same way on this pass.

After the estimated dealing pass, any overflow must be processed. The overflow processing is described by Algorithm 5.2 and shown in Figure 8. The overflow counts in `over` are converted into bucket indices on line 2 and an an overflow buffer to hold these buckets is initialized on line 3. Each record in the input is then dealt into these buckets on lines 4 to 6.

Algorithm 5.2 ProcessOverflow

Input: oD, d
Output: oD.over[] points to populated buckets in oD.overflow

```
1: processOverflow /* O(N) */
2: prefixSum(oD.over)
3: oD.overflow ← newrecord[oD.overflowCount * 2]
4: for i = 0 to od.overflowCount do
5:     oD.overflow[oD.over[oD.input[i][d]]] ← oD.input[i]
6:     oD.over[oD.input[i][d]]++
7: end for
```

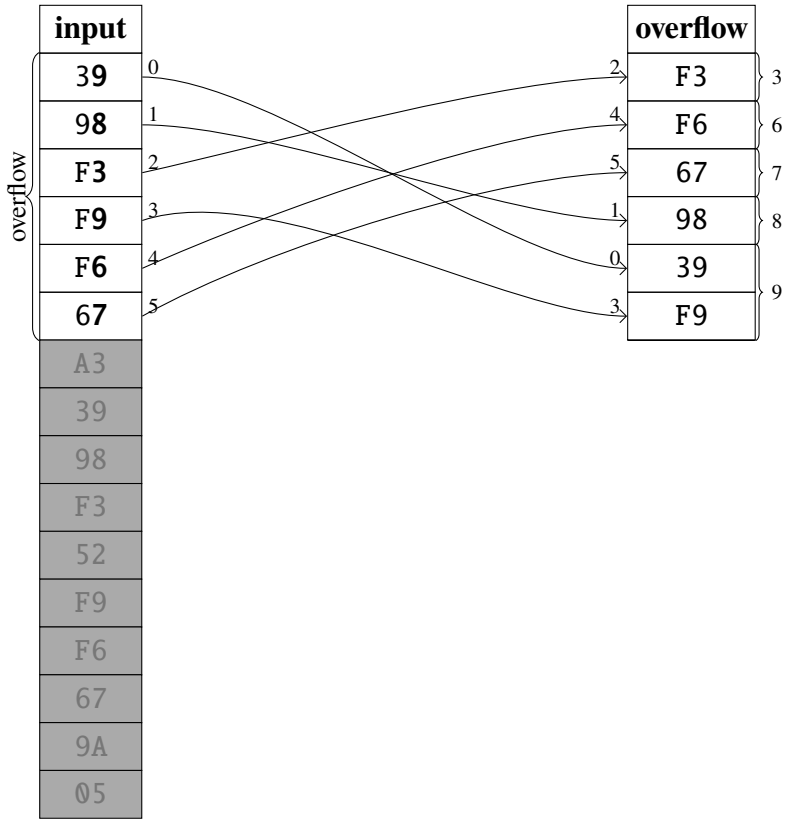

Figure 8: Overflow at the beginning of `input` is dealt into the new `overflow` buffer with the buckets built from the overflow counts in the previous pass.

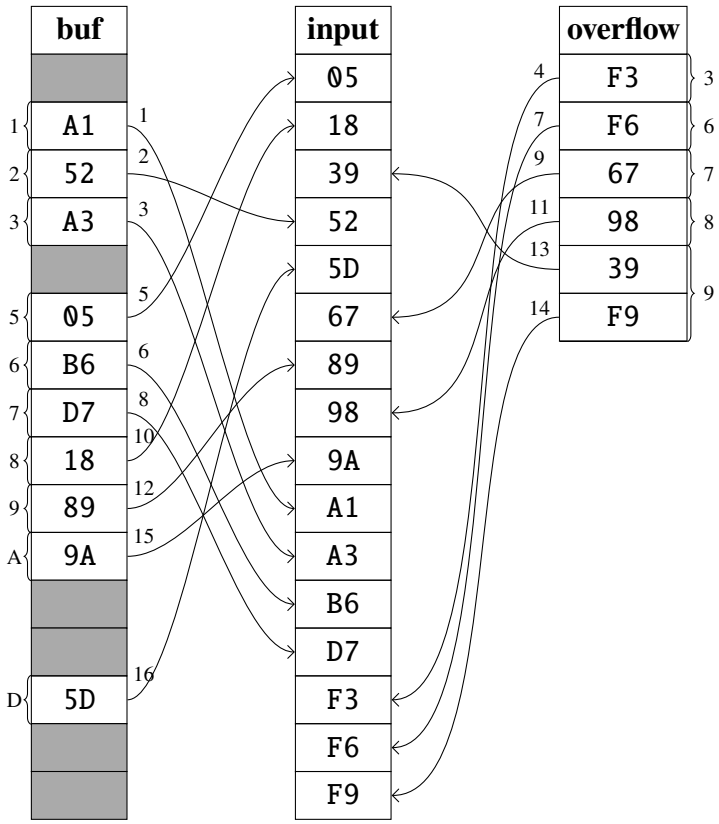

Figure 9: Dealing pass. The buckets' low-order hex digits are shown with braces on *B* and *Over*. The interleaving is shown in the dealing, represented by numbered arrows.

Algorithm 5.3 deal

Input: iD, buf[], exact, d

```
1: deal /* Θ(N) */
2: for i = 0 to N − 1 do
3:     record ← getNext()
4:     buf[exact[record[d]]] ← record /* Deal into buf based on next */
5:     exact[record[d]]++ /* Increment next for the bucket that was dealt into */
6: end for
```

The final pass on the last digit is described in Algorithm 5.3 and shown in Figure 9. Each record is pulled from their estimated buckets in `buf` or the overflow buckets in `overflow` in the order shown on the labeled arrows, shown on line 3. `getNext` ensures that each estimated bucket that has records is processed in turn, and that any overflow buckets that contain records are processed immediately after their corresponding estimated bucket, with corresponding buckets identified by the labeled braces on the sides of `buf` and `overflow`. The dealing happens on line 4 and the index for the bucket of the dealt digit in `exact` is incremented on line 5. With the records in `buf` and `overflow` dealt exactly back into `input`, `input` is sorted.

3.1.4 Discussion

In this section we have seen how Simple Fast Radix omits the initial counting pass. The subsequent section on Fast Radix contains a more thorough discussion of the details that it has in common with Simple Fast Radix.

Theorem 1. *Simple Fast Radix (Algorithm 5) sorts an input correctly.*

Proof. Simple Fast Radix deals the same digits in the same order as LSD radix sort in every pass. The used portion of an estimated bucket combined with any corresponding overflow bucket in Simple Fast Radix has the same records as a corresponding exact bucket in LSD radix sort after the first pass. The exact buckets identified during the first pass in Simple Fast Radix and LSD radix sort are the same. All passes after the first are otherwise the same for both Simple Fast Radix and LSD Radix Sort. LSD radix sort is known to sort input correctly. Therefore, Simple Fast Radix must yield the same sorted result as LSD radix sort. ■

3.2 Fast Radix

Fast Radix performs its dealing passes using estimations instead of counting for all but the last pass, including an overflow processing step in between passes. Each pass estimates bucket sizes, deals into buckets or the overflow and then processes overflow. Subsequent passes deal from each predicted bucket and then from its overflow counterpart if necessary, such that the algorithm still performs the next deal in stable, bucket order. The algorithm counts the last digit in the penultimate pass to ensure exact buckets in the last pass, avoiding an extraneous copy pass and putting everything in order.

Fast Radix differs from Simple Fast Radix in that only the last pass gets exact counts, with all other passes using estimates. While Simple Fast Radix could deal overflow directly to the beginning of `input` during its initial estimation pass, as there was no opportunity for conflict, Fast Radix cannot make that assumption on subsequent passes. When overflow found on a pass exceeds the overflow space already available in `overflow`, the Fast Radix algorithm creates a new `overflow` buffer that can hold twice the amount of overflow found. The algorithm can then assign overflow buckets to the first half of `overflow` and overflow from the subsequent pass can be dealt into the second half, with the algorithm dealing any excess safely into the beginning of `input`. The algorithm avoids a conflict where dealing overflow would overwrite parts of `input` that it had not yet dealt out by first dealing overflow into an overflow space that is at least as big as the existing overflow buckets.

3.2.1 Algorithm Details

Fast Radix uses some labelled placeholders for specific digits of interest and arrays that act as buffers for dealing from input or holding counts and offsets for specific buckets. For clarity, we refer to `d` as the current digit that the algorithm will use to deal, `c` as the digit that the algorithm will use to count exactly during the penultimate pass and `overflowCount` as the number of records that have

overflown in a given pass. In Fast Radix, the buffer `buf` is the same size as `input`, with alternating passes first dealing from `input` into `buf` and then swapping roles, via `swapRole`, to deal from `buf` into `input`. Fast Radix also uses an overflow buffer `overflow` that is twice the size of the current overflow (initially 0).

In Fast Radix, various bucket arrays store information about the M buckets used in each pass, representing the count or index of each bucket for their corresponding digit. Fast Radix uses four bucket arrays, the current buckets being dealt from `cur`, the overflow buckets `over` which extend the buckets in `cur` as needed, the exact buckets of the last digit `last` to allow for exact dealing on the final pass and the target buckets for the next pass, `next` which is populated by estimates save on the last pass where it is populated based on the exact counts in `last`. On the first pass `cur` and `over` are unused as `input` is the single source bucket. As with `buf` and `input`, the `cur` and `next` bucket arrays swap roles after each pass using `swapRole`. The array `last` is unused until the penultimate pass when it is populated with the exact counts for the last pass. In the last pass, `next` refers to the indices generated from the exact counts in `last`.

To avoid clutter, we specify two structures as variables for the input data and the overflow data which are cumbersome to pass around. The variable `inputData` uses `input`, `cur`, `overflow` and `over`. The variable `overflowData` uses as fields the variables `input`, `overflow`, `over` and `overflowCount`, as well as a convenience field `size` that is just half of `overflow.length`. This structure allows us to pass `inputData` and `overflowData` as needed, and then refer to the components with the dot notation, e.g. `oD.overflowCount` if `overflowData` were passed as `oD`.

Some high level operations are used in the description of Fast Radix, summarized here. `estimate()` populates the `next` array with estimates for the locations of buckets in `buf`. `estimatedDeal()` deals records, based on their `d`-digit, into the buckets in `buf` which are specified in `next` or into overflow space if an estimated bucket is full. `processOverflow()` creates a bigger `overflow` array if needed, and then sorts any overflow into buckets (specified in `over`) in the first half of either the existing `overflow` array or the newly created `overflow` array

(which then becomes the overflow array). `estimatedDealCount()` behaves exactly like `estimatedDeal()`, save that it will also take care to perform an exact count of the last digit in the keys for all records. `deal()` deals like `estimatedDeal()` save that it uses exact buckets so there is no need to check for overflow.

Aside from arithmetic operations, the assignment operation ← and the array notation, we use some operations without further definition. `initialize()` manages the creation of arrays and zeroing the count arrays. `prefixSum()` converts counts in an array into corresponding indices. `swapRole()` swaps the roles of arrays, making for a simpler notation, but not creating new arrays or materially changing contents. `est()` yields the exclusive end of an estimated bucket given a digit from a key, the size of the input and the number of buckets.

In this context `getNext()` encapsulates the logic behind accessing the next value from either buckets specified by `cur` on `input`, buckets specified by `over` on `overflow` or from `input` during the first pass. How `getNext()` processed `inputData` will be omitted for clarity. The important thing is that `getNext()` ensures that estimated and overflow buckets are processed in order.

3.2.2 Pseudo-code

Algorithm 6 can be broken into four parts. The first part is the initialization on line 1 where variables and structures are initialized (`overflowCount`, `inputData` and `overflowData`), internal arrays are initialized, creating new space and zeroing the bucket arrays. The second part is from lines 2 to 9 and is where all but the last two passes are performed. The third part is from lines 10 to 17 where the penultimate pass is performed. In lines 18 to 20 the final exact deal is performed, leaving the array sorted.

The loop making up the second part of Algorithm 6 processes the passes for all but the last two digits specified in `digits`, from least to most significant. On line 3 the current digit for dealing

Algorithm 6 Fast Radix

Input: input[], digits[]
Output: sorted input on digits

1: $initialize()$
   /* estimate and deal for remaining digits save last two*/
2: **for** $i = 0$ **to** $digits.length - 2$ **do**
3: $d \leftarrow digits[i]$ // deal digit
4: $estimate(next[], N)$
5: $estimatedDeal(inputData\{\}, buf[], overflowData\{\}, next[], d)$
6: $processOverflow(overflowData\{\}, d)$
7: $swapRole(input[], buf[])$
8: $swapRole(cur[], next[])$
9: **end for**
   /* get exact count of final digit during second-to-last deal*/
10: $d \leftarrow digits[-2]$ // deal digit
11: $c \leftarrow digits[-1]$ // counting final digit
12: $estimate(next[], N)$
13: $estimatedDealCount(inputData\{\}, buf[], overflowData\{\}, next[], last[], d, c)$
14: $processOverflow(overflowData\{\}, d)$
15: $prefixSum(last[])$
16: $swapRole(input[], buf[])$
17: $swapRole(cur[], next[])$
   /* deal last digit */
18: $next[] \leftarrow last[]$ // next should point to exact counts
19: $d \leftarrow digits[-1]$ // deal digit
20: $deal(inputData\{\}, buf[], next[], d)$

is stored in d. Line 4 calls estimate() to fill next with the estimated indices of the buckets that will be subsequently dealt into given N (the length of input). Line 5 calls estimatedDeal() to deal all records (access to all records coming from inputData), based on the current digit d, into either their estimated buckets in buf, as specified in next or into overflow space when the estimated buckets are full. Overflow space will be in the first half of overflow and the beginning of input, respectively. During this dealing step, the bucket array over will be updated with exact counts for the overflown records, given digit d, and the number of records overflown will be recorded in overflowCount. Line 6 calls processOverflow(), which will expand the overflow buffer to twice overflowCount if overflowCount is more than half the length of the existing overflow buffer overflow. With enough space assured in overflow, the processing of overflow will convert the bucket array over from counts into indices for the exact overflow buckets in the last half of overflow and will then deal into that space from the first half of overflow and if warranted, from the beginning of input. Finally, in lines 7 and 8, the roles of arrays are swapped forinput and buf as well as for cur and next, clearing n.

The third part of Algorithm 6 deals the penultimate pass while also counting for the last pass. Lines 10 and 11 store the next-to-last digit in d and the last digit, which will be counted during this pass, in c. Line 12 fills next with the estimated indices as done previously. In line 13, estimatedDealCount() deals exactly as estimatedDeal() on line 5, save that the bucket array last is filled with the exact counts of digit c on the records dealt. Line 14 processes overflow as normal. Line 15 calls prefixSum to convert the bucket array last into exact bucket indices for the final round of dealing. As in the previous part, the pass completes with the roles of arrays being swapped in lines 16 and 17.

The final part of Algorithm 6 deals the last pass into place, leaving input sorted. On line 18, instead of next being populated with estimates as done previously, it is assigned the exact bucket positions as determined on line 15 of the previous pass. Line 19 assigns the last digit to d. Finally, line 20 calls deal() to deal exactly into buf, which is just the original input array after an even

Algorithm 6.1 estimate

Input: next[], N
Output: next[] will be populated with estimated bucket sizes

```
1: estimate  /* Θ(M) */
2: next[0] ← 0
3: for i = 0 to next.length-1 do
4:     next[i + 1] ← est(i, N, next.length)
5: end for
```

number of passes, but now it is guaranteed sorted.

3.2.3 Fast Radix Example

In order to examine the details of Algorithm 6 we will consider as an example an `input` of $N = 16$ records [$E482$, $7DB9$, $C602$, $792E$, $4A3D$, 0140, $E4E1$, $9A14$, $8B91$, $4F1B$, $2F2D$, $226F$, $EA92$, $1BB7$, $573D$, $9F13$]. Each key is represented by four hexadecimal digits, so $M = 16$ and `digits` are $[0, 1, 2, 3]$.

The loop between lines 2 and 9 will cover the first two passes over the lowest order digits. In each pass, an estimate will be performed using the estimate function described in Algorithm 6.1. As `next.length` is 16 and the size of input is 16, `est` will estimate that each bucket is of size 1, and estimate will populate `next` with bucket locations as shown by the brace brackets to the right of `buf` in Figure 10.

The first call of the `estimatedDeal` function described in Algorithm 6.2 is shown in Figure 10. The labeled arrows show the order in which records are returned by `getNext` and the bold digit of each record in `input` shows the result of calls to `record[d]`. As the check for space in the target bucket on line 4 passes for the 0^{th} and 1^{st} records, they get moved into the estimated 2 and 9 buckets in `buf` respectively and their bucket pointers are incremented. Since the test on 4 fails for the 3^{rd} record, as the estimated bucket for 2 only had space for one record and the 0^{th} record was already

Algorithm 6.2 estimatedDeal

Input: iD, buf[], oD, next, d

```
 1: estimatedDeal  /* Θ(N) */
 2: for i = 0 to N − 1 do
 3:    record ← getNext()
       /* If we have not overflowed */
 4:    if next[record[d]] < est(record[d], N, next.length) then
 5:       buf[next[record[d]]] ← record /* Deal into buf based on next */
 6:       next[record[d]]++ /* Increment next for the bucket that was dealt into */
          /* Else we have overflowed */
 7:    else
 8:       if overflowCount <= oD.size then
 9:          oD.overflow[oD.size + oD.overflowCount] ← record
10:       else
11:          oD.input[oD.overflowCount − oD.size] ← record
12:       end if
13:       oD.overflowCount++
14:       oD.over[record[d]]++ /* Increment over for the digit that was dealt into overflow*/
15:    end if
16: end for
```

placed there, an deal to overflow must be made between lines 7 and 15. As the initial `overflow` buffer has no space (shown just as a label in Figure 10), the check on line 8 fails and the 3^{rd} record is dealt into the beginning of the `input` on line 11. In dealing to the overflow, `overflowCount` is increased (line 13) and the bucket count `over` is also increased for bucket `record[d]` (line 14), with `record[d]` being show in Figure 10 as the underlined digits in the overflow area. The first overflow is dealt to the 0^{th} position in `input` as `overflowCount` was 0 and `oD.size` was also also 0. Dealing proceeds similarly for the remaining 12 records.

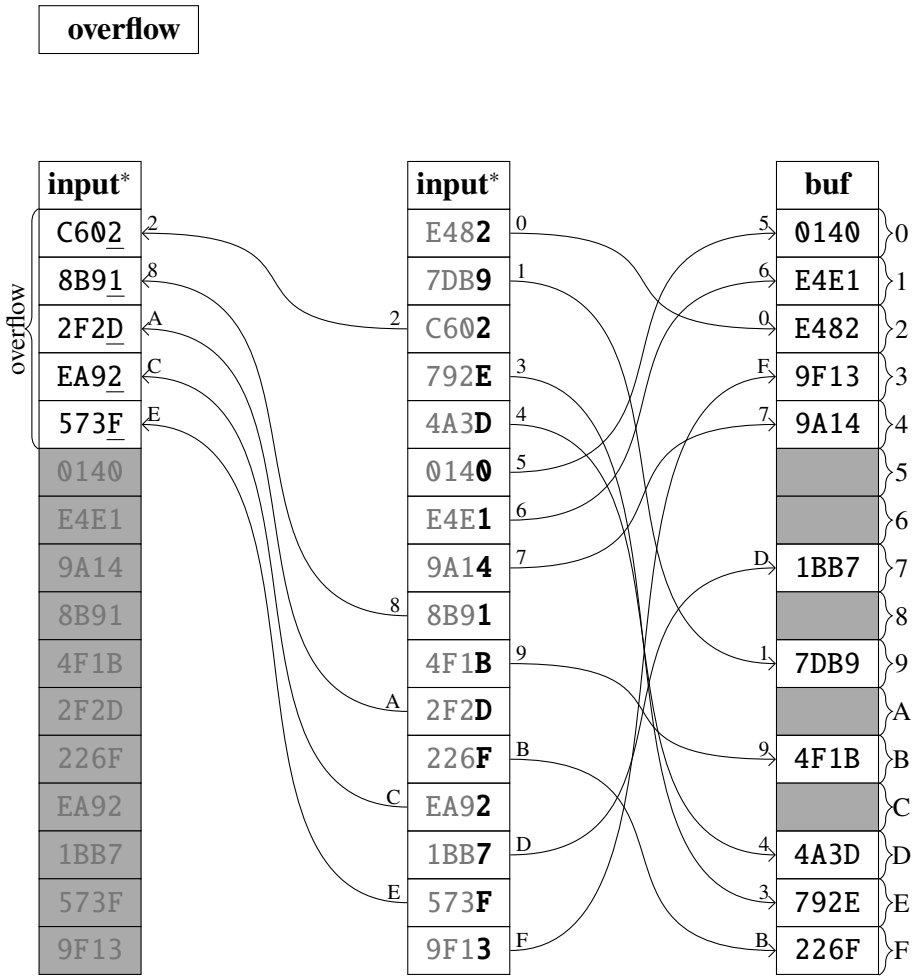

Figure 10: The first estimated dealing pass of our Fast Radix example, dealing on the least significant digit. Dealing is from `input`. The copy of `input` in the center shows the state of the input before this dealing pass, with the digit currently being dealt shown in bold and all other digits obscured. As `overflow` has not been assigned any space yet, it is shown only as a label at the top. Overflow is dealt to the beginning of `input`; given its double-duty `input` is shown again on the left as it would be after dealing is completed for this pass. The overflow area at the beginning of `input` is identified with the brace labeled "overflow". Digits in the overflow area with an underscore are counted for use when processing overflow in the next step (shown in Figure 11). The estimated buckets are labeled with braces to the right of `buf`, which is shown as it would be after dealing is completed for this pass. Labeled arrows show dealing into estimated buckets that correspond to the identified digit being dealt on or to the overflow if the corresponding bucket is already full. The labels on arrows showing the order in which records are dealt, with labels appearing on both sides of the arrow for clarity. Greyed cells represents areas that are unused during dealing in this pass. The * on `input` denotes that it is the original input.

Algorithm 6.3 ProcessOverflow

Input: oD, d
Output: over[] point to populated buckets in the beginning of overflow

```
1:  processOverflow /* O(N) */
2:  prefixSum(oD.over)
3:  if oD.overflowCount > oD.size then
4:      over_tmp ← overflow]
5:      overflowSource ← over_tmp[−oD.size :] /* Slice the last half into overflowSource */
6:      oD.overflow ← newrecord[oD.overflowCount ∗ 2]
7:  else
8:      overflowSource ← overflow[−oD.size :] /* Slice the last half into overflowSource */
9:  end if
10: for i = 0 to Math.min(od.overflowCount, oD.size) do
11:     oD.overflow[oD.over[overflowSource[i][d]]] ← overflowSource[i]
12:     oD.over[overflowSource[i][d]]++
13: end for
14: for i = 0 to oD.overflowCount − oD.size do
15:     oD.overflow[oD.over[oD.input[i][d]]] ← oD.input[i]
16:     oD.over[oD.input[i][d]]++
17: end for
18: oD.size ← oD.overflowCount/2
19: oD.overflowCount ← 0
```

The first call of `procesOverflow` shown in Algorithm 6.3 is described in Figure 11. The buckets identified with braces along the top of `overflow` in Figure 11 are created on line 2 when the counts on the current digit of the overflow records are prefix summed. As the `overflowCount` exceeds the old size of the overflow area (`oD.size`), a new overflow is created in lines 3 to 7. As 5 elements had overflown, the new length of `overflow` is twice that amount, 10. The overflow, entirely at the beginning of `input` can then be moved into the newly prepared buckets on lines 14 to 16. In Figure 11, the bold digit in each record of the input is represented by `oD.input[i][d]`.

Before the next dealing pass, the algorithm switches the buffers' roles, denoted by the * changing from `input` to `buf` and vice versa on each pass.

