UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

COMBINATORIAL ALGORITHMS IN THE APPROXIMATE COMPUTING PARADIGM

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

BY

Aditya Narasimhan
Norman, Oklahoma
2023

COMBINATORIAL ALGORITHMS IN THE APPROXIMATE COMPUTING PARADIGM

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Sridhar Radhakrishnan (Chair)

Dr. Charles Nicholson

Dr. Mohammed Atiquzzaman

Dr. Qi Cheng

Acknowledgements

I would like to take this opportunity to express my heartfelt gratitude to the following individuals, without their support and encouragement, this Ph.D. dissertation would not have been possible.

First and foremost, I express my sincere gratitude to my committee chair, advisor, co-author, and mentor for research and life, Professor Sridhar Radhakrishnan. His unwavering support, invaluable guidance, and encouragement throughout my research journey have been instrumental in shaping my research and developing my skills.

I would like to extend my heartfelt love and thanks to my beloved wife, Uma Manognya Kaipa, for her unconditional love, unwavering support, and the sacrifices she made during this journey. Her encouragement and motivation kept me going during the challenging times.

My family, including my father Dr. Narasimhan, mother Hema, sister Manasa, brother-in-law Kaushik, and dear niece Tara, and my wife's parents, Sreenivas and Jayshree, deserve a special mention for their love, support, and encouragement. They all have been a pillar of strength and a source of motivation for me throughout my academic journey.

I am grateful to my roommate and fellow graduate, Sudhindra Gopal Krishna, for his constant support and motivation. His presence has been a source of

inspiration, and his insights have helped me throughout my research.

I express my gratitude to Dr. C R Subramanian for his valuable help and support in guiding me through my research. His insights and inputs have been of great assistance in shaping the trajectory of my study.

I would like to extend my appreciation to my committee members for their invaluable feedback, insightful suggestions, and constructive critiques throughout the entire process. Their expertise and perspectives have played a pivotal role in enhancing the quality and refinement of my research.

Furthermore, I am grateful to my fellow lab mates, namely Ashesh Gaur and Zuyuan Zhang, whose support and assistance have made this journey not only productive but also enjoyable.

I am thankful to Nicola Manas, Virginie Perez Woods, Philip Johnson, and Annettee Johnson for all their timely help and guidance as the people who have helped me navigate the CS and DSA program.

Once again, I express my gratitude to all the family and friends along with the individuals mentioned above for their invaluable support and encouragement throughout my research and academic journey.

Abstract

Data-intensive computing has led to the emergence of data centers with massive processor counts and main memory sizes. However, the demand for shared resources has surpassed their capacity, resulting in increased costs and limited access. Commodity hardware, although accessible, has limited computational resources. This poses a challenge when performing computationally intensive tasks with large amounts of data on systems with restricted memory. To address these issues, Approximate Computing offers a solution by allowing selective solution approximation, leading to improved resource efficiency.

This dissertation focuses on the trade-off between output quality and computational resource usage in sorting and searching problems. It introduces the concept of Approximate Sorting, which aims to reduce resource usage while maintaining an accepted level of sorting quality. Quality metrics are defined to assess the "sortedness" of approximately sorted arrays. The dissertation also proposes a general framework for incorporating approximate computing into sorting algorithms, presenting an algorithm for approximate sorting with guaranteed upper bounds. The algorithms operate under a constraint on the number of comparisons performed.

The dissertation continues to explore searching algorithms, specifically binary search algorithms on approximately sorted arrays. It addresses cases where met-

rics are given for the input array and cases where metrics are not available. Efficient and optimal algorithms are developed for multidimensional range searches and catalog searches on approximately sorted input. The dissertation further proposes algorithms that analyze patterns in input order to optimize sorting. These algorithms identify underlying patterns and sequences, facilitating faster sorting approaches.

Additionally, the dissertation discusses the growing popularity of approximate computing in the field of High-Performance Computing (HPC). It presents a novel approach to comparison-based sorting by incorporating parallel approximate computing. The dissertation also proposes algorithms for various queries on approximately sorted arrays, such as determining the rank or position of an element. The time complexity of these querying algorithms is proportional to the input metric.

The dissertation concludes by emphasizing the wide range of applications for sorting and searching algorithms. In the context of packet classification in router buffers, approximate sorting offers advantages by reducing the time-consuming sorting step. By capping the number of comparisons, approximate sorting becomes a practical solution for efficiently handling the large volume of incoming packets.