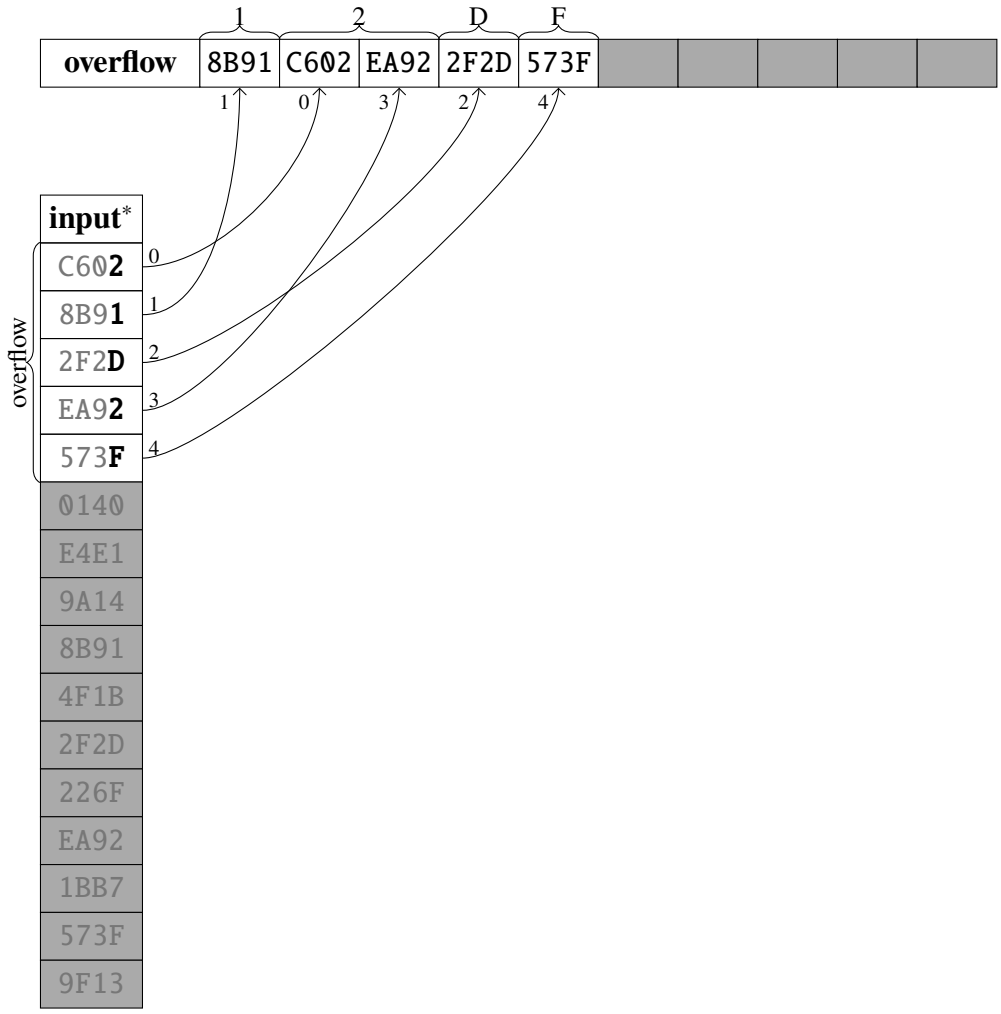

Figure 11: The first overflow processing pass of our Fast Radix example, processing overflow on the least significant digit. The size of `overflow` is set to 10, twice the `overflowCount` established during the first dealing pass (shown in Figure 10), since that is greater than its previous size of 0. Exact buckets based on overflow counts from the first dealing pass (shown in Figure 10) are identified with braces above the left half of `overflow`. Dealing is from beginning of `input`, that overflow area identified by a brace labeled "overflow", with the digit currently being dealt shown in bold and all other digits obscured. Labeled arrows show the dealing into exact buckets in the left half of `overflow`, with the label showing the order in which records are dealt into the buckets that correspond to their identified digit. Greyed cells represents areas that are unused during overflow processing in this pass. The * on `input` denotes that it is the original input.

The second call of the `estimatedDeal` function described in Algorithm 6.2 is shown in Figure 12. Like the first call of `estimatedDeal`, estimate buckets are shown as braces on the right of `buf` (as expected, the same estimates). However, this pass now as an input split into several buckets, not all contiguous, with some corresponding overflow buckets. These buckets are ordered by the braces to the left of `input` and the braces above `overflow`. The arrows are still labeled in the order in which `getNext` will return records, with a bucket in `input` being dealt out and then any corresponding bucket in `overflow` being dealt out before moving to the next available bucket in `input`. In this second pass, there is overflow space available at the end of `overflow`, so the first 5 overflows pass the check on line 8 of Algorithm 6.2 and are dealt to the second half of `overflow` on line 9. However, the last element (F^{th}) dealt had its estimated bucket filled by the B^{th} element and thus the check on line 4 fails and the overflow space is full so the check on line 8 also fails, leaving $573F$ to be dealt to the beginning of `input` on line 11.

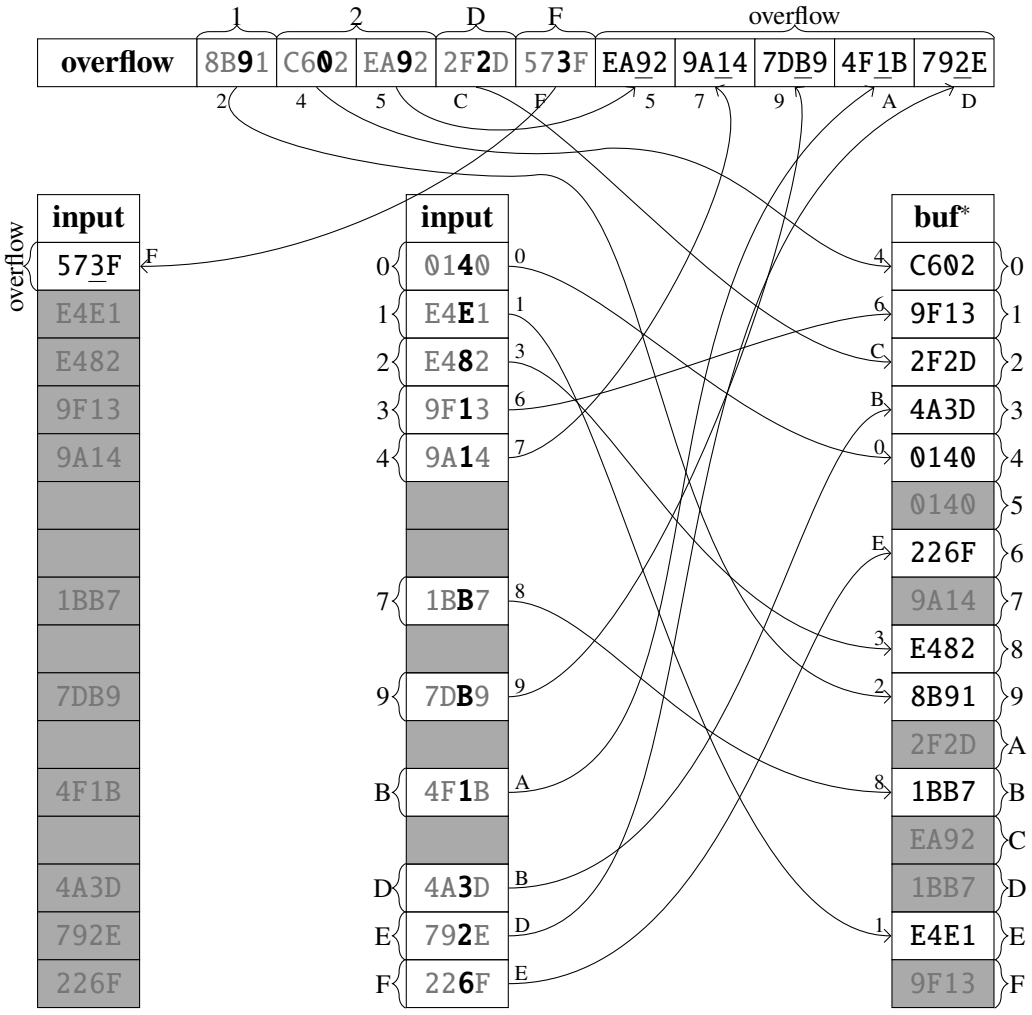

Figure 12: The second estimated dealing pass of our Fast Radix example, dealing on the next least significant digit. Dealing is from the buckets specified with braces to the left of `input` and from the buckets specified with braces above the left half of `overflow`. The copy of `input` in the center shows the state of the input before this dealing pass. The non-empty buckets in `input` and `overflow` show the digit currently being dealt in bold and all other digits obscured. Any overflow that would not fit in the right half of `overflow` will be dealt to the beginning of `input`; given its double-duty `input` is shown again on the left as it would be after dealing is completed for this pass. The overflow areas on the right half of `overflow` and the beginning of `input` are identified with the braces labeled "overflow". Digits in the overflow areas with an underscore are counted for use when processing overflow in the next step (shown in Figure 13). The estimated buckets are labeled with braces to the right of `buf`, which is shown as it would be after dealing is completed for this pass. Labeled arrows show dealing into estimated buckets that correspond to the identified digit being dealt on or to the overflow if the corresponding bucket is already full. The labels on arrows showing the order in which records are dealt, with labels appearing on both sides of the arrow for clarity. Greyed cells represents areas that are unused during dealing in this pass. Note that the roles of `input` and `buf` have been swapped, with the * on `buf` identifying that it is the original input.

The second `processOverflow` pass highlights how overflow is dealt from the old overflow, `over_temp`, into the newly define overflow buckets in the new, larger `overflow` before dealing overflow from `input` as in the first `processOverflow` pass.

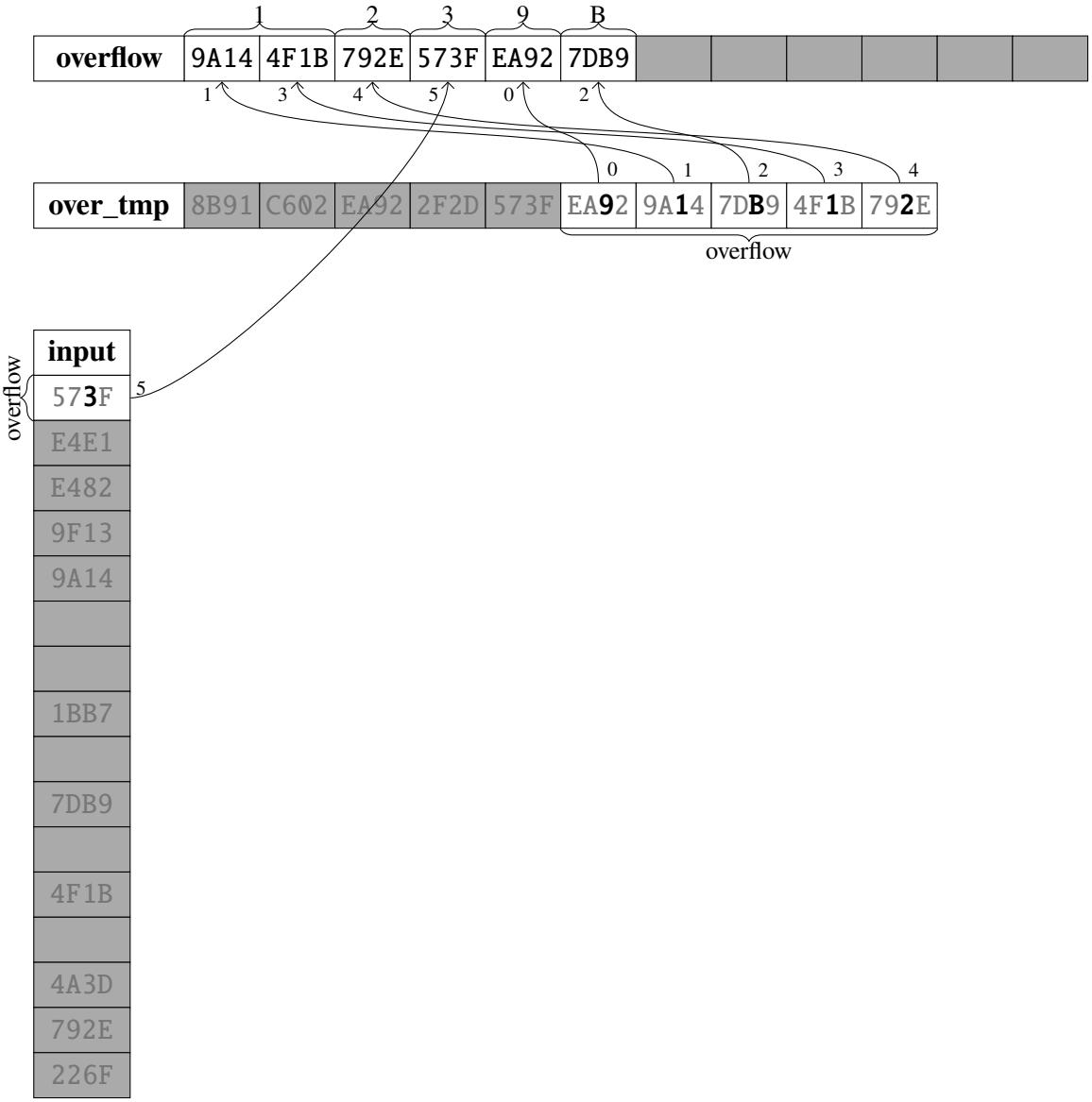

Figure 13: The second overflow processing pass of our Fast Radix example, processing overflow on the next least significant digit. The size of `overflow` is set to 12, twice the `overflowCount` established during the second dealing pass (shown in Figure 12), since that is greater than its previous size of 10. The old `overflow` is shown as `over_tmp` and exists only long enough to deal out its records into the new `overflow`. Exact buckets based on overflow counts from the second dealing pass (shown in Figure 12) are identified with braces above the left half of `overflow`. Dealing is from the right half of `over_tmp` and then the beginning of `input`, with overflow areas identified by a brace labeled "overflow". The digit currently being dealt is shown in bold and all other digits obscured inside the overflow areas. Labeled arrows show the dealing into exact buckets in the left half of `overflow`, with the label showing the order in which records are dealt into the buckets that correspond to their identified digit. Greyed cells represents areas that are unused during overflow processing in this pass.

Algorithm 6.4 estimatedDealCount

Input: iD, buf[], oD, next, last, d, c

```
 1: estimatedDealCount  /* Θ(N) */
 2: for i = 0 to N − 1 do
 3:    record ← getNext()
 4:    last[record[c]]++
       /* If we have not overflowed */
 5:    if next[record[d]] < est(record[d], N, next.length) then
 6:       buf[next[record[d]]] ← record /* Deal into buf based on next */
 7:       next[record[d]]++ /* Increment next for the bucket that was dealt into */
          /* Else we have overflowed */
 8:    else
 9:       if overflowCount <= oD.size then
10:          oD.overflow[oD.size + oD.overflowCount] ← record
11:       else
12:          oD.input[oD.overflowCount − oD.size] ← record
13:       end if
14:       oD.overflowCount++
15:       oD.over[record[d]]++ /* Increment over for the digit that was dealt into overflow*/
16:    end if
17: end for
```

The penultimate deal pass uses Algorithm 6.4 instead of Algorithm 6.2 and is shown in Figure 14. The main practical difference is that every dealt record is used to increment the counts stored in `last` on line 4, identified as the underlining of the most significant digit (`c` from line 11 of Algorithm 6) of records dealt into `buf` or the overflow area of `overflow`. The overflow area also has the usual underscore on the current digit (`d` from line 10 of Algorithm 6) to prepare the overflow counts for the subsequent overflow processing pass.

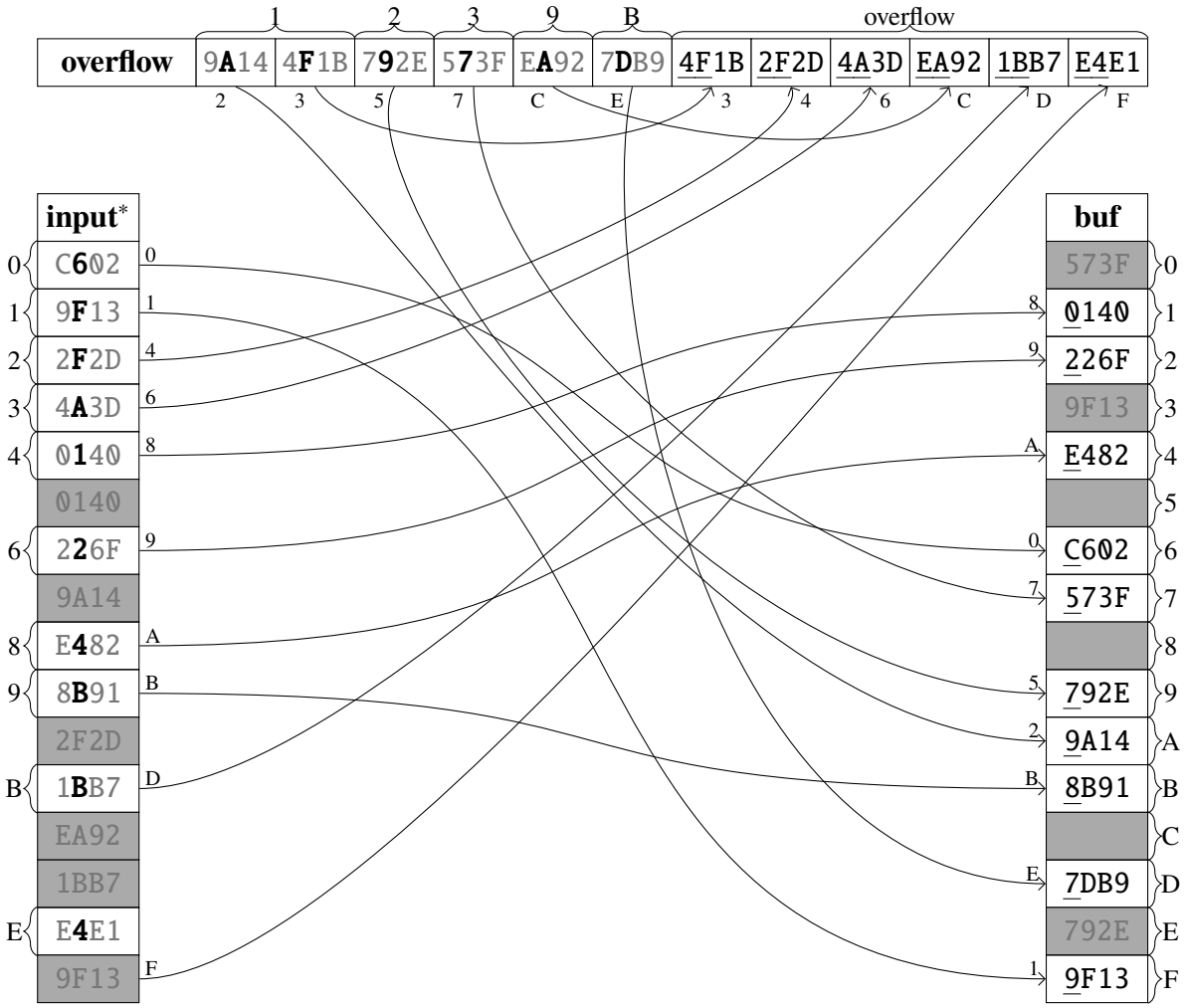

Figure 14: The penultimate dealing pass of our Fast Radix Example, dealing on the next most significant digit into estimated buckets and counting the final digit in preparation for the last pass. Dealing is from the buckets specified with braces to the left of `input` and from the buckets specified with braces above the left half of `overflow`. `input` shows the state of the input before this dealing pass. The non-empty buckets in `input` and `overflow` show the digit currently being dealt in bold and all other digits obscured. The overflow area on the right half of `overflow` is identified with a brace labeled "overflow". Digits in the overflow area with an underscore are counted for use when processing overflow in the next step (shown in Figure 15). The estimated buckets are labeled with braces to the right of `buf`, which is shown as it would be after dealing is completed for this pass. Labeled arrows show dealing into estimated buckets that correspond to the identified digit being dealt on or to the overflow if the corresponding bucket is already full. The labels on arrows showing the order in which records are dealt, with labels appearing on both sides of the arrow for clarity. Greyed cells represents areas that are unused during dealing in this pass. Note that the roles of `input` and `buf` have been swapped, with the * on `input` identifying that it is the original input. In this pass, the beginning of `input` is not needed for overflow as the space in `overflow` is sufficient for this pass, as such the extra `input` shown in previous dealing passes is omitted for clarity.

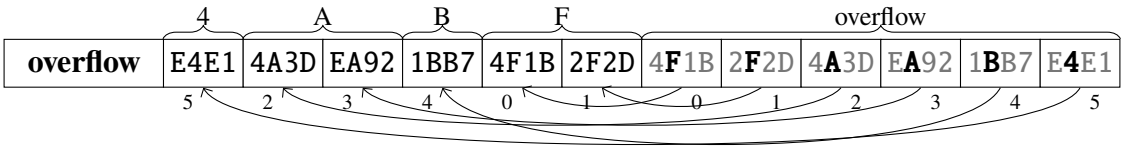

Figure 15: The overflow processing during the penultimate pass of our Fast Radix example, processing overflow on the second most significant digit. The size of `overflow` is unchanged at 12, as twice the `overflowCount` established during the penultimate dealing pass (shown in Figure 14) would not be larger. Exact buckets based on overflow counts from the penultimate dealing pass (shown in Figure 14) are identified with braces above the left half of `overflow`. Dealing is from the right half of `overflow`, with the overflow area identified by a brace labeled "overflow". The digit currently being dealt is shown in bold and all other digits obscured inside the overflow area. Labeled arrows show the dealing into exact buckets in the left half of `overflow`, with the label showing the order in which records are dealt into the buckets that correspond to their identified digit. As there was no overflow in `input`, it is omitted for clarity.

Algorithm 6.5 deal

Input: iD, buf[], next, d

```
1: deal /* Θ(N) */
2: for i = 0 to N − 1 do
3:     record ← getNext()
4:     buf[next[record[d]]] ← record /* Deal into buf based on next */
5:     next[record[d]]++ /* Increment next for the bucket that was dealt into */
6: end for
```

The final deal is shown in Figure 16, using `deal` from Algorithm 6.5. As the algorithm determined the exact bucket counts in the penultimate pass, the buckets shown with braces to the right of `buf` are exact, and there is no contention while dealing.

3.2.4 Discussion

Fast Radix, as explained here, does not describe sorting when there is an odd number of digits or precisely where there is a single digit. As Diverting Fast Radix can request that Fast Radix sort an odd number of passes, this comes up and is an issue with some LSD radix sort implementations. This thesis's implementations demonstrate trivial solutions, and there is room for future work to pick solutions that work well with specific technical optimizations.

The pseudocode uses the `getNext` function to avoid a detailed description of how the algorithm processes buckets in order. A comprehensive examination of approaches to do this appears uninformative. Implementations used for results in Chapter 4 use a few approaches, with few showing direct performance benefits; some approaches are shorter, if not more transparent than others.

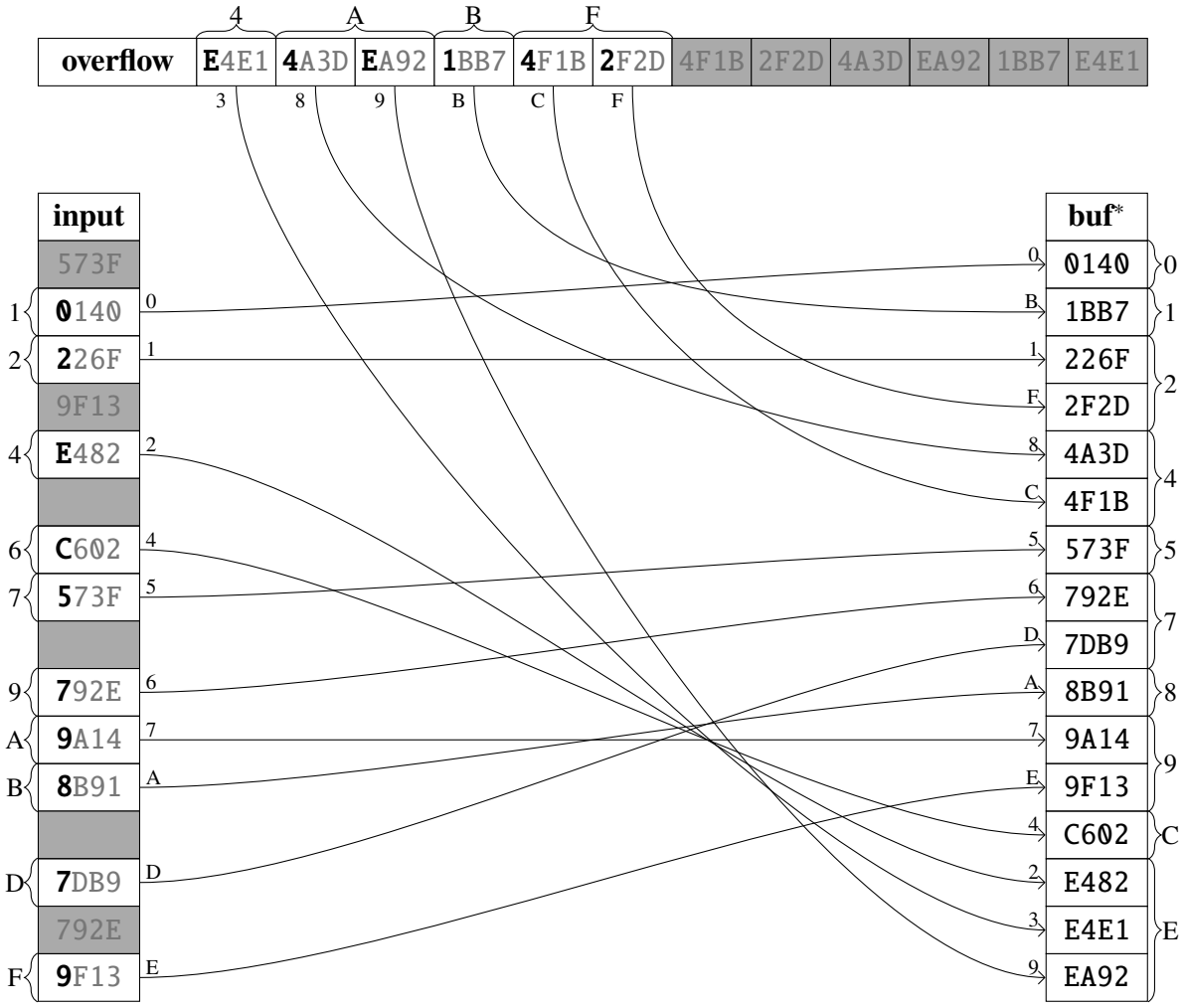

Figure 16: The final dealing pass of our Fast Radix Example deals all records exactly into sorted position based on the most significant digit. Dealing is from the buckets specified with braces to the left of `input` and from the buckets specified with braces above the left half of `overflow`. `input` shows the state of the input before this dealing pass. The non-empty buckets in `input` and `overflow` show the digit currently being dealt in bold and all other digits obscured. As this is an exact dealing pass, there is no need for the overflow areas identified in the previous dealing passes. The exact buckets are labeled with braces to the right of `buf`, which is shown as it would be after dealing is completed for this pass. Labeled arrows show dealing into exact buckets that correspond to the identified digit being dealt on. The labels on arrows show the order in which records are dealt, with labels appearing on both sides of the arrow for clarity. Greyed cells represents areas that are unused during dealing in this pass. Note that the roles of `input` and `buf` have been swapped, with the * on `buf` identifying that it is the original input, and at the end of this pass contains all records in sorted order.

A decision on the optimal approach to managing overflow is an area for future work. Some technical approaches in the literature might be adjusted to take on this overflow area's role. There are also edge cases where, for example, more than half the records overflow and the algorithm could apply different strategies for better results. The best starting size of the overflow is also a detail that one could consider in future work; in this chapter, the algorithm initially left it empty for simplicity.

There are different viable approaches for the estimation of bucket sizes. The use of an input size that is a multiple of the number of buckets in the examples in this chapter hides the simple approach used. An optimal approach is outside the scope of this thesis. However, future work should weigh the advantages of cache-aligning buckets, using smaller buckets, using larger buckets with a larger overflow buffer and the impact of various technical approaches in the literature on estimated bucket size.

Theorem 2. *Fast Radix (Algorithm 6) sorts an input correctly.*

Proof. Fast Radix deals the same digits in the same order as LSD radix sort in every pass. The used portion of an estimated bucket combined with any corresponding overflow bucket in Fast Radix has the same records as a corresponding exact bucket in LSD radix sort after every pass before the last pass. The exact buckets identified for the last pass in both Fast Radix and LSD radix sort are the same. LSD radix sort is known to sort input correctly. Therefore, Fast Radix must yield the same sorted result as LSD radix sort. ■

3.3 Diverting LSD Radix Sort

Diverting LSD radix sort estimates the number of most significant digits over which an LSD radix sort must run to achieve optimal diversion for the remaining partially ordered records. After an LSD radix sort runs on those high-order digits, the diverting LSD radix sort will perform a scan using the signature of the sorted most significant digits to determine if any large buckets remain. Small

Algorithm 7 Diverting MSD Radix Sort Revisited

Input: input, buf, digits
Output: sorted input on digits

```
 1: if end-start <= DIVERSION then
 2:     insertionSort(input)
 3:     if isOdd(length(digits)) then
 4:         memcopy(input, buf)
 5:     end if
 6:     return
 7: end if
 8: initialize()
    /* count and deal */
 9: c ← d ← digits[−1] // deal/count digit
10: count(input, buf, c)
11: deal(input, buf, d)
    /* recurse for subsequent passes */
12: if length(digits) == 1 then
13:     return
14: end if
15: for j ∈ buckets do
16:     recurse(j, input, digits[: −1])
17: end for
```

buckets are sorted in bulk using a simple, stable sort, most readily Insertion Sort. Any large buckets are sorted recursively with diverting LSD radix sort.

Diverting LSD radix sort differs from diverting MSD radix sort. It estimates the number of passes required to reach optimal diversion and then performs an LSD radix sort on those passes, recursing on remaining large buckets or diverting on runs of small buckets. On the other hand, diverting MSD Radix sort runs recursively on each level at an ever-increasing cost before diverting. Suppose the algorithms use the same diversion threshold. In that case, diverting LSD radix sort can accurately estimate the average number of passes for input with random records from a uniform distribution, which we will prove in Chapter 5.

3.3.1 Algorithm Details

Diverting LSD radix sort makes use of some labeled placeholders for specific sets of digits of interest and an array which acts as a buffer for use in any called sorting algorithms. In diverting LSD radix sort, the buffer `buf` is the same size as `input`. `top` and `bottom` represent subsets of `digits` with `top` holding most significant digits and `bottom` holding least significant digits.

To avoid clutter, we specify two structures as variables for the ranges discovered when scanning `input` after initial sorting on the `top` digits. The structures, `r1` and `r2` store the same type of data, but by convention `r1` will contain zero or more consecutive small buckets and `r2` will contain a single large bucket. The ranges themselves can be used as inputs when passed to other sorting algorithms and expose the inclusive `start` and exclusive `end` of their ranges using the dot notation (eg. `r1.start`).

Aside from arithmetic and set operations, the assignment operation ← and the array notation, there are several operations that we use without further definition. `initialize()` manages the creation of arrays and the zeroing of the count arrays. `estimateRelevant()` returns an estimate of the most significant digits from `digits` that are needed to partially sort `input` such that appropriate diversion can happen on the resulting records. `DivertingLSDRadixSort` is a recursive call to the algorithm described here. `LSDRadixSort()` is any LSD radix sort implementation. `hasMoreRecords()` takes `input` to determine if there are more records to scan. `scanForLargeBucket()` scans an `input` for buckets based on `top` digits as the input will be ordered on those digits and consecutive records with the same `top` digits will constitute a bucket, with the current bucket in the scan being recorded internally for the next call on `foundLargeBucket()`. `foundLargeBucket()` returns a range containing a single large bucket or null. `divert()` performs an in-place sort of the array of records passed in.

Algorithm 8 Diverting LSD Radix Sort

Input: input, buf, digits
Output: sorted input on digits

```
 1: initialize()
 2: top ← estimateRelevant(input, digits)
 3: bottom ← digits \ top
 4: LSDRadixSort(input, buf, top)
    /* scan for large buckets */
 5: if bottom.length > 0 then
 6:    r1.start ← 0
 7:    while hasMoreRecords(input) do
 8:       while (r2 = foundLargeBucket(input))! = null and hasMoreRecords(input) do
 9:          scanForLargeBucket(input, top)
10:       end while
          /* If a large bucket, r2, is found, r1 is the prior range containing 0..* small buckets. */
11:       if r2 != null then
12:          r1.end ← r2.start
13:       end if
          /* Otherwise, r1 is the range containing 0..* small buckets until the end of input. */
14:       if r2 == null then
15:          r1.end ← input.end
16:       end if
17:       divert(r1)
18:       if r2 != null then
19:          DivertingLSDRadixSort(r2, buf[r2.end :], bottom)
20:          r1.start ← r2.end
21:       end if
22:    end while
23: end if
```

3.3.2 Pseudo-code

Algorithm 8 begins by initializing `buf`, and `r1` on line 1. The `top` most significant digits required to partially sort `input` enough to benefit from a diverting sort are estimated on line 2. Any remaining digits from `digits` are assigned to `bottom` on line 3. On line 4 an LSD radix sort is performed on the `top` digits of `input`. If a check for bottom digits fails on line 5, then `input` is sorted. Otherwise a the scanning loop on lines 5 through 5 begins. The first potential set of small

buckets is set to start at the beginning of `input` on line 6 and if `input` is not empty, the first large bucket is searched for between lines 8 and 10. If a large bucket is found, then its beginning is used to terminate any range of small buckets in `r1` on line 12. If no large bucket was found then the list of small buckets is set to the end of `input` on line 15, and the loop will not continue when `hasMoreRecords` is next checked. In any event, the range of small buckets in `r1` is sorted using `divert` on line 17. If a large bucket was found, it will be in `r2` which will be recursively sorted with diverting LSD radix sort on line 19, passing a section of `buf` that corresponds to the position of `r2` in `input`. Once `r2` is sorted, a new potential range of small buckets is started in `r1` on line 20.

3.3.3 Diverting LSD Radix Sort Example

We can examine diverting LSD radix sort in Algorithm 8 using an example input with $N = 64$ records which have 16-bit keys and 2-bit digits (i.e. $M = 4$). We can specify a Diversion Threshold (DT) of 4. We compare against diverting MSD radix sort and diverting LSD radix sort to highlighted how similarly passes are performed before diversion.

After a given number of passes p we can describe the remaining buckets using sets. We consider the predicate S^p testing that an element is a set of zero or more contiguous small buckets bound by the input boundaries or a large bucket on the left and by a large bucket on the right, the predicate B^p testing for a single large bucket and the set A^p being all buckets. We also consider the predicate C testing that an element $s^p \in S^p(s^p)$ and an element $b^p \in B^p(b^p)$ represent contiguous elements with any buckets in s^p occurring prior to the bucket in b^p. We then consider the set $s^{p\prime}$ representing the zero or more contiguous small buckets bound by the input boundaries or a large bucket on the left and by the input boundary on the right. We can then say that the union of all $\{c^p \in C^p(c^p)\}$ and $s^{p\prime}$ is A^p.

A traditional diverting MSD radix sort would take the records represented in Figure 17 and deals them into four buckets. These four buckets, shown in Figure 18 as the columns grouped by

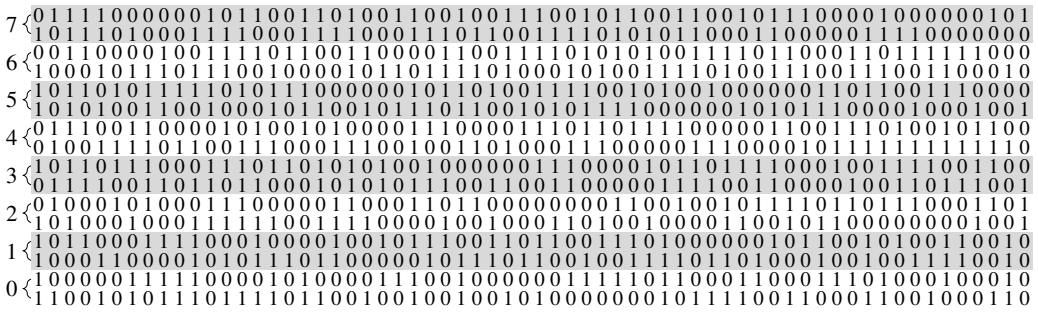

Figure 17: 64 unordered random 16-bit integers, each column representing one integer and each row representing corresponding bits. Most significant bits are at the top. Bits are grouped by 2-bits to form digits, indicated by alternating shading, and each digit is labeled on the left with its index.

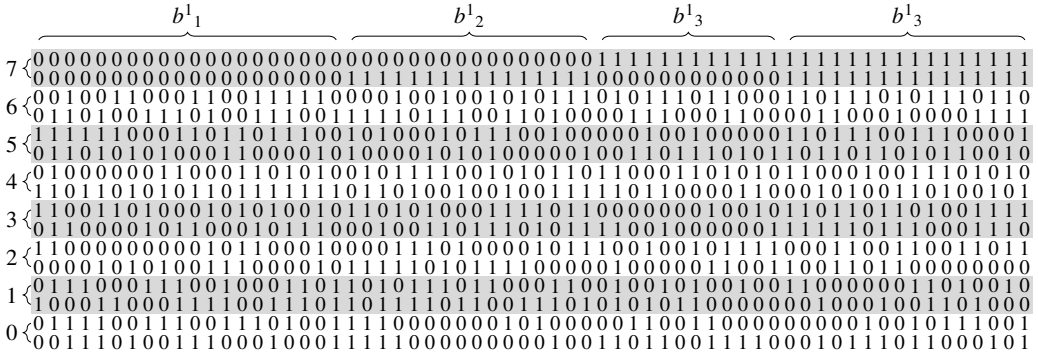

Figure 18: The 64 random 16-bit integers from Figure 17 ordered by their highest digit. Grouping along the top indicates *buckets* with more than $DT = 4$ records. As expected, there are four large buckets.

braces along the top, would each have the diverting MSD radix sort recursively called on them. This pass would, in turn, yield the buckets shown in Figure 19, where only the larger sub-buckets identified would warrant an additional recursive call of diverting MSD radix sort; the algorithm would individually sort the other buckets using a diversion algorithm. The largest remaining buckets shown in Figure 20 fall below the diversion threshold, and thus each of those buckets is sorted with the diversion algorithm.

Diverting LSD radix sort differs from diverting MSD radix sort. It takes an initial LSD approach instead of a recursive MSD approach, reducing the extra cost of count initialization and prefix summing on the initial top passes.

In this example, Diverting LSD radix sort could return 2 for $estimateRelevant$ (line 2). After

Fast Radix ran on the input from Figure 17 (line 4), the resulting set of records would be those in Figure 19. Each loop through through lines 7 and 22 would yield either `r1` containing buckets s^2 and `r2` containing bucket b^2 if a large bucket were found, or `r1` containing buckets $s^{2\prime}$.

The pairs of small and large buckets and small trailing buckets are shown in each of the partially sorted cases for this example, with empty buckets omitted. In Figure 18 there are only four sets of large buckets for the one sorted digit. Figure 19 shows the same buckets for the most significant digit along the top, and immediately below that shows the five pairs of small and large buckets, with a trailing set of small buckets, for the two sorted digits. In Figure 20, the sets of buckets for the top two digits are show as in Figure 19 with the set of buckets, being solely $s^{3\prime}$ show for three digits.

The entire input would be scanned for small-bucket runs and large buckets (lines 5–23). This scanning is broken down into scans for large buckets (lines 8–10) where diverting LSD radix sort would examine the top bits (labeled 7 and 6 on the left of the Figures) until a bucket with more than DT records is found. When a large bucket is found it is identified as the range $r2$ (line 8), and anything unprocessed prior to that bucket must therefore be a run of smaller buckets identified as $r1$ (line 12). Whenever the end of the input is reached without finding a new large bucket, the run at the end must also contain only smaller buckets (line 15).

The diverting LSD radix sort scan for small-bucket runs and large buckets (lines 5–23) after two digits yields $\{s^2{}_0, b^2{}_0, s^2{}_1, b^2{}_1, s^2{}_2, b^2{}_2, s^2{}_3, b^2{}_3, s^2{}_4, b^2{}_4, s^{3\prime}\}$ with $s^2{}_1$ and $s^2{}_2$ being empty sets and not shown in Figure 19. Diverting LSD radix sort would recursively call itself on line 19 five times, on each large bucket in $\{b^2{}_0, b^2{}_1, b^2{}_2, b^2{}_3, b^2{}_4\}$, and would called the diversion algorithm six times, on each set of small buckets in $\{s^2{}_0, s^2{}_1, s^2{}_2, s^2{}_3, s^2{}_4, s^{2\prime}\}$. As Figure 20 shows that the third pass is all small buckets, it can be inferred that the recursive call would sort on only one additional digit for those records before calling diversion on a single run of small buckets.

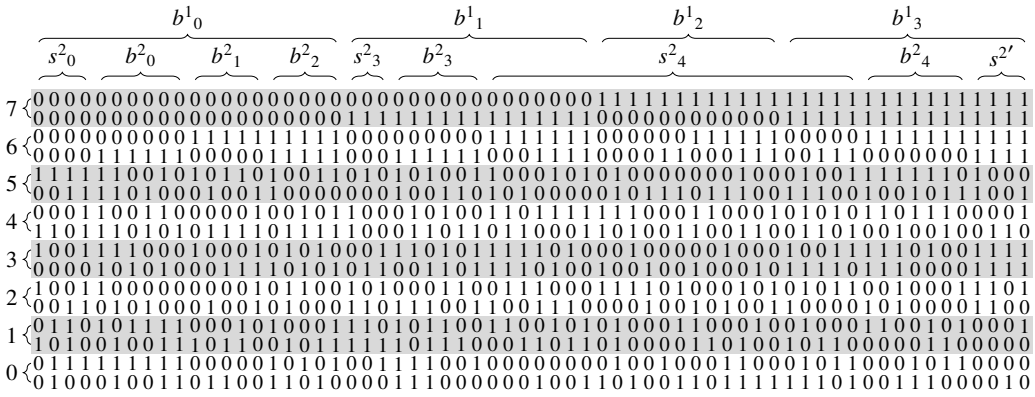

Figure 19: The 64 random 16-bit integers from Figure 17 ordered by their highest two digits. Braces along the top represent the heirarchical *tree* of large buckets. Braces along the bottom represent sequences of small buckets, labeled sequentially by letter and sequence length.

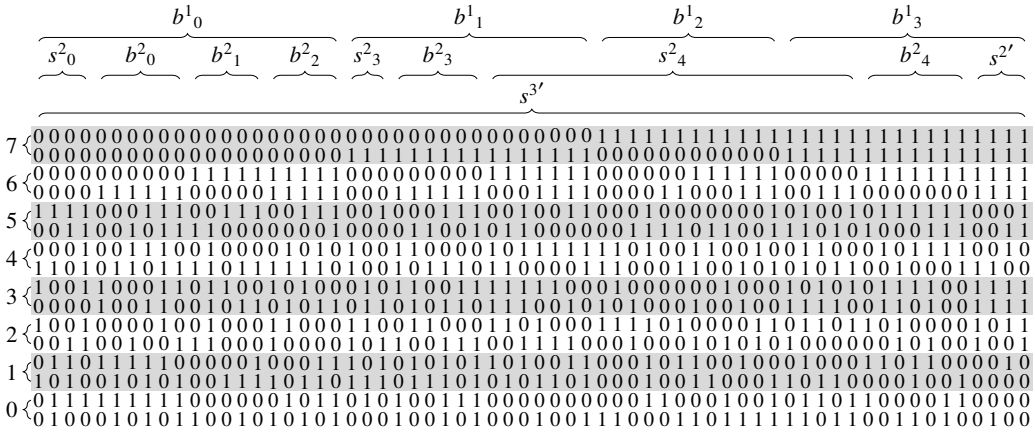

Figure 20: The 64 random 16-bit integers from Figure 17 ordered by their highest three digits. Braces along the top represent the heirarchical *tree* of large buckets for the first two digits, and small buckets bigger than 1 for the third digit. By the third digit, there are no buckets with more than $DT = 4$ records.

3.3.4 Discussion

While a developer must take some care to guarantee that the diversion algorithm's approaches and the LSD radix sort ensure that the ordered input ends in the correct original location, there are few challenging technical pitfalls.

As described in the discussion section for Fast Radix, digits' parity can be an issue when calling

other sorting algorithms that perform passes by digit and do not expect an odd number of digits. Managing this, or even benefiting from it while incorporating technical optimization outside the scope of this thesis, is an avenue for future research. One can expect that `top` will be an odd number of digits. On average, when recursion is required, the `top` in that recursive call will likely contain a single digit, as will be seen in Chapter 5.

While this thesis focuses on proving that diverting LSD can perform the same number of passes as diverting MSD given a common diversion threshold, that may not always be desirable. As the choice of a diversion threshold and an algorithm is dependent on the costs related to continuing with the existing algorithm, the criteria will differ between diverting LSD, and MSD radix sorts as diverting MSD radix sorts become more expensive with each pass. Future work examining diversion thresholds and diversion algorithms combined with future work on performing more cache-sensitive passes in LSD radix sort are one of the most promising future avenues for this research. The opportunities of that avenue of research belie the simplistic (yet still practical) approach used here.

At a higher level, one can think of scanning as favouring left-to-right approaches that ignore recording of supplemental bucket information as is done in most diverting MSD radix sorts. In turn, this favours diversion algorithms like Insertion Sort that allow empty buckets to be collapsed at no cost and consecutive small buckets to be processed together. This thesis has used such approaches, but there may be other approaches to consider in future work that examine less strictly left-to-write approaches or which otherwise improve the characteristics of either scanning or the incorporation of diversion algorithms with this process.

Theorem 3. *Diverting LSD radix sort (Algorithm 8) sorts input correctly.*

Proof. Diverting LSD radix sort sorts on some number of most significant digits using LSD radix sort, which is known to sort correctly. The sorting on those most significant digits implicitly partitions the input into disjoint ordered subsets of the input, where records in each subset are larger than all records in subsets before them and smaller than all records in subsets after them. One can

sort any subset that is smaller than a given diversion threshold with a diversion algorithm known to sort correctly. One can recursively sort any other subset by calling diverting LSD radix sort on the remaining digits not already sorted on. Records sorted on all digits are sorted. Therefore diverting LSD radix sort must sort input correctly. ■

3.4 Diverting Fast Radix

Diverting Fast Radix differs from diverting LSD radix sort only in the use of Fast Radix instead of LSD Radix Sort, shown in line 4 of Algorithm 9.

Theorem 4. *Diverting Fast Radix (Algorithm 9) sorts input correctly.*

Proof. As Diverting Fast Radix is just diverting LSD radix sort using Fast Radix, which is known to sort correctly, it must also sort correctly. ■

3.5 Summary of Algorithms

We have shown our four algorithms. Our Simple Fast Radix algorithm (initially Fast Radix) from 2016 omitted the first counting pass in place of estimation. Our subsequent Fast Radix algorithm omitted all but the last counting pass. Our diverting LSD radix sort algorithm showed that LSD radix sorts could divert as effectively as MSD radix sorts. Our Diverting Fast Radix showed the combination of Fast Radix and diverting LSD radix sort that is the culmination of this research.

The first two algorithms identify that one can almost entirely avoid counting. The bulk of the gains come from skipping the first counting pass as presented in Simple Fast Radix, but with additional gains commensurate to the number of additional passes where Fast Radix omits counting.

Algorithm 9 Diverting Fast Radix

Input: input, buf, digits
Output: sorted input on digits

```
1: initialize()
2: top ← estimateRelevant(input, digits)
3: bottom ← digits \ top
4: FastRadix(input, buf, top)
   /* scan for large buckets */
5: if bottom.length > 0 then
6:    r1.start ← 0
7:    while hasMoreRecords(input) do
8:       while (r2 = foundLargeBucket(input))! = null and hasMoreRecords(input) do
9:          scanForLargeBucket(input, top)
10:      end while
         /* If a large bucket, r2, is found, r1 is the prior range containing 0..* small buckets. */
11:      if r2 != null then
12:         r1.end ← r2.start
13:      end if
         /* Otherwise, r1 is the range containing 0..* small buckets until the end of input. */
14:      if r2 == null then
15:         r1.end ← input.end
16:      end if
17:      divert(r1)
18:      if r2 != null then
19:         DivertingFastRadix(r2, buf[r2.end :], bottom)
20:         r1.start ← r2.end
21:      end if
22:   end while
23: end if
```

The last two algorithms show that one can apply diversion to LSD radix sorts, with Diverting Fast Radix representing the combination of all improvements covered in this chapter.