This dissertation contributes to the field of approximate computing by addressing resource limitations and cost issues in data-intensive computing. It provides insights into approximate sorting and searching algorithms, and their application in various domains, offering a valuable contribution to the advancement of efficient, scalable, and accessible data processing.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Donald Knuth, the renowned computer scientist, once described sorting as follows: "Sorting is a fundamental and pervasive operation in computer science, and its study has spawned a rich and diverse field of research." These words capture the essence of sorting, highlighting its significance and the wide range of approaches that have been developed to solve this fundamental problem.

In the domain of sorting algorithms, comparison-based sorting techniques have gained substantial attention due to their simplicity and generality. Comparison-based sorting algorithms compare elements of a given sequence using pairwise comparisons, and based on the outcomes of these comparisons, they rearrange the elements to achieve the desired order. These algorithms offer a flexible and generic framework for sorting, making them applicable to various problem domains.

A single comparison, at its core, is an operation that determines the relative ordering of two elements. It takes two elements as input and outputs either a "less

than," "equal to," or "greater than" relationship between them. This elementary operation serves as the building block for a wide range of sorting algorithms, and its efficient implementation is crucial for achieving optimal sorting performance.

In the context of comparison-based sorting, it is essential to understand the various types of comparisons that can be performed. The most common type is the key-based comparison, where the elements are compared based on a specific attribute or key associated with each element. For example, when sorting a list of integers, the key-based comparison could involve comparing the numerical values of the integers.

Another type of comparison is lexicographic comparison, which is commonly used when sorting strings or sequences of characters. Lexicographic comparisons involve examining the characters of two elements from left to right and comparing them based on their relative positions in the lexicographic order. This type of comparison plays a crucial role in a variety of applications, including natural language processing, text processing, and dictionary-based operations.

Additionally, there are specialized comparisons that take advantage of the properties and structures of specific data types. For instance, interval comparisons are used when sorting intervals or ranges, where the ordering is determined based on the start and end points of each interval. Similarly, geometric comparisons are employed in sorting geometric objects such as points, lines, or polygons, where the order is determined by their spatial relationships.

Understanding the different types of comparisons and their implications is essential for designing efficient and effective comparison-based sorting algorithms. By leveraging the characteristics of the data being sorted, researchers and practitioners can develop tailored approaches that optimize the sorting process for specific problem domains.

In this dissertation, we delve into the field of combinatorial algorithms in the approximate computing paradigm, focusing on the application of these algorithms to sorting problems. We explore various techniques, strategies, and optimizations that can enhance the performance of comparison-based sorting algorithms in scenarios where approximate solutions are acceptable. Through empirical evaluations and theoretical analysis, we aim to provide valuable insights into the design and implementation of efficient combinatorial algorithms, ultimately contributing to the advancement of approximate computing and its applications in sorting paradigms.

By combining the power of combinatorial algorithms with the flexibility of the approximate computing paradigm, we anticipate opening new avenues for efficient sorting in domains where strict precision can be relaxed in favor of faster and more scalable solutions. This research strives to push the boundaries of sorting algorithms, offering novel perspectives and approaches that can significantly impact the field of computer science and its practical applications.

1.2 Organization of the Dissertation

The dissertation first starts by explaining basic starting algorithms and the motivation behind why a comparison-bound sorting algorithm is warranted. We present a sorting algorithm that takes in the number of comparisons as a constraint (input) and performs sorting as explained in Chapter 3. In the same chapter, we continue to define the basic quality metrics that are used in the dissertation for proving various constraints and bounds for the algorithms that are defined. The chapter also gives a bound on all the metrics for the algorithm that is provided for sorting. The chapter ends with a proposed modified selection sort

algorithm that completes sorting an input approximately sorted array when one of the metrics is an input.

Chapter 4 in this dissertation gives algorithms for preprocessing a given random array of elements, that finds sequences and patterns. The first algorithm presented finds maximal increasing chains that are contiguous. There are optimization approaches mentioned that make this algorithm better with respect to the metrics that are mentioned. There are decreasing and monotonic subsequence pattern recognition algorithms as well provided. There are also non-consecutive increasing subsequence pattern recognition algorithms given along with local Optima finding algorithms as well.

Now that we have covered a lot of sorting and sequence recognition algorithms we jump into querying algorithms starting with search. In Chapter 5, The first algorithm that is provided is a modified binary search algorithm that precisely finds an element given an approximately sorted array with its metric. We continue to Define and provide bounds for an algorithm where the metric is not an input for the approximately sorted array but still be able to search the given array precisely for a query element.