Chapter 4

Experimental Results

In this chapter, we will present the methodology used to evaluate our algorithms' implementations. The methodology will focus on what inputs were used (data sets and distributions), what architectures were used, and what was measured in various experiments. Over time, the focus of experiments reported here changed. While the reported results' applicability warrants their inclusion, individual sections provide context to situate them better.

Work before 2015 focused on Java in Windows desktop environments using positive 32-bit integers (effectively 31-bit integers) with synthetic data sets pulled randomly from uniform and normal distributions. Java simplified development, but its use limited the number of competitive algorithms that considered technical optimizations beyond Java's abstractions. Sufficient memory for casual testing in desktop environments, given that Java tends to require a great deal, limited viable input sizes to approximately 10^8 records.

At the end of 2014, given feedback from reviewers and the thesis committee, the focus shifted to C++ implementations and 64-bit unsigned integers on Linux. *Real* data sets were researched to provide a better baseline for comparison against other algorithms, with the large data sets exceeding 10^9 records needing sorting. In 2016, results using a C++ implementation of a Simple Fast Radix (replacing just the first counting pass) were published in Thiel et al. [2016]. In that publication, Simple Fast Radix replaced the PBBS radix sort implementation in GraphChi[Kyrola et al., 2012], a software that performed Big Data graph processing. Thiel et al. [2016] reported timing improvements and the overflow.

By the end of 2017, pre-prints of Kokot et al. [2018] were available, and the authors were responsive and generous in sharing their implementation. Their technical improvements built upon their previous work in Kokot et al. [2017], which in turn built on Satish et al. [2009], to quickly sort large input. With a faster competitor on the horizon and concern regarding what constituted a significant performance improvement in the field, the focus shifted to extending this research's algorithmic improvements.

The most apparent adjustment was to skip counting on all but the last pass, the algorithm that we call Fast Radix (not shown here), but this did not improve on Simple Fast Radix enough to be competitive against RADULS2[Kokot et al., 2018]. Kokot et al. [2017, 2018] credited the use of diversion in their MSD radix sort as providing much of the gains over the LSD radix sort approach in Satish et al. [2009]. We considered the novel approach of diverting in LSD radix sorts in place of relying on technical improvements relating to cache-optimization that were well-researched elsewhere.

Kokot et al. [2017] focused on uniform random distributions, and all radix sort papers reviewed used this distribution, sometimes exclusively. This distribution's frequent use led us to adopt 64-bit unsigned integers with uniformly random distributions for this research. on As the implementation in Thiel et al. [2016] showed little impact from differences between real data sets and random inputs drawn from uniform distributions, after accounting for inexpensive approaches used to filter for structure (e.g. when high-order bytes were unused in the social media graphs where `user ids` started at 1 and counted up), our focus changed to

This chapter's remainder is broken into sections presenting the general methodology for the experiments in this chapter, noting how it changed over time. The remaining sections present results for a Java implementation of Simple Fast Radix, the C++ implementation of Simple Fast Radix used in Thiel et al. [2016], a C++ implementation of Fast Radix to show the progression of the improvements, and the C++ implementation of Diverting Fast Radix. Each section summarizes the algorithms compared, gives any context for the experiment on those algorithms, describes the setup,

and shows results with a brief discussion.

4.1 Methodology

Initial experiments involved a Java implementation of Simple Fast Radix on a Windows desktop with other things running. Initial experiments varied the input size in steps of 10^6 until 10^8, with ten runs for each algorithm at each step. As results were consistent, subsequent experiments recorded the average of two tests over the same range of input sizes, shown here. Running time was taken from `System.nanoTime()` and we discarded the least significant three digits of precision.

The 2016 experiments used a C++ implementation of Simple Fast Radix[Thiel et al., 2016] on a shared computational resource at Concordia University. All experiments ran via a script, and the results shown here are from a consistent set of the results of those runs; the impact of contention with other jobs was not subtle and recorded runs were from off-hours and repeatable on a "quiet" server. These experiments used GraphChi[Kyrola et al., 2012] to generate the records from the imported graphs, which GraphChi then sorted. GraphChi's timing reporting system tracked sorting time.

In 2017, to avoid contention with other user jobs, hardware funded through Hexagram's CFI grant and generously made available exclusively for this research was used. This server was a bare-metal system, with no windowed environment, no other users, and minimal services. We performed all subsequent experiments in this chapter on this environment. As test runs were relatively consistent in this environment, experiments were run once and recorded. As all algorithms were C++ code, we took a consistent approach to generate inputs from uniform distributions. Experiments used the `chrono` library for timing on standalone runs.

4.2 Java Simple Fast Radix

Simple Fast Radix's initial implementation capitalized on a revelation regarding inputs' properties in general: the least significant bits do not matter. MacLaren noted a similar thing when writing that "... much time will be wasted examining low order digits, and these seldom affect the order of the records." [MacLaren, 1966]. Our analysis in Chapter 5 bears this out, as do the results here comparing Simple Fast Radix to the default `Arrays.sort`, a standard implementation of Quicksort and a generic LSD radix sort.

Here we compare the results of timing experiments on the Java implementation of Simple Fast Radix, the default `Arrays.sort`, a standard implementation of Quicksort and a generic LSD radix sort.

4.2.1 Setup

The hardware used was a 3.4GHz Intel Core i7-2600 (4-core) with 4x4GB DDR3-1066 RAM running Windows 7. The script which performed the experiments automatically generated inputs up to 10^8 at intervals of 10^6 and recorded the average of two runs for each step. uniform and normal distributions were generated using `java.util.Random`. The two uniform distributions used were on unsigned integers up to $2^{31} - 1$ and up to $2^{24} - 1$. The two normal distributions used were unsigned integers with a mean of 2^{30} and standard deviation of $\frac{2^{30}}{3}$ and a mean of 2^{30} and standard deviation of $2^{24} - 1$

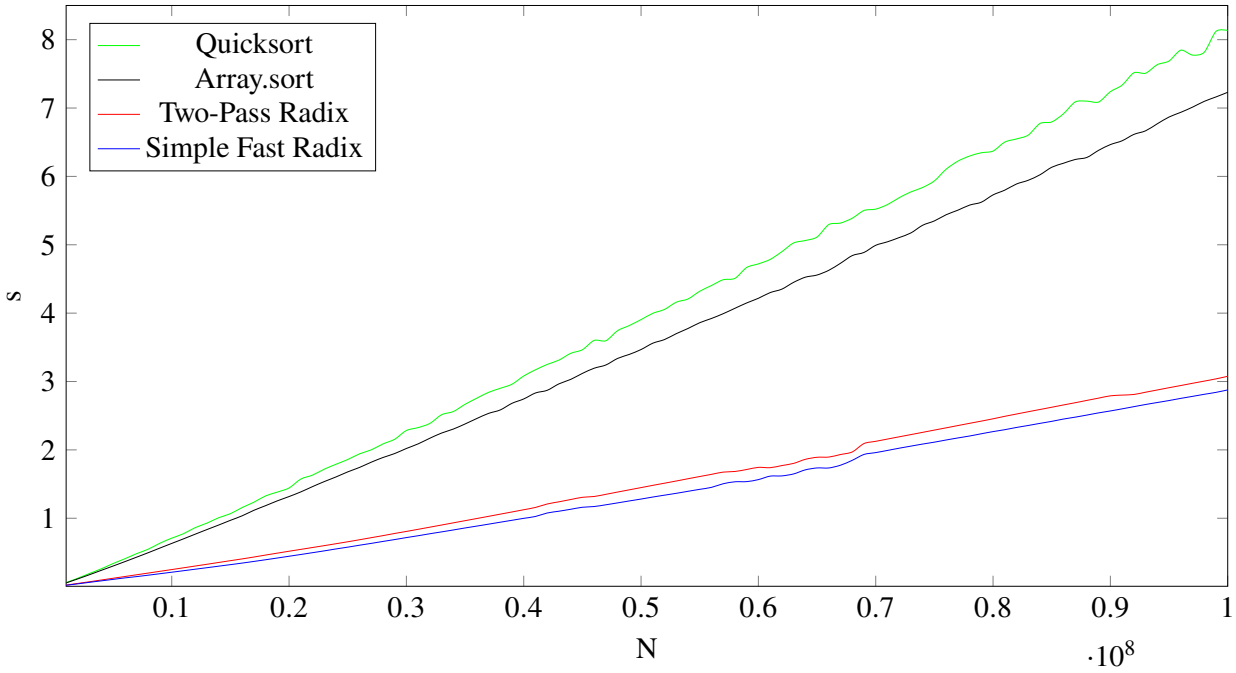

Figure 21: Time of Sorting Algorithm vs. Input Size for a Uniform Distribution from 0 to $(2^{31} - 1)$.

4.2.2 Results

Figure 21 shows the first uniform distribution between 1–$2^{31} - 1$ (the full range of positive `int` in Java), and Figure 22 shows the second uniform distribution between 1–$2^{24} - 1$. Figure 23 shows the wider normal distribution with a mean value of 2^{30} and ar standard deviation of $\frac{1}{3}$ of the mean. Figure 24 shows the results of a much narrower distribution with a standard deviation of $2^{24} - 1$ around the same mean.

4.2.3 Discussion

The main takeaway from these results is that skipping the initial counting pass reliably reduces an LSD radix sort's overall time. The use of normal distributions had a minimal impact, even narrow ones (given that the skipped pass was the least significant digit). It is also visible that radix sorts perform much better than the standard sorting, at least in Java. In the two constrained distributions

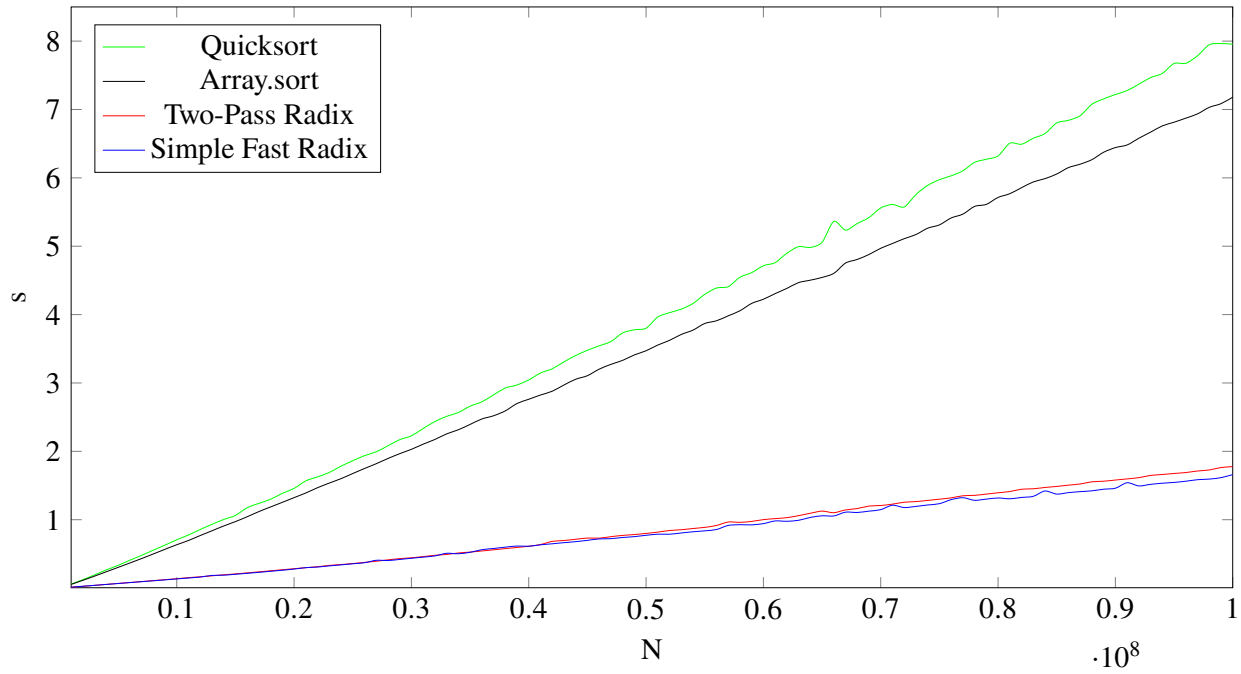

Figure 22: Time of Sorting Algorithm vs. Input Size for a Uniform Distribution from 0 to $(2^{24} - 1)$.

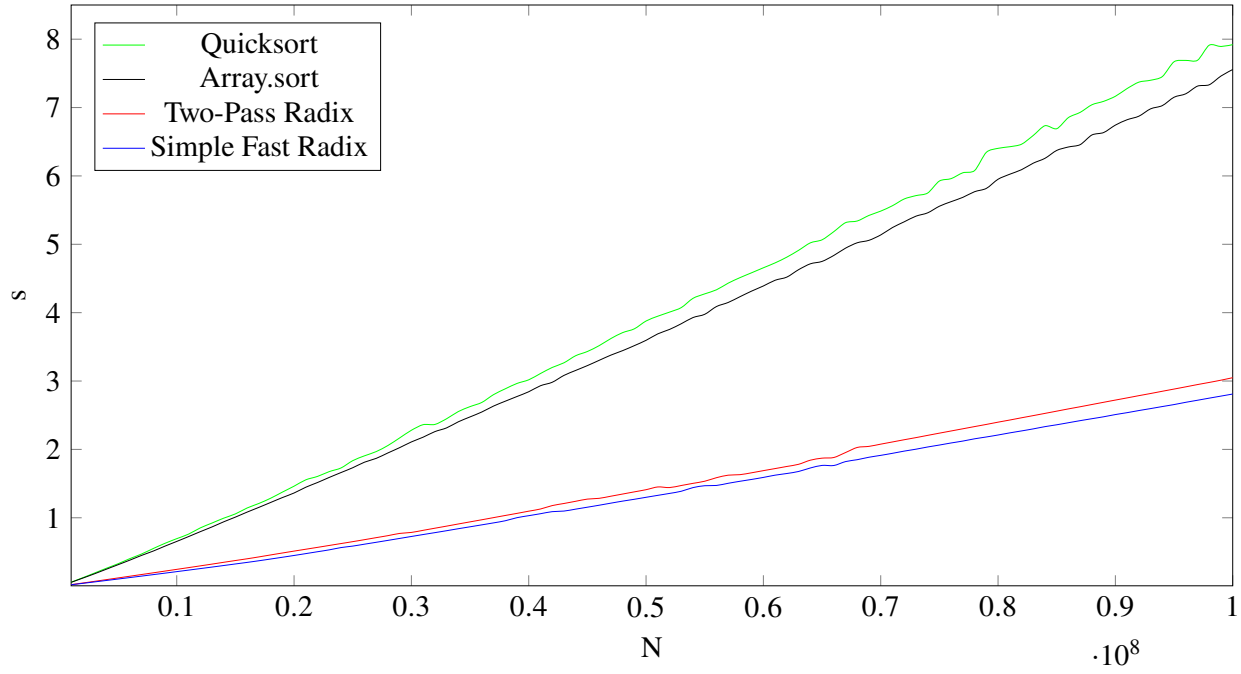

Figure 23: Time of Sorting Algorithm vs. Input Size for a Normal Distribution with mean of 2^{30} and stdev of $\frac{2^{30}}{3}$.

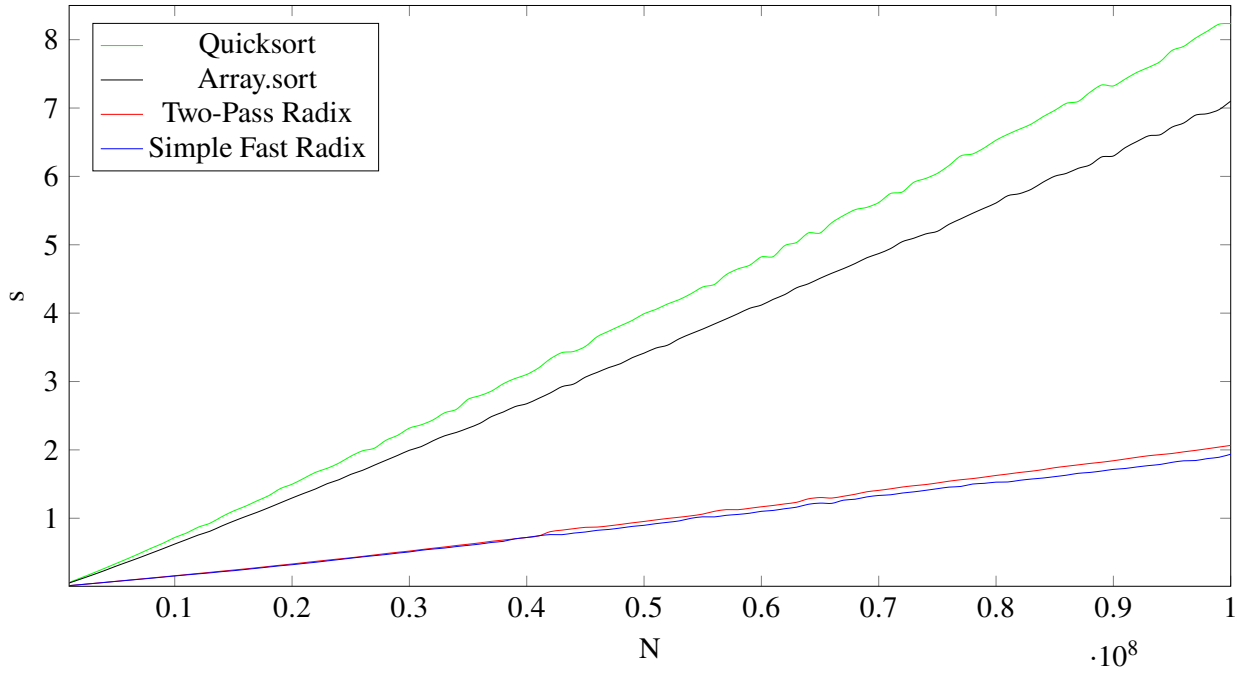

Figure 24: Time of Sorting Algorithm vs. Input Size for a Normal Distribution with mean of 2^{30} and stdev of $2^{24} - 1$.

in Figure 22 and Figure 24, we see that dealing into very few buckets in the most significant digit pass significantly reduced the overall time taken for radix sorts.

4.3 C++ Simple Fast Radix

In our previous work, we demonstrated a significant improvement in graph operations using GraphChi[Kyrola et al., 2012] applied to traditional social media big data sets[Thiel et al., 2016]. The only change in the GraphChi software was replacing the PBBS Radix Sort[Shun et al., 2012] implementation used in the initial work with Simple Fast Radix[Thiel et al., 2016].

Graph Name	Vertices	Edges	Citation
google	900K	5M	Leskovec et al. [2008]
live-journal	4.8M	69M	Backstrom et al. [2006]
hollywood	1.1M	113M	Boldi and Vigna [2004], Boldi et al. [2011]
twitter-2010	42M	1.5B	Kwak et al. [2010]
friendster	66M	1.8B	Yang and Leskovec [2012]
uk-union	134M	5.5B	Boldi et al. [2008]

Table 3: Summary of Big Data Graphs
This table shows *Vertices* and *Edges* in each of the graphs used in Thiel et al. [2016].

Graph Name	Vertices	Edges	GraphChi	Fast Radix	Δ Sort	overflow
google	900K	5M	0.15s	0.077s	99%	0.65%
live-journal	4.8M	69M	5.4s	5.4s	0%	0.72%
hollywood	1.1M	113M	7.2s	4.9s	47%	0.50%
twitter-2010	42M	1.5B	128s	106s	21%	1.54%
friendster	66M	1.8B	165s	133s	24%	0.12%
uk-union	134M	5.5B	469s	329s	43%	0.49%

Table 4: Summary of Fast Radix improvements in GraphChi over PBBS Radix Sort.
This table shows the number of *Vertices* and *Edges* for each graph. The *GraphChi* and *Fast Radix* columns show the total sort times for GraphChi sharding using PBBS radix sort and Fast Radix, respectively; Δ *Sort* is the percentage improvement. The *overflow* column shows the percentage of elements that overflow during Fast Radix. Taken from Thiel et al. [2016] with minor formatting adjustments.

4.3.1 Real Data Setup

The experiment recorded the timings reported by GraphChi[Kyrola et al., 2012] when sorting with a C++ implementation of Simple Fast Radix and the default PBBS radix sort[Shun et al., 2012] which shipped with GraphChi. We ran the experiment on a 2.60GHz Intel Xeon E5-2697 (4 x 14-core) with 768GB RAM running Linux. All inputs were generated internally by GraphChi based on the identified Big Data graphs shown in Table 3.

Graph Name	Vertices	Edges	GraphChi	Fast Radix	Δ Time
google	900K	5M	1.7s	1.5s	14%
live-journal	4.8M	69M	32s	32s	0%
hollywood	1.1M	113M	45s	38s	18%
twitter-2010	42M	1.5B	759s	693s	9.4%
friendster	66M	1.8B	1022s	921s	11%
uk-union	134M	5.5B	2549s	2135s	19%

Table 5: Pre-processor timing changes on experiment graphs.
This table shows he number of *Vertices* and *Edges* for each graph. The *GraphChi* and *Fast Radix* columns show the total pre-processing times for GraphChi using PBBS radix sort and Fast Radix, respectively; Δ *Time* is the percentage improvement. Taken from Thiel et al. [2016] with minor formatting adjustments.

4.3.2 Real Data Results

Data sets were selected based on those reported initially in [Kyrola et al., 2012], with additional similarly sized data sets replacing those that were not readily available, and in the case of the smallest data set, filling out what seemed like a gap in the range. Table 4 shows that Edges range from 5×10^6 to 5.5×10^9, with edges representing the size of inputs being sorted during GraphChi runs. Sorted data included the key to sort on, but GraphChi also included payload data in the records.

We performed single timing runs on each graph for each algorithm using an automated script. Separate runs with instrumented code allowed for the reporting of percentage overflow in Table 4 without interfering with timings. In Table 5 the overall pre-processing times were reported for the same graph runs, demonstrating the impact of improving sorting as a component of another software.

4.3.3 Discussion

As discussed in Thiel et al. [2016], the results showed that Simple Fast Radix provided a notable improvement. In a few cases, there was unexpectedly no improvement, and in general, both Simple Fast Radix and PBBS Radix sort slowed down significantly with the larger data sets.

4.3.4 Synthetic Data Setup

These experiments were run on a 2.90GHz Intel Xeon E5-2690 (8-core) with 16GB RDIMMS-1600MHz RAM with a 512GB SSD running Linux. We set this server up explicitly to support this research, and we did not allow other users to log onto it. As results on the server were consistent, we ran experiments recorded here once. Random inputs from a uniform distribution were generated using the standard `random` library with `std::mt19937` as a generator and `uniform_int_distribution` as a distribution and using the `chrono` library for timing.

The experiments with synthetic data differed in several ways besides the source of data. Both Simple Fast Radix[Thiel et al., 2016] and PBBS Radix[Shun et al., 2012]) implementations included optimizations to take advantage of extra information provided by GraphChi to allow for the sort to skip some unused passes. The records in GraphChi consisted of a combination of source and destination data, so GraphChi performed sorting on one or the other part as key but moved the entire record around; the experiments on synthetic data used a single key for the entire record. The GraphChi implementations used signed long integers, and while Simple Fast Radix was adjusted to accommodate unsigned long integers, PBBS Radix sort needed a more significant adjustment. Hence, its test was still on 63-bit unsigned integers, though the impact here was likely minor.

4.3.5 Synthetic Data Results

In Figure 25 we show the performance of Simple Fast Radix, PBBS Radix and std::sort on 64-bit integers generated from a uniform distribution, with the raw data in Table 6.

N	Simple Fast Radix	PBBS Radix	std::sort
100	36μs	20μs	9μs
1000	99μs	66μs	99μs
10000	605μs	701μs	1297μs
100000	7520μs	8667μs	14260μs
1000000	55286μs	55024μs	99030μs
10000000	968203μs	996272μs	762735μs
100000000	8733755μs	9423967μs	8109444μs
1000000000	123499985μs	135640439μs	91525398μs

Table 6: Timing comparison of Simple Fast Radix[Thiel et al., 2016], PBBS Radix[Shun et al., 2012] and std::sort on a log scale using random 64-bit unsigned integers from a uniform distribution, with PBBS Radix sort using 63-bit unsigned integers.

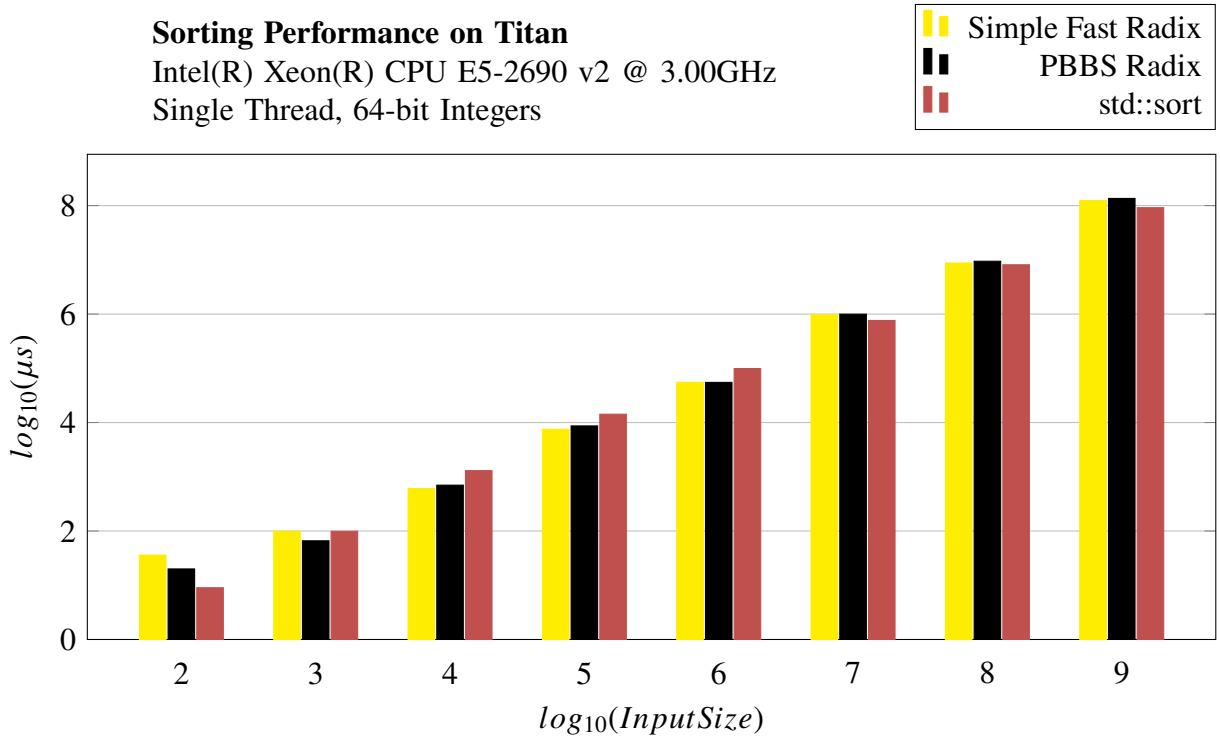

Figure 25: Timing comparison of Simple Fast Radix[Thiel et al., 2016], PBBS Radix[Shun et al., 2012] and std::sort on a log scale using random 64-bit unsigned integers from a uniform distribution, with PBBS Radix sort using 63-bit unsigned integers.

4.3.6 Discussion

We posit that the improved results were due to having almost no overflow when estimating bucket sizes while having saved the traditional first pass through the data, which would have counted bucket sizes exactly. Real data showed slightly higher amounts of overflow than predicted for random data from a uniform distribution. However, it was still less than 1% in most cases, as shown in Table 4. Synthetic data, as expected, matched the predicted results.

Simple Fast Radix's results, now programmed in C++, support the initial experimental results, which showed an improvement when skipping the first counting pass. As noted in Table 4, Simple Fast Radix generally performed better by upwards of 20%, with similar performance for the `live-journal` data set (which was repeatable) and was not related to overflow processing. While overflow was more than it might be for random inputs from a uniform distribution, it was still minimal. Table 5 shows that an improvement in sorting can yield a noticeable improvement in overall application performance, as the changes in sorting time yielded between 10 and 20% improvements in the pre-processing step of GraphChi, save with the `live-journal` data set where performance remained the same. "Given that the pre-processing time often exceeds the actual graph processing time, this improvement has practical benefit"[Thiel et al., 2016].

The synthetic experiments showed more muted results. There was a mixed improvement on the smaller data sets, near-identical results on the 10^6 and 10^7 data sets and a modest 10% improvement on the 10^8 and 10^9 data sets, compared to the more considerable improvements noted in Thiel et al. [2016]. The improvement over PBBS Radix sort is still notable, and showing these experiments using synthetic data allows a common baseline for comparison with the algorithms presented later in this chapter. The use of `std::sort` in these timing results reinforces that baseline as the same results for `std::sort` will be used when showing the later results in this chapter. The comparison with `std::sort` also highlights the cache-performance issue that begins to impact results with radix sorts at some point, in this case after inputs of 10^6.

N	Simple Fast Radix	Fast Radix	Diverting Fast Radix
100	400%	2800%	800%
1000	100%	367%	107%
10000	47%	74%	28%
100000	53%	53%	27%
1000000	56%	61%	31%
10000000	127%	83%	53%
100000000	108%	67%	52%
1000000000	135%	67%	47%

Table 7: Timing comparison of Simple Fast Radix, Fast Radix, Diverting Fast Radix, normalized to `std::sort`. Synthetic data generated from random 64-bit unsigned integers from a uniform distribution.

4.4 Fast Radix

Fast Radix is an intermediate step between Simple Fast Radix and Diverting Fast Radix. We include results for an implementation of the Fast Radix algorithm presented in this thesis, situated with respect to the other presented Fast Radix algorithms and showing the progressive improvement over the course of this research. Table 7 shows that Fast Radix performed between 50 to 100% faster than Simple Fast Radix on $N \geq 10^7$, in part due to savings from reduced counting effort. As expected, Diverting Fast Radix performs better than Fast Radix in all cases.

4.5 Diverting Fast Radix

Results from Diverting Fast Radix in Figure 26 show that our implementation is faster on all save the smallest input sizes up until input sizes on the order of one million (10^6), without any of the cache optimizations that support RADULS2. Even after cache-related performance impacts of Diverting Fast Radix, it remains competitive with RADULS2 and is faster than std::sort. We also present results for Ska SortSkarupke [2017], a diverting MSD Radix sort with publicly available (and easy to use) source code that does not use the cache-optimization found in RADULS2.

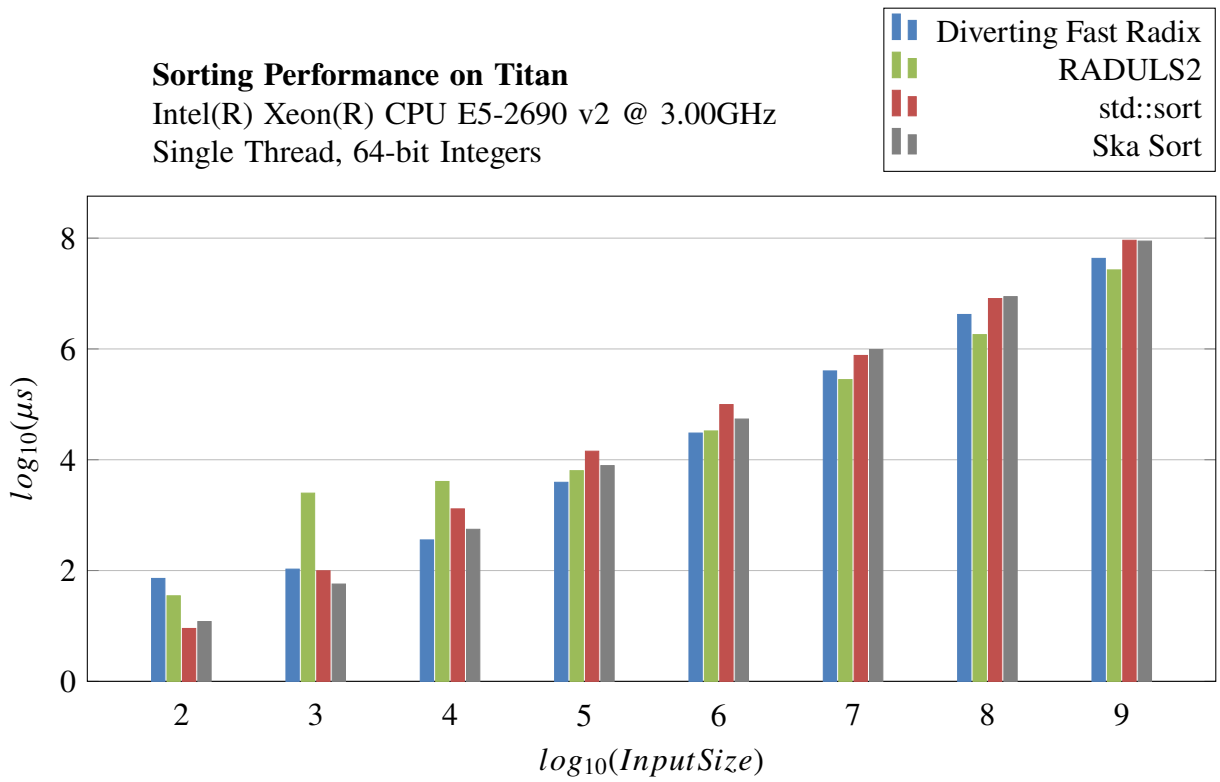

Figure 26: Timing comparison of Diverting Fast Radix, RADULS2[Kokot et al., 2017], std::sort, Ska Sort[Skarupke, 2017] on a log scale using random 64-bit unsigned integers from a uniform distribution.

4.5.1 Setup

This experimental setup is the same as in Section 4.3.4, using the Intel Xeon E5-2690, running experiments once and using standard libraries for random number generation and timing.

4.5.2 Results

In Figure 26 we show the performance of Diverting Fast Radix, RADULS2, std::sort and Ska Sorton 64-bit integers generated from a uniform distribution, with the raw data in Table 8.

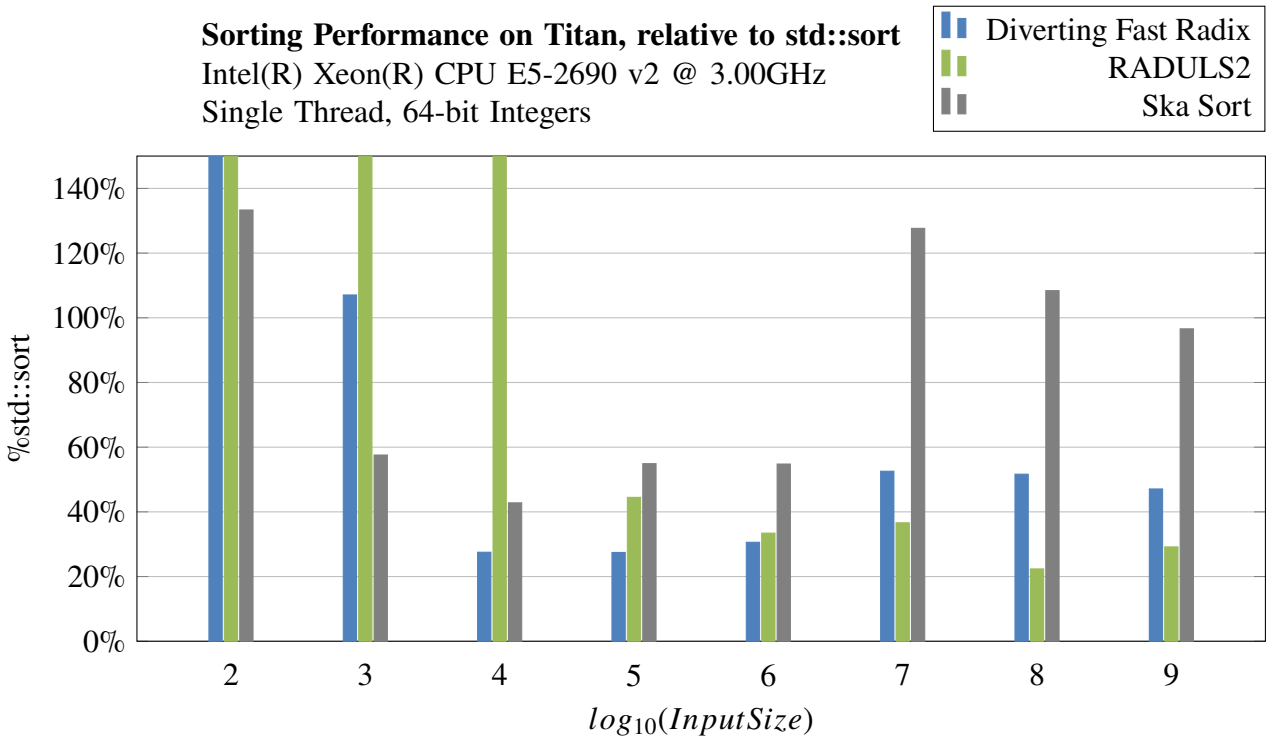

Figure 27: Timing comparison of Diverting Fast Radix, RADULS2[Kokot et al., 2017], std::sort, Ska Sort[Skarupke, 2017] relative to std::sort using random 64-bit unsigned integers from a uniform distribution.

N	Diverting Fast Radix	RADULS2	std::sort	Ska Sort
100	72μs	35μs	9μs	12μs
1000	106μs	2495μs	99μs	57μs
10000	357μs	4050μs	1297μs	555μs
100000	3915μs	6344μs	14260μs	7829μs
1000000	30305μs	33106μs	99030μs	54260μs
10000000	400963μs	279323μs	762735μs	973693μs
100000000	4188510μs	1813349μs	8109444μs	8789985μs
1000000000	43092593μs	26701920μs	91525398μs	88402766μs

Table 8: Timing comparison of Diverting Fast Radix, RADULS2[Kokot et al., 2017], std::sort, Ska Sort[Skarupke, 2017] using random 64-bit unsigned integers from a uniform distribution.

N	Diverting Fast Radix	RADULS2	Ska Sort
100	800.00%	388.89%	133.33%
1000	107.07%	2520.20%	57.58%
10000	27.53%	312.26%	42.79%
100000	27.45%	44.49%	54.90%
1000000	30.60%	33.43%	54.79%
10000000	52.57%	36.62%	127.66%
100000000	51.65%	22.36%	108.39%
1000000000	47.08%	29.17%	96.59%

Table 9: Timing comparison of Diverting Fast Radix, RADULS2[Kokot et al., 2017]Ska Sort[Skarupke, 2017], normalized to std::sort, using random 64-bit unsigned integers from a uniform distribution.

4.5.3 Discussion

Diverting Fast Radix, `std::sort` and Ska Sort used the usual random input generation and timing. On the other hand, RADULS2[Kokot et al., 2018] generated random numbers and did timing on their own, though the implementation of their approach was consistent with our methods. RADULS2 also prepared, and memory aligned the standard additional buffers. They ensured that the input array was also memory aligned and *touched* the data in arrays to prime the system before timing commenced. This difference had a material effect on timing. Developers might not traditionally do some of these things or do them inside the sorting algorithm implementations where timing experiments would count them. Their documentation provided reasoning for these approaches and indicated how some aspects were necessary for their implementation's technical optimizations. As such, we used the timings reported by their implementation.

Figure 26 is a log-scaled graph of the timing results from 10^2–10^9 scaling by 10 with each step. We performed each experiment once. Timings for RADULS2, in particular, varied from run-to-run, but the log graph elides this degree of sensitivity to show a general trend. Somewhere shortly after input sizes pass 1000, Diverting Fast Radix is markedly faster than the other considered algorithms.

Table 8 is the underlying data for Figure 26. The takeaway here is that despite the slowdowns

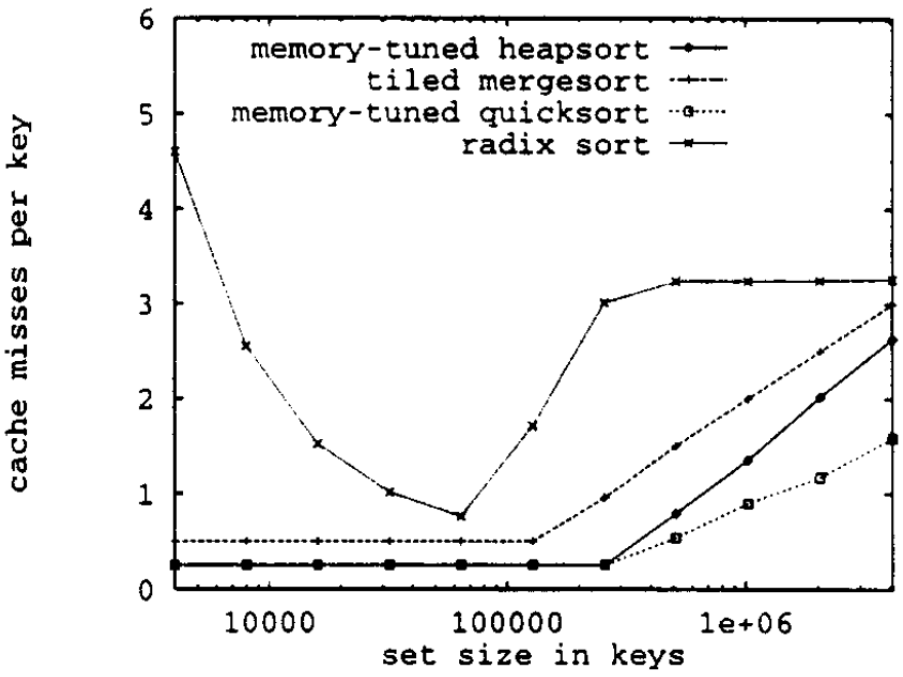

Figure 28: The impact of cache misses as a function of input size[LaMarca and Ladner, 1999, p. 33].

due to cache misses, `std::sort`'s non-linear complexity allows Diverting Fast Radix to run more than twice as fast at inputs of size 10^9. While Diverting Fast Radix does start to lose ground to RADULS2, falling behind as input sizes grow beyond 10^6, it maintains its lead on the other algorithms. LaMarca and Ladner describes how the separation in target buckets and the randomness of selecting from several buckets for the next element dealt (among other things that might occupy cache lines) leads to an increase in cache misses that becomes prominent between input sizes of 10^5 and 10^6, shown in Figure 28. In exactly this range, we see RADULS2, using its non-temporal writes to avoid the cache beginning to make ground.

Table 9 shows timing results normalized against `std::sort`. For tiny input sizes, the approaches of both Diverting Fast Radix and RADULS2 are not well-optimized. Starting at inputs of size 10^4, Diverting Fast Radix holds a clear advantage at nearly four times as fast as `std::sort`. While at 10^4, there is a small-input optimization issue for RADULS2 that ceases to be an issue at 10^5, where Diverting Fast Radix is more than 60% faster than RADULS2 and still nearly four times as fast

as `std::sort`. At 10^6, Diverting Fast Radix is still almost 10% faster than RADULS2, and three times as fast as `std::sort`. Beyond that range, the lack of cache and TLB-optimization is clear in both Diverting Fast Radix and Ska Sort as RADULS2 maintains its linear performance against `std::sort`'s higher-order algorithmic complexity. However, while Diverting Fast Radix is slower than RADULS without cache and TLB-optimization in the 10^7 to 10^9 range, it is still roughly twice as fast as `std::sort`.

Chapter 5

Analysis

In the previous chapter, we showed how our specific implementations of standard radix sorts, as well as Fast Radix and Diverting Fast Radix, performed against other recognized implementations of radix sorts and also against std::sort. In this chapter, we will analyse the count estimation approach used in the Fast Radix algorithm and separately, the diversion approach we use with LSD Radix Sort as compared to the established diversion approach with MSD Radix Sort. Our analysis will consider the cost of an average pass and then the number of passes performed on average. Our Diverting Fast Radix algorithm combines those two approaches such that their separate analysis will show the overall improvement due to reduced passes and less work per pass which allows Diverting Fast Radix to perform competitively against state-of-the-art algorithms.

Previous research in radix sorts highlights that the amortized cost for per-pass work is less in LSD Radix Sorts than in MSD Radix Sorts due primarily to the many extra buckets that one must keep track of [Cormen et al., 2009, p. 197]. We will model the difference in average cost to perform a given radix sort pass when comparing a traditional LSD Radix Sort pass to that of Fast Radix, showing that Fast Radix allows for a lower cost by avoiding counting.

In considering the average number of passes performed by diverting radix sorts, we will consider diverting MSD Radix Sort and our proposed diverting LSD Radix Sort. Current research on MSD Radix Sorts consider diversion as a means to reduce cost[Kokot et al., 2018]. There is limited occurrence of any diverting LSD Radix Sorts, with the summary consideration of diversion after a constant number of passes[Sedgewick, 1998, Ch.10] and a more recent approach involving LSD Radix

Sorting floating point numbers that proposed diverting after "$Math.min(\log 2n + 9, 32)$"[Maus, 2019] bits. We will model the passes performed by both diverting MSD Radix Sort and our diverting LSD Radix Sort, where diversion is based on bucket size, to show that the average number of passes performed is the same in either case.

5.1 Recap on the LSD Algorithms

The difference between Fast Radix and LSD Radix Sort can be described as a reduction in counting passes by way of estimation of counts and the processing of any overflow due to difference from the estimation.

The LSD Radix Sort algorithm (LSD) sorts an array of N records by iteratively sorting into M buckets, one for each of the possible digits $d \in \mathcal{D}$ for each of the component digits $\mathcal{D} \in \mathcal{K}$ in increasing order of their significance in the key, $\mathcal{K}$. Note that M is $|\mathcal{D}|$.

The steps of each pass of the algorithm are

Count the number N_d of entries with digit d, for each $d \in \mathcal{D}$, by making one scan of the array;

Deal the entries of the array into their corresponding bucket of size N_d, by making one scan of the array;

Iterate by applying these steps to the newly dealt buckets in order of corresponding digit d for the next most significant $\mathcal{D} \in \mathcal{K}$.

The time, $T_{LSD}(N)$, for the LSD Radix Sort algorithm on N records, can be expressed in terms of

$T_{C(LSD)}(N)$ the time for the counting step on N records;

$T_{D(LSD)}(N)$ the time for the dealing step on N records into M buckets of size N_d, for $d \in \mathcal{D}$;

Iteration $\sum_{\mathcal{D}\in\mathcal{K}} (T_{LSD}(N))$ across the $|\mathcal{K}|$ subproblems of size N, for $\mathcal{D} \in \mathcal{K}$.

We define the constants for the basic steps of the LSD Radix Sort algorithm: c_{init}, the cost to initialize each of the M counts to zero; $c_{C(LSD)}$, the cost to process each of the N array entries and increment the corresponding count; c_{psum}, the cost to convert the M counts to indexes representing the dealing destinations of corresponding buckets; $c_{D(LSD)}$, the cost to deal each of the N array entries into the corresponding bucket;

Hence, $T_{C(LSD)}(N) = \left(\left(c_{init} + c_{psum}\right) \times M + c_{C(LSD)} \times N\right)$ and $T_{D(LSD)}(N) = c_{D(LSD)} \times N$.

So we can express $T_{LSD}(N)$ as the summation

$$T_{LSD}(N) = \sum_{\mathcal{D}\in\mathcal{K}} \left(T_{C(LSD)}(N) + T_{D(LSD)}(N)\right) \tag{5.1.1}$$

where $\sum_{\mathcal{D}\in\mathcal{K}}$ represents iteration over the digits in the key from least to most significant.

The Fast Radix Sort algorithm (FR) sorts an array of N records by recursively sorting into M buckets, one for each of the digits $d \in \mathcal{D}$. The counting step is replaced by estimates of the bucket sizes; the dealing step is done using an additional overflow bucket, and then a step to process the overflow corrects the bucket contents by performing a counting pass on the overflow bucket and a second deal of entries from the overflow bucket to extension buckets that correspond to the full buckets. Prior to the last pass, counting is performed identically to LSD Radix Sort and the last dealing pass is the same.

The steps of each pass of the algorithm are

Estimate the number X_d of entries with digit d, for each $d \in \mathcal{D}$;

Deal the entries of the array into their corresponding bucket of size X_d, or the overflow bucket, by making one scan of the array;

Process Overflow by counting the number O_d of entries with digit d in the overflow bucket and then by dealing the X_{ov} entries of the overflow bucket into their corresponding buckets of size O_d, such that $N_d = O_d + X_d$ where bucket X_d became full and N_d is the amount of X_d that was filled otherwise;

Iterate by applying these steps to the newly dealt buckets in order of corresponding digit d, first from the corresponding estimated bucket and than from any overflow bucket, for the next most significant $\mathcal{D} \in \mathcal{K}$.

The time, $T_{(FR)}(N)$, for the Fast Radix Sort algorithm on N records, can be expressed in terms of

$T_{E(FR)}(N)$ the time to initialize the estimates X_d, $d \in \mathcal{D}$;

$T_{D(FR)}(N)$ the time for the dealing step on N records into M buckets of size X_d, for $d \in \mathcal{D}$, or into the overflow bucket;

$T_{Pg_o(FR)}(N)$ the time for processing the overflow by scanning the overflow bucket of size X_{ov} and dealing the entries into the corresponding bucket;

Iteration $\sum_{\mathcal{D}\in\mathcal{K}} (T_{FR}(N))$ across the $|\mathcal{K}|$ subproblems of size N, for $\mathcal{D} \in \mathcal{K}$.

We define the constants for the basic steps of the Fast Radix Sort algorithm: $c_{E(FR)}$, the cost to initialize each of the M estimates; $c_{D(FR)}$, the cost to deal each of the N array entries into the corresponding bucket, or the overflow bucket; $c_{Pg_o(FR)}$, the cost to count and then deal each of the X_d entries of the overflow bucket into the corresponding bucket;

Hence, $T_{E(FR)}(N) = c_{E(FR)} \times M$ $T_{D(FR)}(N) = c_{D(FR)} \times N$. and $T_{Pg_o(FR)}(N) = c_{Pg_o(FR)} \times X_{ov}$.

So we can express $T_{FR}(N)$ as a single LSD Radix Sort pass on the most significant digit and a summation over the remaining digits

$$T_{(FR)}(N) = \left(T_{C(LSD)}(N) + T_{D(LSD)}(N)\right) + \sum_{\mathcal{D}\in\mathcal{K}'} \left(T_{E(FR)}(N) + T_{D(FR)}(N) + T_{Pg_o(FR)}(N)\right) \quad (5.1.2)$$

where $\sum_{\mathcal{D} \in \mathcal{K}'}$ represents iteration over the digits in the key from least to most significant, excluding the most significant digit.

5.2 Recap on the Diverting Algorithms

The concept of diversion is relevant to hybrid sort algorithms which utilise multiple sort algorithms at the different passes during recursion [Kokot et al., 2018]. Diversion strategies in radix sort are predicated on a threshold of a bucket size N_d. If the size is below the threshold then the recursive call to radix sort on that bucket defaults to a different sort algorithm. For our analysis, we will assume the diverting sort algorithm is non-recursive so does not involve further passes. We deal with the non-diverting case by regarding it as adopting a null diversion strategy which never diverts from the radix sort.

The diverting MSD Radix Sort algorithm (dMSD) sorts an array of N records by recursively sorting into M buckets, one for each of the possible digits $d \in \mathcal{D}$ for each of the component digits $\mathcal{D} \in \mathcal{K}$ in decreasing order of their significance in the key, $\mathcal{K}$. If a bucket is of size DT or smaller, a diversion algorithm is used to finish the sorting of that bucket with no further recursion. Note that M is $|\mathcal{D}|$.

The steps of each pass of the algorithm are

Count the number N_d of entries with digit d, for each $d \in \mathcal{D}$, by making one scan of the array;

Deal the entries of the array into their corresponding bucket of size N_d, by making one scan of the array;

Recurse by applying by applying Radix Sort to each of the M buckets of size N_d, for $d \in \mathcal{D}$, so long as $DT < N_d$.

Divert by applying the diverting algorithm to buckets $N_d < DT$.

The time, $T_{dMSD}(N)$, for the dMSD Radix Sort algorithm on N records, can be expressed in terms of

$T_{C(dMSD)}(N)$ the time for the counting step on N records;

$T_{D(dMSD)}(N)$ the time for the dealing step on N records into M buckets of size N_d, for $d \in \mathcal{D}$;

$T_{DIV}(N)$ the time for running the diversion sort on N records;

Recursion $\sum_{d\in\mathcal{D}} (T_{dMSD}(N_d))$ across the $|\mathcal{D}|$ subproblems of size N_d, using the next most significant digit $\mathcal{D} \in \mathcal{K}$.

We define the constants for the basic steps of the dMSD Radix Sort algorithm: c_{init}, the cost to initialize each of the M counts to zero, defined previously; $c_{C(dMSD)}$, the cost to process each of the N array entries and increment the corresponding count; c_{psum}, the cost to convert the M counts to indexes representing the dealing destinations of corresponding buckets, defined previously; $c_{D(dMSD)}$, the cost to deal each of the N array entries into the corresponding bucket;

Hence, $T_{C(dMSD)}(N) = \left(\left(c_{init} + c_{psum}\right) \times M + c_{C(dMSD)} \times N\right)$ and $T_{D(dMSD)}(N) = c_{D(dMSD)} \times N$.

So we can express $T_{dMSD}(N)$ as the recursion

$$T_{dMSD}(N) = \begin{cases} T_{DIV}(N), & N <= DT \\ \text{Do Nothing}, & \text{all digits processed} \\ \left(T_{C(dMSD)}(N) + T_{D(dMSD)}(N) + \sum_{d\in\mathcal{D}} (T_{dMSD}(N_d))\right), & \text{otherwise} \end{cases} \quad (5.2.1)$$

where $N = \sum_{d\in\mathcal{D}} N_d$.

The diverting LSD Radix Sort algorithm (dLSD) sorts an array of N records by performing LSD Radix Sort on $P(N)$ high-order digits $\mathcal{D} \in \mathcal{K}$, then recursively sorting any remaining buckets that are larger than the diversion threshold DT.

The steps of each pass of the algorithm are

Sort using LSD Radix Sort into $M^{P(N)}$ buckets of size N_d of records with digit d, for each $d \in \mathcal{D}^{\mathcal{P}}$ where $\mathcal{D}^{\mathcal{P}}$ is the digit composed of the $P(N)$ high-order digits $\mathcal{D} \in \mathcal{K}$;

Recurse by sorting each of the buckets of size N_d, for $d \in \mathcal{D}^{\mathcal{P}}$, so long as $DT < N_d$;

Divert by applying the diverting algorithm to buckets $N_d <= DT$;

The time, $T_{dLSD}(N)$, for the dLSD Radix Sort algorithm on N records, can be expressed in terms of

$T_{LSD}(N)$ the time to perform LSD Radix Sort on N records for the $P(N)$ high-order digits $\mathcal{D} \in \mathcal{K}$, defined previously;

$T_{S(dLSD)}(N)$ the time to find if any buckets of size $N_d <= DT$, for $d \in \mathcal{D}^{\mathcal{P}}$;

$T_{DIV}(N)$ the time for running the diversion sort on N records, defined previously;

Recursion $\sum_{d \in \mathcal{D}^{\mathcal{P}}} (T_{dMSD}(N))$ across the $|\mathcal{D}|$ subproblems of size N_d, using the next $P(N_d)$ most significant digits $\mathcal{D} \in \mathcal{K}$ (or just any remaining digits if there are too few).

We define the constants for the basic steps of the dLSD Radix Sort algorithm: $c_{scan(dLSD)}$, the cost to scan the array for buckets larger than DT; Hence, $T_{S(dLSD)}(N) = c_{scan(dLSD)} \times N$.

So we can express $T_{dLSD}(N)$ as the recursion

$$T_{dLSD}(N) = \begin{cases} T_{DIV}(N), & N <= DT \\ \text{Do Nothing}, & \text{all digits processed} \\ \left(T_{LSD}(N) + T_{S(dLSD)}(N) + \sum_{d \in \mathcal{D}^{\mathcal{P}}} (T_{dLSD}(N_d))\right), & \text{otherwise} \end{cases} \quad (5.2.2)$$

where $N = \sum_{d \in \mathcal{D}^{\mathcal{P}}} N_d$.

5.3 The Analysis in Overview

We use *uniformly random input* to mean that the input to be sorted contains records which have fixed-digit keys, where each possible key has the same probability of occurring in each record. We only consider cases where the key takes a fixed number of bits and do not consider the implications on distribution choice for cases such as variable length string sorting considered in some work[Andersson and Nilsson, 1994]. As in this analysis, most authors chose keys from a uniform distribution for at least some of their results, allowing for any possible key to occur and allowing duplicates. While some work considered digits of variable length[Al-Badarneh and El-Aker, 2004, Maus, 2002], for both practical reasons and simplicity of analysis, this work implements and performs analysis using a fixed-length digit of eight bits, a common decision in most other work.

We use the term pass to describe an algorithm processing all records in an input. A pass generally encompasses the processing of all records where that processing is based on the consideration of the same parts of each record's key, but need not be the same as a pass described in an algorithm or implementation. The sum of all passes performed by an algorithm encompasses all work on the input for that algorithm. The processing within a pass need not be contiguous in time, it is strictly a measure of how much of an input was processed by the same criteria overall once the algorithm was completed. A pass may be incomplete if not all records are processed, and when counting such passes, only the fraction of the pass completed is counted.

It is important to distinguish a pass considered during this analysis from how one might consider a pass in an algorithm or how an implementation may loop through its input. This distinction can be illustrated by considering the cost summation from the LSD Radix Sort algorithm as defined above, $T_{LSD}(N) = \sum_{\mathcal{D}\in\mathcal{K}} \left(T_{C(LSD)}(N) + T_{D(LSD)}(N)\right)$. In practical implementations, all but the first counting pass would happen in conjunction with a dealing pass[Friend, 1956], and thus the cost of the first counting pass as implemented would be more than the counting component of subsequent passes that were also dealing[Thiel et al., 2016]. In some implementations, multiple digits may be

counted in a single dealing pass[Rahman and Raman, 2001]. There are also hidden costs relating to cache and TLB misses that are impacted by counting while dealing[Rahman and Raman, 2001] that are not distinctly attributable to either a counting or a dealing pass. The definition of pass we use here wraps such details in constants, as in $c_{C(LSD)}$ for counting on the LSD Radix Sort algorithm, amortizing costs across like passes and eliding issues of difference in cache and TLB miss timing that are outside the scope of this analysis.

Our primary approach to analysing work in radix sorts is to model radix sorts through occupancy distribution, specifically the uniformly random occupancy distribution. This is often referred to as the ball and urn model as one can imagine the problem space as that of a number of balls having been randomly placed into a number of urns; the theory of this model deals with the characteristics of such a system.

Given the standard ball and urn model where $N, M \in Z_{\geq 1}$, we can use Sedgewick and Flajolet's description:

> The average number of urns with k balls, when N balls are randomly distributed in M urns, is:
>
> $$M\binom{N}{k}(\frac{1}{M})^k(1-\frac{1}{M})^{N-k}$$
>
> [Sedgewick and Flajolet, 2013, p. 503, Theorem 9.5]

Given the average number of urns with k balls, we can also know the average number of balls going into urns with k balls by multiplying the occupancy distribution equation by k. If we want to know the number of urns with up to a certain number of values, we can perform a summation between 0 and that number, using k as the index. The number of balls in urns that contain values above a threshold is the total number of balls take away the previous summation. The term inside the summation can also be multiplied by k to find average numbers of balls in the specified urns.

We represent the average number of balls within urns containing exactly k balls when uniformly distributing N balls among M urns as $f(N, M, k)$:

$$f(N, M, k) = kM\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \tag{5.3.1}$$

This is applicable to Radix Sort analysis as the number of urns M represents the number of ways a digit can be represented, which is the number of buckets radix sorts use, and is 256 for the standard 8-bit digit. The number of balls N, corresponds directly to the number of records in an input. This approach groups urns by the number of balls they hold, and so k represents that number. The summing of these groups from 0 to T is just the sum over $f(N, M, k)$, and we consider it as $g_l(N, M, p, T)$:

$$g_l(N, M, p, T) = \sum_{k=0}^{T} f(N, M^p, k) = \sum_{k=0}^{T} kM^p\binom{N}{k}(\frac{1}{M^p})^k(1 - \frac{1}{M^p})^{N-k} \tag{5.3.2}$$

p represents the passes considered, where $p = 1$ in cases where only one pass of radix sort is considered. For our purpose, when summing over k, the threshold T corresponds to one of two cases: 1) a static diversion threshold representing how few records must be in buckets before they are sorted by diverting to another algorithm; or 2) an overflow threshold representing how many records we allow into a bucket before we must overflow. In either case, we will supplement our application of the ball and urn model with Pascal's Rule and the Trinomial Revision identity.

Given the nature of binomial coefficients, and for $N, k \in \mathbb{R}_{>0}$, Pascal's Rule allows us to decompose a binomial coefficient into a sum of two other binomial coefficients:

$$\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1} \tag{5.3.3}$$

[Graham et al., 1994, p.174, Table 174, 4^{th} identity]

Equation 5.3.3 is useful when we need to increase or decrease the number of terms. In this thesis, Pascal's Rule is often the immediate path to change an equation into a suitable form.

When working with binomial coefficients, it is not uncommon to find that one does not have a particular binomial coefficient that one is looking for, but Trinomial Revision can often be used to reshuffle binomial coefficients:

$$\binom{N}{j}\binom{j}{k} = \binom{N}{k}\binom{N-k}{j-k} \tag{5.3.4}$$

[Graham et al., 1994, p.174, Table 174, 6^{th} identity]

Equation 5.3.4 describes how the number of ways to pick the j men from the N mortals times the number of ways to pick the k Socrates from those j men is the same as the number of ways to pick the same k Socrates from those N mortals times the number of ways to pick the $j-k$ men who are not Socrates from the $N-k$ mortals who are not Socrates. It is a way of getting a $\binom{N}{k}$ factor when you have something in the form of $\binom{N}{j}\binom{j}{k}$.

Given the recap of algorithms that detail how cost will be considered in this analysis, the clarification of terms used and the general mathematical model and approach so defined, we can approach the analysis in two parts. Firstly, we will consider Theorem 8 where we will compare the cost of counting in LSD Radix Sort with the modeled cost of overflow processing in Fast Radix, showing that Fast Radix costs less. Secondly, we will consider Theorem 11, where we will model the cost of diverting MSD Radix Sort and compare it with the model of our diverting LSD Radix Sort and show that they perform the same number of passes on average. Diverting Fast Radix combines both these improvements and is thus shown to have a lower cost, the details of which will be discussed in our interpretation of the analysis at the end of this chapter.

5.4 Diverting Fast Radix Average Work Per Pass

In this section we consider the average cost per pass, focusing on the removal of the cost of counting and the analysis of the cost of processing overflow. We can compare the cost of counting in LSD Radix Sort against the overflow processing of Fast Radix and similarly compare the cost of their corresponding dealing.

$$
\begin{aligned}
T_{C(LSD)}(N) &= \left(\left(c_{init} + c_{psum}\right) \times M + c_{C(LSD)} \times N\right) \\
T_{D(LSD)}(N) &= c_{D(LSD)} \times N \\
T_{E(FR)}(N) &= c_{E(FR)} \times M \\
T_{D(FR)}(N) &= c_{D(FR)} \times N \\
T_{Pg_o(FR)}(N) &= c_{Pg_o(FR)} \times X_{ov}
\end{aligned}
$$

While $M << N$ in many cases rendering the M component less significant in LSD Radix Sorts, we note that the per-bucket cost to initialize in LSD Radix Sorts c_{init} is about same as the per-buck cost to set bucket estimations in Fast Radix $c_{E(FR)}$, as it is either initializing to zero or initializing to the estimated bucket sizes. Fast Radix has no cost that corresponds to the prefix summation $c_{psum} \times M$ in LSD Radix Sort.

The difference between the per-record cost of joint dealing and counting in LSD Radix Sort $c_{D(FR)} + c_{C(LSD)}$ and the corresponding per-record cost of dealing $c_{D(FR)}$ in Fast Radix consists of an extra check against overflow in Fast Radix compared to reading a count, incrementing it and writing it back out in LSD Radix Sort. As there is no corresponding initial counting pass in Fast Radix[Thiel et al., 2016], the difference in cost is even higher.

If this per-record difference in cost exceeds the overflow processing cost $c_{Pg_o(FR)} \times X_{ov}$ then Fast

Radix must have a lower cost for that pass. What follows is a model of the average size of overflow ov as a function of N and the effectively fixed value M which shows that $\frac{ov}{N}$ quickly becomes insignificant as N increases, showing that Fast Radix costs less per pass, save on the last pass where it costs the same as LSD Radix Sort.

As we know that the average number of records in each bucket must be $\frac{N}{M}$ when randomly distributing N records into M buckets, it makes sense that the total capacity for records, distributed randomly over M buckets is the same.

The specific approach to distributing capacity at random can vary, but we can consider the approach where some buckets have an integer capacity of c and the remaining buckets have a capacity of $c + 1$ and the total capacity of all buckets is N. It does not matter which buckets have which capacities as the bucket a record ends up in is random, and the likelihood of ending up in that bucket is not related to the capacity of that bucket.

The actual capacity of each bucket must be an integer value. It is practical to consider that $M - a$ random buckets have a capacity of $\lfloor \frac{N}{M} \rfloor = c$ records and the remaining a buckets have a capacity of $\lceil \frac{N}{M} \rceil = c + 1$ records where $N = cM + a$, $a < M$ and $c, a \in \mathbb{Z}_{\geq 0}$.

If $a = 0$ then each bucket contains $c = \frac{N}{M}$ records. If $a > 0$ then the average bucket has a capacity of $\frac{(M-a)c}{M} + \frac{a(c+1)}{M} = \frac{cM+a}{M} = \frac{N}{M}$ records.

Theorem 5. *Given uniformly random input. Then the fraction of N that overflows can be modeled as:*

$$g_o(N, M) = \frac{1}{N} \sum_{k=0}^{\lceil \frac{N}{M} \rceil - 1} (\frac{N}{M} - k) M \binom{N}{k} (\frac{1}{M})^k (1 - \frac{1}{M})^{N-k} \tag{5.4.1}$$

If we consider the capacity in and average bucket that contains k records is $c(N, M, k)$ and the average number of records in those buckets as the already defined $f(N, M, k)$ from Equation 5.3.1, then we can define overflow in terms of the difference of those functions over the range of buckets

with more records than capacity.

$$f(N, M, k) = kM\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \tag{5.3.1 revisited}$$

Proof. As Equation 5.3.1 represents the number of records in buckets that contain k records, we can represent the capacity of those buckets as $c(N, M, k)$ by multiplying the number of buckets with k records times the average capacity $\frac{N}{M}$:

$$c(N, M, k) = \frac{N}{M}M\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \tag{5.4.2}$$

As such, the fraction of N that is overflow or underflow within buckets containing k records can be defined as $g_o(N, M, k)$, which must be the difference between average capacity in buckets with k records and those records.

$$g_o(N, M, k) = \frac{1}{N}c(N, M, k) - \frac{1}{N}f(N, M, k) \tag{5.4.3}$$

As overflow must happen in buckets containing $\lceil\frac{N}{M}\rceil$ or more records, no underflow can happen in that range of buckets and as overflow is a zero-sum system, the overflow and underflow are the same, then the average overflow must be the difference between capacity and balls allocated in urns with k values where the range of k is from 0 to $\lceil\frac{N}{M}\rceil - 1$:

$$\begin{aligned} g_o(N, M) &= \sum_{k=0}^{\lceil\frac{N}{M}\rceil-1} g_o(N, M, k) \\ &= \sum_{k=0}^{\lceil\frac{N}{M}\rceil-1} (\frac{N}{M} - k)\frac{M}{N}\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \end{aligned}$$

■

Theorem 6. *The overflow for buckets containing k records can be defined recursively as:*

$$g_o(N, M, k) = \frac{1}{M}g_o(N - 1, M, k - 1) + (1 - \frac{1}{M})g_o(N - 1, M, k) \tag{5.4.4}$$

Proof. First we note the expansion of Equation 5.4.3.

$$g_o(N, M, k) = (\frac{N}{M} - k)\frac{M}{N}\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k}$$

If we apply this to $N + 1$ we get the following:

$$g_o(N + 1, M, k) = (\frac{N + 1}{M} - k)\frac{M}{N + 1}\binom{N + 1}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N+1-k}$$

We can then apply Pascal's Rule to split the right side into two terms:

$$\begin{aligned} g_o(N + 1, M, k) =&(\frac{N + 1}{M} - k)\frac{M}{N + 1}\binom{N}{k - 1}(\frac{1}{M})^k(1 - \frac{1}{M})^{N+1-k} \\ &+(\frac{N + 1}{M} - k)\frac{M}{N + 1}\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N+1-k} \end{aligned}$$

We can then factor out the $\frac{1}{M}$ and $1 - \frac{1}{M}$ factors, adjusting the probability exponents appropriately:

$$\begin{aligned} g_o(N + 1, M, k) =&\frac{1}{M}(\frac{N + 1}{M} - k)\frac{M}{N + 1}\binom{N}{k - 1}(\frac{1}{M})^{k-1}(1 - \frac{1}{M})^{N-(k-1)} \\ &+(1 - \frac{1}{M})(\frac{N + 1}{M} - k)\frac{M}{N + 1}\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \end{aligned}$$

At this point we note that it looks very close to what we want, but there is an uncooperative factor in each term.

$$\begin{aligned} g_o(N + 1, M, k) =&\frac{1}{M} \quad \frac{N + 1 - kM}{N + 1}\binom{N}{k - 1}(\frac{1}{M})^{k-1}(1 - \frac{1}{M})^{N-(k-1)} \\ &+(1 - \frac{1}{M}) \quad \frac{N + 1 - kM}{N + 1}\binom{N}{k}(\frac{1}{M})^k(1 - \frac{1}{M})^{N-k} \end{aligned}$$

We know that $\frac{N+1-kM}{N+1}$ can be decomposed as follows:

$$\frac{N + 1 - kM}{N + 1} = \frac{N - kM}{N} + \frac{kM}{N(N + 1)}$$

$\frac{N-kM}{N}$ is the factor that we are looking for, so we can pull out the unneeded term and collapse the

remainder into the desired $g_o(N, M, k)$ term:

$$\begin{aligned} g_o(N+1, M, k) = & \frac{1}{M}\frac{N+1-kM}{N+1} & \binom{N}{k-1}(\frac{1}{M})^{k-1}(1-\frac{1}{M})^{N-(k-1)} \\ & +(1-\frac{1}{M})\frac{kM}{N(N+1)} & \binom{N}{k}(\frac{1}{M})^{k}(1-\frac{1}{M})^{N-k} \\ & +(1-\frac{1}{M})g_o(N, M, k) \end{aligned}$$

We can adjust the second term such that the binomial coefficient matches by noting the following identity derived from considering the binomial coefficient's factorial form:

$$\binom{N}{k} = \frac{N+1-k}{k}\binom{N}{k-1}$$

This yields the following:

$$\begin{aligned} g_o(N+1, M, k) = & \frac{1}{M}\frac{N+1-kM}{N+1} & \binom{N}{k-1}(\frac{1}{M})^{k-1}(1-\frac{1}{M})^{N-(k-1)} \\ & +(1-\frac{1}{M})\frac{N+1-k}{k}\frac{kM}{N(N+1)} & \binom{N}{k-1}(\frac{1}{M})^{k}(1-\frac{1}{M})^{N-k} \\ & +(1-\frac{1}{M})g_o(N, M, k) \end{aligned}$$

We can once again adjust the probability exponents appropriately, which also changes the $(1-\frac{1}{M})$ factor into the desired $(1-\frac{1}{M})$ form:

$$\begin{aligned} g_o(N+1, M, k) = & \frac{1}{M}\frac{N+1-kM}{N+1} & \binom{N}{k-1}(\frac{1}{M})^{k-1}(1-\frac{1}{M})^{N-(k-1)} \\ & +\frac{1}{M}\frac{N+1-k}{k}\frac{kM}{N(N+1)} & \binom{N}{k-1}(\frac{1}{M})^{k-1}(1-\frac{1}{M})^{N-(k-1)} \\ & +(1-\frac{1}{M})g_o(N, M, k) \end{aligned}$$

We can then refactor once more given that the binomial theorem parts are the same in both terms:

$$\begin{aligned} g_o(N+1,M,k) =& \frac{1}{M}(\frac{N+1-kM}{N+1} + \frac{N+1-k}{k}\frac{kM}{N(N+1)}) \\ & \binom{N}{k-1}(\frac{1}{M})^{k-1}(1-\frac{1}{M})^{N-(k-1)} \\ & +(1-\frac{1}{M})g_o(N,M,k) \end{aligned}$$

We now collapse the last undesirable factor using the following:

$$\frac{N+M-kM}{N} = \frac{N+1-kM}{N+1} + \frac{N+1-k}{k}\frac{kM}{N(N+1)}$$

which yields:

$$\begin{aligned} g_o(N+1,M,k) &= \frac{1}{M}g_o(N,M,k-1) + (1-\frac{1}{M})g_o(N,M,k) \\ g_o(N,M,k) &= \frac{1}{M}g_o(N-1,M,k-1) + (1-\frac{1}{M})g_o(N-1,M,k) \end{aligned}$$

■

Theorem 7. *The average overflow can then be defined recursively as:*

$$g_o(N,M) = g_o(N-1,M) - \frac{1}{M}g_o(N-1,M,\lceil\frac{N}{M}\rceil - 1) \tag{5.4.5}$$

Proof. If we replace the internals of the summation in the equation $g_o(N,M) = \sum_{k=0}^{\lceil\frac{N}{M}\rceil-1} g_o(N,M,k)$ with Equation 5.4.4 from Theorem 6, we see that we have two paired summations and some extra terms:

$$\begin{aligned} g_o(N,M) =& \frac{1}{M}g_o(N-1,M,-1) \quad +\frac{1}{M}(\sum_{k=0}^{\lceil\frac{N}{M}\rceil-2} g_o(N-1,M,k)) + 0 \\ & +0 \qquad\qquad +\frac{M-1}{M}(\sum_{k=0}^{\lceil\frac{N}{M}\rceil-2} g_o(N-1,M,k)) + \frac{M-1}{M}g_o(N-1,M,\lceil\frac{N}{M}\rceil - 1) \end{aligned}$$

As the a negative term for the bucket size yields a zero term, it can be discarded. The summations

are identical internally, and their factors merge cleanly as $\frac{1}{M} + \frac{M-1}{M} = 1$:

$$g_o(N, M) = (\sum_{k=0}^{\lceil \frac{N}{M} \rceil - 2} g_o(N-1, M, k)) + \frac{M-1}{M} g_o(N-1, M, \lceil \frac{N}{M} \rceil - 1)$$

By adding $\frac{1}{M} g_o(N-1, M, \lceil \frac{N}{M} \rceil - 1)$ to the second term and collapsing it into the summation, raising its upper bound and subtracting it again afterward, we arrive at the desired equation. ■

Theorem 8. *Given uniformly random input. Given that a sufficiently large input size is used. Then, on average, Fast Radix has a lower cost on a given pass than LSD Radix Sort.*

Proof. The counting component of LSD Radix Sort has a linear cost associated with it. By not counting, the cost of each pass is reduced by a factor. The cost of counting is replaced by the cost of processing overflow. The cost of processing overflow as a fraction of N decreases monotonically as the size of the input increases, as shown by Theorem 6, and thus the cost of processing overflow must be sub-linear. Evaluating $g_o(N, M)$ shows that it becomes small quickly. ■

5.5 Diverting Radix Sort Average Number of Passes

Evaluating the recursive costs for $T_{dMSD}(N)$ and $T_{dLSD}(N)$ becomes about measuring how many records recurse after a certain number of passes. We define these costs more formally as $g_m(N, M, P, DT)$ and $g_l(N, M, P, DT)$ respectively, where they represent the number of records that divert after pass P. Assuming that these are mathematically identical and that a $P(N)$ exists wherein there is no diversion in prior passes, then the difference in overall cost is the difference between the exponential cost of count initialization and prefix summing for MSD Radix Sort $\sum_{p=0}^{P(N)-1} \left(\left(c_{init} + c_{psum} \right) \times M^p \right)$ on the one hand and the linear cost of that counting work in LSD Radix Sort along with the scan for large buckets $P(N) \times \left(\left(c_{init} + c_{psum} \right) \times M \right) + c_{scan(dLSD)} \times N$, where the scan for large buckets $c_{scan(dLSD)}$

represent sequential reads, registry updates and comparisons that can be optimized to be quite small.

If we show that diversion on one pass means that any remaining diversion will happen on the next pass, then the difference in cost narrows as every bucket that did not divert will need to perform an additional pass which will have an identical cost (effectively a simple counting sort on that digit) for both algorithms, with this cost eclipsing diverting MSD Radix Sort's counting costs for all prior passes as the number of buckets requiring an additional pass approaches $M^{P(N)}$.

This section of the analysis proves that the assumptions made here are valid and that the expected number of passes before diversion $P(N)$ can be known for a given input size N, digit size M and diversion threshold DT.

In considering how distribution during radix sorts leads to diversion, we examine the following equivalence: If one randomly deals N balls into M^{a+b} urns, the fraction of those M^{a+b} urns that would get k balls would be the same as if one first randomly deals N balls into M^a urns and then randomly deals the balls from each of those M^a urns into a separate set of M^b urns, adding up the fraction that get k balls (either emptying the M^b urns between each of these secondary dealings, or using new ones). A simple sanity check suggests that this makes sense, as if one has M^a sets of M^b urns then one would have the same number of urns as in the first approach and how one randomly distributes should not affect the average urns getting k balls as long as all approaches to dealing are completely random.

Theorem 9. *Given $N, k \in \mathbb{Z}_{\geq 0}$. Given $k \leq N$. Given $M, DT \in \mathbb{Z}_{>0}$. Given $a, b \in \mathbb{Z}_{>1}$. Then*

$$\binom{N}{k}(\frac{1}{M^{a+b}})^k(1-\frac{1}{M^{a+b}})^{N-k} = \sum_{j=0}^{N}\binom{N}{j}(\frac{1}{M^a})^j(1-\frac{1}{M^a})^{N-j}\binom{j}{k}(\frac{1}{M^b})^k(1-\frac{1}{M^b})^{j-k}$$

Proof. The approach used will be to transform the right-hand side into the left-hand side. We can perform trinomial reversion, $\binom{N}{j}\binom{j}{k} = \binom{N}{k}\binom{N-k}{j-k}$, to convert the binomial coefficients to the desired

form, factoring $\binom{N}{k}$ outside of the summation.

$$\binom{N}{k} \sum_{j=0}^{N} \binom{N-k}{j-k} (\frac{1}{M^a})^j (1 - \frac{1}{M^a})^{N-j} (\frac{1}{M^b})^k (1 - \frac{1}{M^b})^{j-k}$$

As the $(\frac{1}{M^{a+b}})^k (1 - \frac{1}{M^{a+b}})^{N-k}$ factor from the left-hand side is independent of the summation, we can multiple it against the outside of our summation and divide by the inside of the summation to give the form of the left-hand side equation outside of the summation:

$$\binom{N}{k} (\frac{1}{M^{a+b}})^k (1 - \frac{1}{M^{a+b}})^{N-k}$$

so long as the summation being multiplied against equals 1:

$$\sum_{j=0}^{N} \binom{N-k}{j-k} (\frac{1}{M^a})^j (1 - \frac{1}{M^a})^{N-j} (\frac{1}{M^b})^k (1 - \frac{1}{M^b})^{j-k} (\frac{1}{M^{a+b}})^{-k} (1 - \frac{1}{M^{a+b}})^{-(N-k)} = 1$$

To complete the proof, we transform the equation such that it looks like the binomial theorem. Shifting the index by k is an intuitive starting point:

$$\sum_{j=-k}^{N-k} \binom{N-k}{j} (\frac{1}{M^a})^{j+k} (1 - \frac{1}{M^a})^{N-j-k} (\frac{1}{M^b})^k (1 - \frac{1}{M^b})^j (\frac{1}{M^{a+b}})^{-k} (1 - \frac{1}{M^{a+b}})^{k-N} = 1$$

While perhaps not as intuitive, we note that Knuth defines $\binom{N}{k} = 0$ when N is a non-negative integer and k is less than 0[Knuth, 1997, p. 53]. Accordingly, we can start the index at zero as the terms we are ignoring are all equal to zero:

$$\sum_{j=0}^{N-k} \binom{N-k}{j} (\frac{1}{M^a})^{j+k} (1 - \frac{1}{M^a})^{N-j-k} (\frac{1}{M^b})^k (1 - \frac{1}{M^b})^j (\frac{1}{M^{a+b}})^{-k} (1 - \frac{1}{M^{a+b}})^{k-N} = 1$$

If we define $N - k = N'$ and replace, we get

$$\sum_{j=0}^{N'} \binom{N'}{j} (\frac{1}{M^a})^{j+k} (1 - \frac{1}{M^a})^{N'-j} (\frac{1}{M^b})^k (1 - \frac{1}{M^b})^j (\frac{1}{M^{a+b}})^{-k} (1 - \frac{1}{M^{a+b}})^{-N'} = 1$$

We can then factor into the form $\sum_{j=0}^{N'} \binom{N'}{j} A^j B^{N'} C^k = 1$, given:

$$\begin{aligned} A &= \frac{1}{M^a} \frac{M^a}{M^a - 1} \frac{M^b - 1}{M^b} &= \frac{M^b - 1}{(M^a - 1)M^b} \\ B &= \frac{M^a - 1}{M^a} \frac{M^{a+b}}{M^{a+b} - 1} &= \frac{(M^a - 1)M^b}{M^{a+b} - 1} \\ C &= \frac{1}{M^a} \frac{1}{M^b} M^{a+b} &= 1 \end{aligned}$$

As $C = 1$, we can reform this as the binomial theorem given that we know $AB + B = 1$:

$$\sum_{j=0}^{N'} \binom{N'}{j} (AB)^j B^{N'-j} = 1 \tag{5.5.1}$$

■

Whereas Theorem 9 considered counting the fraction of urns filled with only k balls, here we extend that to a range of values for k and consider the result in terms of number of balls in those urns.

Theorem 10. *Given $N \in \mathbb{Z}_{\geq 0}$. Given $M, DT \in \mathbb{Z}_{>0}$. Given $a, b \in \mathbb{Z}_{>1}$. Then*

$$\sum_{k=0}^{DT} k M^{a+b} \binom{N}{k} (\frac{1}{M^{a+b}})^k (1 - \frac{1}{M^{a+b}})^{N-k}$$

is equal to

$$\sum_{j=0}^{N} M^a \binom{N}{j} (\frac{1}{M^a})^j (1 - \frac{1}{M^a})^{N-j} \sum_{k=0}^{DT} k M^b \binom{j}{k} (\frac{1}{M^b})^k (1 - \frac{1}{M^b})^{j-k}$$

Proof. As this is only the summation to DT of k in Theorem 9, scaled by k and M^{a+b}, this is directly true. ■

We can now consider our model for LSD Radix Sort diversion by pass P.

Lemma 1. *Given $N \in \mathbb{Z}_{\geq 0}$. Given $M, DT \in \mathbb{Z}_{>0}$. Given $P \in \mathbb{Z}_{\geq 1}$. Then*

$$g_l(N, M, P, DT) = \sum_{k=0}^{DT} f(N, M^P, k) = \sum_{k=0}^{DT} k M^P \binom{N}{k} (\frac{1}{M^P})^k (1 - \frac{1}{M^P})^{N-k}$$

Proof. We can directly use the first summation in Theorem 10 by replacing $a + b$ with P. This is simply Equation 5.3.2. ∎

We can also consider our model for MSD Radix Sort diversion by pass P.

Lemma 2. *Given $N \in \mathbb{Z}_{>0}$. Given $M \in \mathbb{Z}_{>0}$. Given $DT \in \mathbb{Z}_{>0}$. Given $P \in \mathbb{Z}_{\geq 1}$. Then*

$$g_m(N, M, P, DT) = \sum_{j=0}^{N} M \binom{N}{j} (\frac{1}{M})^j (1 - \frac{1}{M})^{N-j} g_m(j, M, P - 1, DT)$$

and

$$g_m(N, M, 1, DT) = g_l(N, M, 1, DT)$$

Proof. By noting that the second summation in Theorem 10 can represent a recursive expansion that models MSD Radix Sort's recursive approach in dealing to buckets, we can replace $a + b$ with P as in Lemma 1. While Theorem 10 has the restriction $a, b \in \mathbb{Z}_{>1}$, as in Lemma 1, we can take advantage of the special of Equation 5.3.2. ∎

Theorem 11. *Given uniformly random input. Then the models for the average number of passes for diverting MSD Radix Sort and diverting LSD Radix Sort are the same.*

Proof. Lemma 1 and Lemma 2 show the models for diversion by a given pass in LSD and MSD Radix Sorts and Theorem 10 shows that these models are equivalent. ∎

Corollary 1. *Given uniformly random input. Given an input size N. Given buckets of size M and a diversion threshold DT that is not significantly larger than M. Given that after P_{max} passes all*

records can be said to have diverted. Given there is a minimum number of passes P after which some, but not all records will divert. Then all records will divert after $P + 1$ passes.

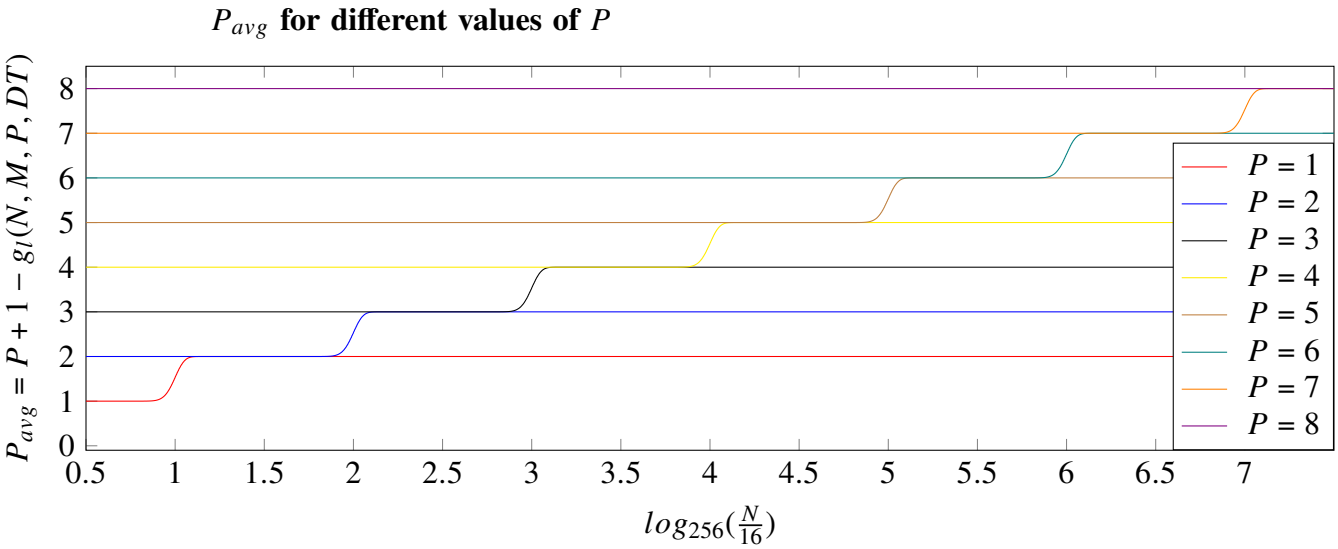

Figure 29: This graph shows the change in P_{avg} for diverting LSD Radix Sort, given undiverted buckets requiring one additional pass, as N increases given $DT = 16$, $M = 256$ and $P_{max} = 8$.

Proof. If we interpolate a graph of $P_{avg} = P + 1 - g_l(N, M, P, DT)$ for all values of P that shows very distinctly that diversion happens in either one pass or the next, as shown in Figure 29. While we cannot be sure that this corollary holds for all M and DT, if it did not hold for practical values, as considered in Figure 29 it would be immediately apparent that the transition areas (centered on the units of the log scale in the x-axis) would begin to overlap. ■

Corollary 2. *P can be defined as a function of N, M and DT for the purpose of identifying the number of passes before diversion is practical.*

Proof. Given Corollary 1, one can pre-calculate a lookup table that can be used at runtime by an implementation to determine an appropriate number of passes to perform before attempting diversion. ■

N_P \ DT	12	13	14	15	16
2	2152	2369	2587	2807	3028
3	568543	624985	681781	738891	796283
4	1.515134e+08	1.662638e+08	1.81093e+08	1.959924e+08	2.109549e+08
5	4.088219e+10	4.476093e+10	4.865587e+10	5.256526e+10	5.648764e+10
6	1.128276e+13	1.231412e+13	1.334814e+13	1.438454e+13	1.542307e+13
7	3.284352e+15	3.565789e+15	3.847232e+15	4.128679e+15	4.41013e+15
8	8.646911e+17	9.367487e+17	1.008806e+18	1.080864e+18	1.152922e+18

Table 10: Calculated values of the first N at which $g_l(N, M, P-1, DT)$ is effectively 0. DT are common diversion thresholds and N_P is the number of passes that should be performed at the specified threshold.

Table 10 is an example lookup table with standard values of DT for Insertion Sort, the diversion algorithm used by our implementation of Diverting Fast Radix and $M = 256$ as it is in most cases. This table provides a conservative approach in that a large proportion of buckets could be required to perform one additional pass, similar to the costs incurred by MSD Radix Sorts on their last pass.

5.6 Interpretation of Analysis: Diverting Fast Radix

Avoiding the counting pass in Fast Radix provides a clear reduction in cost. Diverting LSD Radix Sort can be shown to perform no more passes than diverting MSD Radix Sort and the cost can be shown to be similar. Where there is ambiguity in the cost of diverting LSD Radix Sort is in the scan for large buckets. If we use Fast Radix instead of LSD Radix Sort then we have our proposed Diverting Radix Sort algorithm and the removal of counting directly offsets any cost of the single scanning pass, with even a single counting pass being comparable in terms of operations performed during the scan for buckets.

The results of this analysis suggest further considerations. Depending on compile-time characteristics, where to move the transition threshold would be adjusted to balance the cost of this extra pass before checking for diversion against the cost of a partial pass after the initial diversion.

While such discussion is outside the scope of this thesis, building such lookup tables via calculation (such as Table 10) takes only seconds.

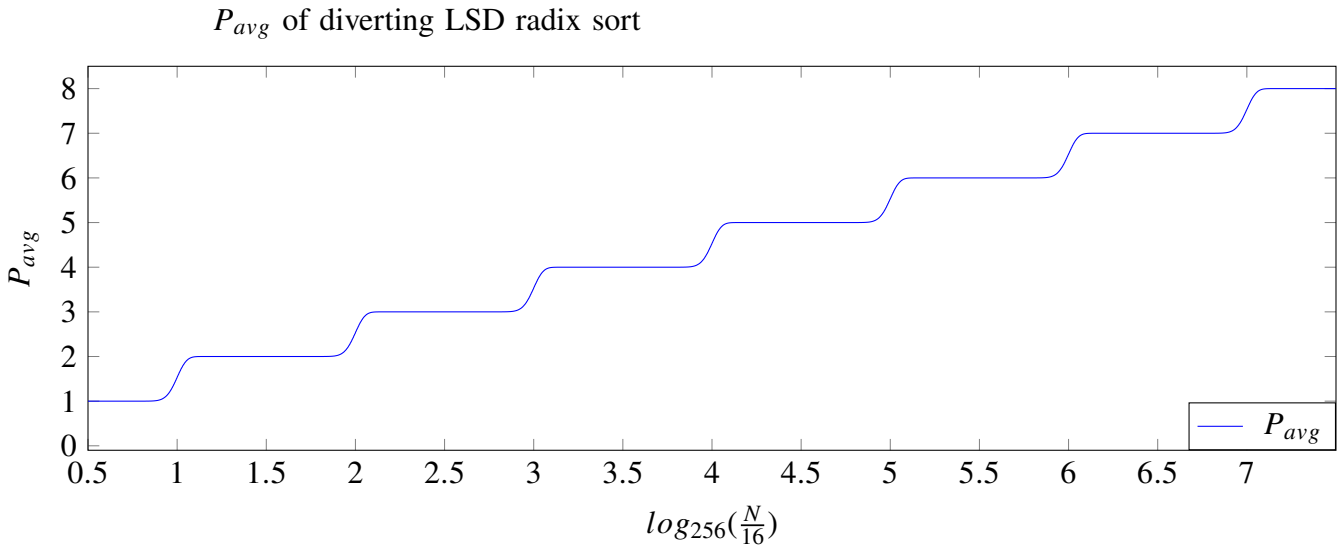

Figure 30: This figure represents the calculated average number of passes performed by diverting LSD Radix Sort given $DT = 16$, $M = 256$ and $P_{max} = 8$ using values from Table 10 to adjust P as N increases. A logarithmic scale is used as in Table 10 and highlights that this graph is composed of the best pieces of Figure 29.

Given Corollary 1 and the thresholds from Table 10, we see that Figure 30 demonstrates the average passes of Diverting Fast Radix for varying N. The conservative thresholds in Table 10 were chosen such that Figure 30 is also the graph of the average passes for a diverting MSD Radix Sort with a comparable diversion algorithm.

Chapter 6

Conclusion

"I'm glad my genes for cynicism as well as my genes for computers have been transmitted."
–(Larry Thiel, personal communication, February 1, 2021)

Sorting consumes a major proportion of computing cycles, so improvements in sorting algorithms and implementations address a key concern in today's age of Big Data. This thesis looks at improvements to radix sort, both in terms of algorithm and implementation, with a focus on algorithms, as we believe algorithm improvements have *"more bang for the buck"* especially over the long term. Specifically, the thesis outlines our contribution to sorting when "keys are known to be randomly distributed with respect to some continuous numerical distribution" [Knuth, 1998, p.5]. Specifically, we will consider the sorting of records whose keys are accessible and whose keys can be broken into a fixed number of digits. While the implementations developed for this thesis sort unsigned integer types, signed integers and floating-point standards exist that are known to work well with radix sorts. Using stochastic analysis will identify reliable and measurable improvements which will be demonstrated in our empirical results.

This thesis presents three new algorithms: Algorithm 6, Fast Radix; Algorithm 8, a diverting least significant digit (LSD) radix sort algorithm that performs just as many passes as a diverting most significant digit (MSD) radix sort; and Algorithm 9, the Diverting Fast Radix algorithm.

These algorithms offer significant average-case improvement on radix sorts given the most commonly used input distribution used in radix sort literature. An exact mathematical model is

developed for the difference in performance between Fast Radix and traditional least-significant digit (LSD) radix sort. We also develop an exact mathematical model of how the standard technique of diversion reduces passes of the traditional diverting most-significant digit (MSD) radix sorts, and then show that a diverting LSD radix sort is possible. The model of passes performed for the presented LSD radix sort is shown to be identical to that of diverting MSD radix sorts. Diverting Fast Radix is presented as the combined algorithm. Experimental results of the implementation of these theoretical improvements is shown to match the models.

The approaches considered in this thesis are orthogonal to most recent research on technical implementation where the promise of linearly scaling performance, advances is competitive cache sensitivity and scalability across processing cores is forefront. Our analysis of the algorithm improvements show that less work can be performed on average without needing technical improvements to do so. The goal is that other researchers apply their future technical improvements to Diverting Fast Radix as it is a better starting point than other radix sort variants that currently get attention.

The survey of literature around radix sorts shows an increase in activity and a focus on technical optimizations, particularly around cache and TLB efficiency that is often identified as the main failing of radix sorts, and parallelization which is ever more relevant with the ubiquity of multi-core computing. These improvements are applied to either traditional LSD radix sorts or diverting MSD radix sorts without considering that these established algorithms, themselves, may be holding back performance.

6.1 Contributions

The work presented in this thesis creates a new baseline for radix sort algorithm research. Even before popular optimizations are applied, Diverting Fast Radix, shows a consistent improvement in

execution time over optimized std:sort once past a threshold in the range of $N \sim 10^3$, running several times as fast. Even after the impact of cache and TLB misses, described in LaMarca and Ladner [1999], Diverting Fast Radix is on the order of twice as fast as std:sort for inputs as large as $N = 10^9$. When compared to the current best cache-optimized diverting MSD radix sort, RADULS2[Kokot et al., 2018], Diverting Fast Radix is comparable or faster for $N \leq 10^6$, falling behind only on larger inputs due to the absence of cache and TLB optimization.

The work in Thiel et al. [2016] has been extended to show that all but one counting pass can be omitted by our Fast Radix algorithm without significant cost. The cost savings from the omission of counting passes via estimation in Fast Radix has been modeled mathematically in the average case, with the predicted improvement being borne out in implementation showing it to be 50 to 100% faster than Simple Fast Radix on $N \geq 10^7$.

Independently of the removal of counting passes, this thesis demonstrates that diversion can be consistently applied to LSD radix sorts to reduce passes in a manner consistent with how diversion reduces passes in MSD radix sorts. The number of passes performed has been modeled for both diverting MSD and LSD radix sorts. The models of both these algorithms have been shown to be equivalent, accurately representing actual passes performed in implemented algorithms. This is all-the-more significant as it appears diverting LSD radix sorts have never been more than superficially explored in the literature.

The RADULS2 code measures in the 10s of thousands of lines of source code and optimized binaries approach 10MB in size. The reference implementations of the algorithms presented in this thesis have source code that is more than an order of magnitudes smaller and optimized binaries are more than two orders of magnitude smaller. The code is small enough to fit in an appendix of this thesis and has also been made available on a public GitHub repo (https://github.com/ramou/dfr/)

To summarize, this thesis makes the following contributions to the field:

- The Fast Radix algorithm, an LSD radix sort that omits all but one counting pass
- A diverting LSD radix sort algorithm that performs just as many passes as a diverting MSD radix sort
- The Diverting Fast Radix algorithm, Fast Radix using a diverting LSD radix sort
- A mathematical model of the average case overflow when estimating counts in Fast Radix
- A mathematical model of the average case number of passes in diverting MSD radix sort
- A mathematical model of the average case number of passes in diverting LSD radix sort
- A proof that the mathematical models of diverting LSD and MSD radix sorts are equivalent
- Public implementations of the three original algorithms presented in this thesis

6.2 Limitations

While it might otherwise go without mention, though this work demonstrates that some counting passes may be reduced and pass estimation eliminates entire passes, the overall algorithm remains asymptotically linear.

The work in this thesis serves to provide a baseline model for future research about radix sorts. The work itself does not assert the null hypothesis for the proposed algorithm's impact on all distributions. However, it does provide a baseline by which the null hypothesis can be examined and likely discarded as models for other distributions are developed and shown to be different from the one proposed herein. LaMarca and Ladner remarks that "the choice of randomly chosen data yields a fair comparison among the algorithms" in considering records drawn randomly from a uniform distribution, but this does not deny the value of analysis with other distributions.

The current work models sorting random inputs drawn from a uniform distribution. The impact on the algorithms' performance described herein, given other distributions or pathological cases, will yield different results. Notwithstanding, many non-uniform distributions impact bucket estimations most noticeably on the most significant digits, which are not estimated. Even in cases where inputs are such that estimates are entirely wrong on many or all digits and the algorithm would deal all records into a single bucket, the impact causes performance to degenerate effectively into that of an LSD radix sort. If all but M records overflow, then the dealing pass becomes a glorified counting pass. The algorithm counts each of the $N - M$ digits as they overflow, and the overflow processing becomes a dealing pass, thus limiting performance impact compared to a traditional LSD radix sort.

We have not modelled the impact of distribution on pass estimation in this work, but it tentatively appears limited compared to the impact on diverting MSD radix sorts. The current model cannot estimate more passes than required by a corresponding diverting MSD radix sort, and other distributions similarly impact diverting MSD radix sort average passes negatively. A degenerate case where Diverting Fast Radix estimated one pass and then recursed would emulate a diverting MSD radix sort but also supposes a trivially small input size to estimate that only a single pass is required. Diverting Fast Radix should not perform significantly worse than a diverting MSD radix sort with these distributions. Without a model, we cannot predict the exact difference in performance caused by different distributions and the performance gains reported here. The results presented here do not imply a similar impact for various structured inputs or random inputs drawn from non-uniform distributions.

Aside from focusing on random inputs drawn from uniform distributions, this thesis identifies several areas as out of scope. This thesis's results do not address comparison-based sorting, save in the use of comparison-based algorithms for diversion; we do not show the results in this thesis as having any impact on the nature of comparison-based sorting. Our implementation focuses on sorting integers and our results were tested specifically with 64-bit integers, but we note that "[i]nteger sorting is not an exotic special case, but in fact is one of the sorting problems most

frequently encountered" and that "the IEEE 754 floating-point standard was designed specifically to facilitate the sorting of floating-point numbers by means of integer-sorting subroutines"[Andersson et al., 1998].The algorithms developed in this thesis do not replace technical optimizations related to parallel processing, cache/TLB sensitivity or any other technical work researched elsewhere but instead work in conjunction with advances in these areas. We demonstrate the calculation of pass estimation thresholds in Chapter 5. We do not suggest that any diversion or pass thresholds hold special significance; these values are machine-specific. Their calculation is trivially fast enough that a user could perform them at compile-time instead of run-time. Lastly, we note that this work describes an algorithm designed for in-memory application, with input sizes in not exceeding $N = 10^{10}$, and even less on machines with smaller amounts of RAM; as with other technical optimizations, one should take care to implement approaches that preserve locality of reference to expand this algorithm beyond in-memory usage.

6.3 Impact

At the time of the writing, the two established areas of impact are at Texas A & M University and a Rust developer in France.

A doctoral thesis by Sarker Tanzi Ahmed references our early unpublished Fast Radix work from 2015 in their section on "High-Performance" sorting[Ahmed, 2016]. In 2018, Di Xiao requested source code from our Graph Chi publication to include our 2016 work in a performance survey; in return, they shared a table of its results compared to other work in the field, showing the previously cited RADULS2 being faster. While ultimately Di Xiao did not publish the survey, in 2020, a copy of the 2018 results appear in Hanel et al. [2020] using Simple Fast Radix as an example of an out-of-place "LSB radix" sort. The common thread is Texas A & M University, with Dmitri Loguinov acting as the chair of Sarker Tanzi Ahmed's thesis committee while also being an author along with Di Xiao in Hanel et al. [2020].

In late 2019, Axelle Piot found a bug in the Diverting Fast Radix code published on GitHub. We quickly patched the bug and exchanged some email regarding their final, very positive results. Nearly a year later, at the end of 2020, he published a Rust container for his Voracious Sort[Piot, 2020] whose documentation referenced both the early unpublished work used by Sarker Tanzi Ahmed's doctoral thesis and the Diverting Fast Radix GitHub. Aside from Voracious Sort, he also implemented a few other algorithms in this container, including Ska Sort, Simple Fast Radix and a diverting LSD radix sort. He implemented his Simple Fast Radix as described in the unpublished work and based on a GitHub repository that appears unrelated to this work (mistakenly cited as mine). He implemented his diverting LSD radix sort based on our Diverting Fast Radix code, not using count estimation, citing concerns regarding non-uniform distributions, estimating passes before diversion differently and using a variable radix. Still, this implementation capitalized on the diversion component of this work. They used diverting LSD radix sort instead of Voracious Sort for unsigned integers up to $N = 10^5$, irrespective of data distribution.

6.4 Future Work

The most obvious future work is to apply cache optimizations to Diverting Fast Radix. According to LaMarca and Ladner [1999], counting while doing a dealing pass impacts cache/TLB performance, but the bulk of the problem still lies with the random bucket locations. While Diverting Fast Radix reduces the interference from the counting, applying approaches from Rahman and Raman [2001] should remove a significant amount of the remaining performance hit. Kokot et al. [2018] discusses improvements on the order of doubling speed on regular passes, and a similar improvement on Diverting Fast Radix's regular passes should allow Diverting Fast Radix to end up on the order of 25% faster than RADULS2 when $N = 10^9$. While RADULS2 used what would be described as a "block" approach to cache efficiency, Rahman and Raman [2001] also described other approaches that would be worth consideration.

When working with *real data* in Thiel et al. [2016], some effort was made to deal with standard input structural issues, such as high-order bits being unused. During the development of Diverting Fast Radix, similar considerations were made and a handful of techniques were attempted to deal with this. An important future work would be to more systematically consider sampling, and other approaches to best deal with the impact of structure on both counting estimates and the estimates of passes in the initial Fast Radix component of Diverting Fast Radix.

The mathematical models in this thesis were for an input pulled from a uniform distribution. While the impact of normal distributions on diverting MSD radix sorts are significant, it would seem useful for future work to consider how closely Diverting Fast Radix could be made to approximate the number of passes or how to estimate when it is more expedient to do more initial passes to save on the extra cost of a significant portion of the input not diverting. Research on sampling for normal distributions would also benefit count estimations. It would potentially be informative to see whether the impact of a normal distribution on digits of different significance could play a role in either count estimations or for performance reasons.

Both Kokot et al. [2018] and Hanel et al. [2020] used sorting networks for their diversion algorithm. At least with Kokot, this had a significant impact on the size of compiled code, so much so that it adversely impact the sorting of inputs that were orders of magnitude smaller than the code being loaded to sort them. At the same time, our estimates on the performance impact of the diversion pass on a cache-sensitive diverting radix sort place it at between 10 and 20% of overall time used at $N = 10^9$, and approaching 50% for much smaller inputs where you effectively only have one or two regular passes. To that end, a study of diversion strategies, and potentially ways to combine them with the Diverting Fast Radix scanning pass might yield very tangible performance benefits.

6.5 Postscript

After submitting this thesis, Larry Thiel[Thiel, 2021] produced an implementation of Diverting Fast Radix with cache-sensitive writes: ThielSort. A summary of the results is shown in Table 11, normalized to `std::sort`. These results validate our theory that Diverting Fast Radix is a suitable baseline for future radix sort research into technical optimizations, with ThielSort sorting records 50% faster at $N = 10^9$ and otherwise outperforming RADULS2 at all measured values.

ThielSort uses intermediary buckets to achieve more locality of reference before block writes to actual buckets. This approach achieves the block-based class of cache/TLB sensitivity described in Rahman and Raman [2001], whereas RADULS2 uses non-temporal writes via AVX/AVX2 (dependent on what is available at compile-time) for the same purpose. We note that the overhead involved is noticeable at smaller N, where Diverting Fast Radix outperforms both RADULS2 and ThielSort as cache sensitivity is not a noticeable issue and Diverting Fast Radix has no penalty when it deals directly into buckets.

N	Diverting Fast Radix	RADULS2	ThielSort
100	800.00%	388.89%	2922.22%
1000	107.07%	2520.20%	992.93%
10000	27.53%	312.26%	85.04%
100000	27.45%	44.49%	29.30%
1000000	30.60%	33.43%	23.93%
10000000	52.57%	36.62%	24.15%
100000000	51.65%	22.36%	20.84%
1000000000	47.08%	29.17%	20.07%

Table 11: Timing comparison of Diverting Fast Radix, RADULS2 and ThielSort normalized to `std::sort`, using random 64-bit unsigned integers from a uniform distribution.

Bibliography

[Ahmed, 2016] Sarker Tanzir Ahmed. *Analysis, Modeling, and Algorithms for Scalable Web Crawling*. PhD thesis, Texas A & M University, 2016.

[Al-Badarneh and El-Aker, 2004] Amer Al-Badarneh and Fouad El-Aker. Efficient adaptive inplace radix sorting. *Informatica*, 15(3):295 – 302, 2004. ISSN 08684952. URL https://lib-ezproxy.concordia.ca/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=18106146&site=ehost-live&scope=site.

[Andersson and Nilsson, 1994] Arne Andersson and Stefan Nilsson. A new efficient radix sort. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science, 1994*, pages 714–721, 1994. doi: 10.1109/SFCS.1994.365721.

[Andersson et al., 1998] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

[Backstrom et al., 2006] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 44–54. ACM, 2006.

[Bentley and Sedgewick, 1997] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. ISBN 0-89871-390-0.

[Berg, 2014] Sven Berg. *URN Models*. American Cancer Society, 2014. ISBN 9781118445112. doi: 10.1002/9781118445112.stat03042. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat03042.

[Boldi and Vigna, 2004] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[Boldi et al., 2008] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Graph. *SIGIR Forum*, 42(2):33–38, 2008.

[Boldi et al., 2011] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[Chakraborty et al., 2007] Soubhik Chakraborty, Suman Kumar Sourabh, Mausumi Bose, and Kumar Sushant. Replacement sort revisited: The "gold standard" unearthed! *Applied Mathematics and Computation*, 189(1):384–394, 2007.

[Chen and Reif, 1993] Shenfeng Chen and John H. Reif. Using difficulty of prediction to decrease computation: fast sort, priority queue and convex hull on entropy bounded inputs. In *Proceedings, 34th Annual Symposium on Foundations of Computer Science, 1993.*, pages 104–112, 1993. doi: 10.1109/SFCS.1993.366877.

[Cormen et al., 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL `http://mitpress.mit.edu/books/introduction-algorithms`.

[Delorme et al., 2013] Michael C. Delorme, Tarek S. Abdelrahman, and Chengyan Zhao. Parallel radix sort on the amd fusion accelerated processing unit. In *2013 42nd International Conference on Parallel Processing*, pages 339–348, 2013. doi: 10.1109/ICPP.2013.43.

[Estivill-Castro and Wood, 1992] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992. ISSN 0360-0300. doi: 10.1145/146370.146381. URL `http://doi.acm.org/10.1145/146370.146381`.

[Freedman et al., 2007] David Freedman, Robert Pisani, and Roger Purves. *Statistics: Fourth International Student Edition*. International student edition. W.W. Norton & Company, 2007. ISBN 9780393930436. URL https://books.google.ca/books?id=mviJQgAACAAJ.

[Friend, 1956] Edward H. Friend. Sorting on electronic computer systems. *J. ACM*, 3(3):134168, July 1956. ISSN 0004-5411. doi: 10.1145/320831.320833. URL https://doi.org/10.1145/320831.320833.

[Graham et al., 1994] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1994. ISBN 0201558025.

[Hanel et al., 2020] Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov. Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 623–638, 2020.

[Jiménez-González et al., 2003] Daniel Jiménez-González, E Guinovart, J-L Larriba-Pey, and Juan J Navarro. CC-Radix: a cache conscious sorting based on Radix sort. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 101–108, 2003. doi: 10.1109/EMPDP.2003.1183573.

[Kärkkäinen and Rantala, 2009] Juha Kärkkäinen and Tommi Rantala. Engineering Radix Sort for Strings. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *String Processing and Information Retrieval*, volume 5280 of *Lecture Notes in Computer Science*, pages 3–14. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-89096-6. doi: 10.1007/978-3-540-89097-3_3. URL http://dx.doi.org/10.1007/978-3-540-89097-3_3.

[Knuth, 1997] Donald E. Knuth. *The Art of Computer Programming: Fundamental algorithms*. Number v. 1 in Addison-Wesley series in computer science and information processing. Addison-Wesley, 1997. ISBN 9780201896831. URL https://books.google.ca/books?id=5oJQAAAAMAAJ.

[Knuth, 1998] Donald E. Knuth. *The Art of Computer Programming*. Number v. 3 in Addison-Wesley series in computer science and information processing. Addison-Wesley, 1998. ISBN 9780201896855. URL https://books.google.ca/books?id=CH1GAAAAYAAJ.

[Kokot et al., 2017] Marek Kokot, Sebastian Deorowicz, and Agnieszka Debudaj-Grabysz. Sorting data on ultra-large scale with raduls. In Stanisław Kozielski, Dariusz Mrozek, Paweł Kasprowski, Bożena Małysiak-Mrozek, and Daniel Kostrzewa, editors, *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, pages 235–245, Cham, 2017. Springer International Publishing. ISBN 978-3-319-58274-0.

[Kokot et al., 2018] Marek Kokot, Sebastian Deorowicz, and Maciej Długosz. Even faster sorting of (not only) integers. In Aleksandra Gruca, Tadeusz Czachórski, Katarzyna Harezlak, Stanisław Kozielski, and Agnieszka Piotrowska, editors, *Man-Machine Interactions 5*, pages 481–491. Springer, Cham, 2018. doi: 10.1007/978-3-319-67792-7_47. URL https://doi.org/10.1007/978-3-319-67792-7_47.

[Kumar, 2019] Ravin Kumar. Modified counting sort. In P. K. Kapur, Yury Klochkov, Ajit Kumar Verma, and Gurinder Singh, editors, *System Performance and Management Analytics*, pages 251–258. Springer Singapore, Singapore, 2019. ISBN 978-981-10-7323-6. doi: 10.1007/978-981-10-7323-6_21. URL https://doi.org/10.1007/978-981-10-7323-6_21.

[Kwak et al., 2010] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: http://doi.acm.org/10.1145/1772690.1772751.

[Kyrola et al., 2012] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[LaMarca and Ladner, 1999] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66 – 104, 1999. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1998.0985. URL http://www.sciencedirect.com/science/article/pii/S0196677498909853.

[Lee et al., 2002] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned parallel radix sort. *Journal of Parallel and Distributed Computing*, 62(4):656 – 668, 2002. ISSN 0743-7315. doi: https://doi.org/10.1006/jpdc.2001.1808. URL http://www.sciencedirect.com/science/article/pii/S0743731501918088.

[Leskovec et al., 2008] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR*, abs/0810.1355, 2008. URL http://arxiv.org/abs/0810.1355.

[Lorin, 1975] Harold Lorin. *Sorting and Sort Systems (The Systems Programming Series)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1975. ISBN 0201144530.

[MacLaren, 1966] M. Donald MacLaren. Internal sorting by radix plus sifting. *J. ACM*, 13(3):404–411, July 1966. ISSN 0004-5411. doi: 10.1145/321341.321349. URL http://doi.acm.org/10.1145/321341.321349.

[Mannila, 1985] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *Computers, IEEE Transactions on*, C-34(4):318–325, April 1985. ISSN 0018-9340. doi: 10.1109/TC.1985.5009382.

[Maus, 2002] Arne Maus. Arl, a faster in-place, cache friendly sorting algorithm. In *Norsk Informatik Konferranse NIK2002*, pages 85–95, 2002.

[Maus, 2019] Arne Maus. Radixinsert, a much faster stable algorithm for sorting floating-point numbers. In *Norsk IKT-Konferanse for Forskning og Utdanning*, 2019.

[Mcllroy et al., 1993] Peter M Mcllroy, Keith Bostic, and Malcolm Douglas Mcllroy. Engineering Radix Sort. *Computing Systems*, 6(1):5–27, 1993.

[Nilsson, 2000] Stefan Nilsson. The Fastest Sorting Algorithm? *Dr. Dobb's journal (1989)*, 25(4):38–+, 2000.

[Paige and Tarjan, 1987] Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[Petersson and Moffat, 1995] Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995.

[Piot, 2020] Axelle Piot. lakwet/voracious_sort, Nov 2020. URL `https://github.com/lakwet/voracious_sort`.

[Polychroniou et al., 2015] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2747645. URL `http://doi.acm.org/10.1145/2723372.2747645`.

[Rahman and Raman, 2001] Naila Rahman and Rajeev Raman. Adapting radix sort to the memory hierarchy. *J. Exp. Algorithmics*, 6, December 2001. ISSN 1084-6654. doi: 10.1145/945394.945401. URL `http://doi.acm.org/10.1145/945394.945401`.

[Satish et al., 2009] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2009. doi: 10.1109/IPDPS.2009.5161005.

[Sedgewick, 1977] Robert Sedgewick. Quicksort with Equal Keys. *SIAM Journal on Computing*, 6(2): 240–267, 1977. doi: 10.1137/0206018. URL `http://epubs.siam.org/doi/abs/10.1137/0206018`.

[Sedgewick, 1998] Robert Sedgewick. *Algorithms in C++*. Number v. 1-2 in Algorithms in C++. Addison-Wesley, 1998. ISBN 9780201350883. URL `https://books.google.ca/books?id=331YAAAAYAAJ`.

[Sedgewick and Flajolet, 2013] Robert Sedgewick and Phillippe Flajolet. *An Introduction to the Analysis of Algorithms*. Pearson Education, 2nd edition, 2013. ISBN 978-0-321-90575-8. URL http://it-ebooks.info/book/1608/.

[Sedgewick and Wayne, 2011] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011. ISBN 978-0-321-57351-3.

[Shun et al., 2012] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 68–70. ACM, 2012.

[Singh, 2012] Chakraborty Singh. Partition sort revisited: Reconfirming the robustness in average case and much more! *IJSEA*, 2(1), 2012. URL http://www.airccse.org/journal/ijcsea/papers/2112ijcsea03.pdf.

[Singh and Chakraborty, 2012a] Niraj Kumar Singh and Soubhik Chakraborty. A statistical approach to the relative performance analysis of sorting algorithms. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, page 5762, New York, NY, USA, 2012a. Association for Computing Machinery. ISBN 9781450313100. doi: 10.1145/2393216.2393227. URL https://doi.org/10.1145/2393216.2393227.

[Singh and Chakraborty, 2012b] Niraj Kumar Singh and Soubhik Chakraborty. Smart sort: Design and analysis of a fast, efficient and robust comparison based internal sort algorithm. *CoRR*, abs/1204.5083, 2012b. URL http://dblp.uni-trier.de/db/journals/corr/corr1204.html#abs-1204-5083.

[Singh and Chakraborty, 2011] NirajKumar Singh and Soubhik Chakraborty. Partition sort and its empirical analysis. In VinuV Das and Nessy Thankachan, editors, *Computational Intelligence and Information Technology*, volume 250 of *Communications in Computer and Information Science*, pages 340–346. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25733-9. doi: 10.1007/978-3-642-25734-6_51. URL http://dx.doi.org/10.1007/978-3-642-25734-6_51.

[Skarupke, 2017] Malte Skarupke. Ska Sort. https://github.com/skarupke/ska_sort, 2017.

[Skiena, 1988] Steven S Skiena. Encroaching lists as a measure of presortedness. *BIT Numerical Mathematics*, 28(4):775–784, 1988.

[Sourabh and Chakraborty, 2008] Suman Kumar Sourabh and Soubhik Chakraborty. How robust is quicksort average complexity? *CoRR*, abs/0811.4376, 2008. URL http://dblp.uni-trier.de/db/journals/corr/corr0811.html#abs-0811-4376.

[Stevens, 1939] Stanley Smith Stevens. On the problem of scales for the measurement of psychological magnitudes. *Journal of Unified Science*, 9:94–99, 1939.

[Stevens, 1946] Stanley Smith Stevens. On the Theory of Scales of Measurement. *Science*, 103(2684): 677–680, 1946. ISSN 00368075, 10959203. URL http://www.jstor.org/stable/1671815.

[Thiel, 2021] Larry Thiel. Private Communication, 2021.

[Thiel, 2019] Stuart Thiel. Diverting fast radix, Nov 2019. URL https://github.com/ramou/dfr.

[Thiel et al., 2016] Stuart Thiel, Greg Butler, and Larry Thiel. Improving GraphChi for Large Graph Processing: Fast Radix Sort in Pre-Processing. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, pages 135–141, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4118-9. doi: 10.1145/2938503.2938554. URL http://doi.acm.org/10.1145/2938503.2938554.

[Williamson et al., 2009] Patricia Pepple Williamson, Darcy P. Mays, Ghidewon Abay Asmerom, and Yingying Yang. Revisiting the classical occupancy problem. *The American Statistician*, 63(4): 356–360, 2009. doi: 10.1198/tast.2009.08104. URL https://doi.org/10.1198/tast.2009.08104.

[Yang and Leskovec, 2012] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities based on Ground-truth. *CoRR*, abs/1205.6233, 2012. URL http://arxiv.org/abs/1205.6233.