In chapter 6 the dissertation continues to explore range search algorithms. given an approximately sorted array along with its metric algorithm for finding the number of elements within a given range and finding the exact elements that do fall within this given range are provided. These algorithms are given for a one-dimensional input, two-dimensional input, or multidimensional input. One of the important ideas that two-dimensional range search uses is the fractional cascading structure that is also performed for an approximately sorted order. The dissertation provides the construction of this fractional cascading structure along with its use in the 2D and multi-dimensional range search algorithms.

In Chapter 7, There are parallel algorithms that are provided to approximately start a given random set of elements. We also provide an algorithm to find the exact position of an element in an approximately sorted array given its rank. There are also algorithms defined that find the rank given the position of an element in the approximately sorted array. There are parallel multicore algorithms mentioned for each of these algorithms. The chapter ends with the Sequential and parallel multi-select approach that is performed on an approximately sorted array.

Chapter 8 of this dissertation heads into the application of approximate Computing and approximate starting in the software defined Network Realm. There are packet classification algorithms provided that make packet classification in a router speed up by approximately sorting the incoming packets before processing and classifying them into rule sets. We establish a relationship between the quality metrics and the decision tree approach that is used here. The comparisons are made between traditional decision tree approaches and an approximately sorted packet decision tree approach. Experiments prove the speed up and the searching that can be done on an approximately sorted set of packets in the buffer of the routers.

The last chapter of the dissertation concludes each of the above-mentioned chapters and ends with future work.

Chapter 2

Literature Survey

The literature survey chapter serves as a pivotal foundation for the comprehensive exploration of existing knowledge and research related to Approximate Sorting, Quality Metrics for an approximately sorted order, and also existing techniques that have been looked into. We also explore the applications where these techniques have already been applied.

The world is moving into an era where everything is centered around data. The large amount of data that is being generated by the second, needs to be processed to get various insights from them. We are at a point where the democratization of such resources has become a necessity to process these large amounts of data.

One way is to reduce the dependency on such resources. Specifically, for applications where a very accurate or precise solution is not necessary, one can leverage the quality of the solution/output for a reduction in the resources. This idea is called Approximate Computing **(Mittal, 2016)**.

One of the most fundamental problems in computer science is sorting. Broadly,

they can be classified into comparison-based and non-comparison-based sorting. It has been established **(Knuth, 1997)** that it suffices to perform $O(n \log n)$ pairwise comparisons **(Knuth, 1997)** to sort a given set of n unique elements in increasing or decreasing order.

We can either measure the sortedness of a given order by how sorted it is or by how disordered it is. The measure of disorder would be zero when it is fully sorted. For a survey of measures of disorder, see **(Estivill-Castro and Wood, 1992)**. The measure of number of inversions has been looked into by **(Knuth, 1997; Mannila, 1985; Wang et al., 2015; Disser and Kratsch, 2017)**. A closely related measure from **(Wang et al., 2015)** is the *Kendall Tau rank distance* defined as the number of pairwise adjacent transpositions of elements x_i and x_{i+1} that transforms one permutation to another. In **(Wang et al., 2015)**, the number of inversions $inv(\tau)$, the inversion vector and the Spearman's Footrule defines *inversion-ℓ_1 distance* between two given permutations. In **(Korba, 2018; Diaconis, 1988)**, Hamming distance is mentioned, which is also a well-studied metric that counts the number of elements that disagree between two permutations. Here, the two permutations are the approximately sorted output and a completely sorted version of the same array.

Related to the maximum displacement that we define in this dissertation, is the *Spearman's Footrule* which is defined in **(Diaconis and Graham, 1977)** and further analyzed in **(Wang et al., 2015; Giesen et al., 2006)**. It is the sum of the displacements for each x_i in the given permutation τ with respect to a sorted order.

Work has been done in the area of measuring the quality of an approximately sorted array as seen above. Our approach for searching an approximately sorted array using the maximum displacement (explained in later chapters) ends up with

a factor of complexity of $4L + 1$ number of comparisons where L is the quality metric. We point out that the term $4L+1$ can be bought down to $3L$ by following an approach that is taken by Disser and Kratch **(Disser and Kratsch, 2017)**, that starts to look for the element to be searched for from the center, looking between L index positions to the left and the right. Then based on the comparison made between these elements and the element to be found, we can either search the L left out elements from either the left or the right half.

We further study the various metrics that this paper defines and explains. One of the popular metrics that they use in their paper is k_{sum}, which is the sum of the displacements of an approximately sorted array. We were able to come up with a counter-example (worst case input) for theorem 5 from their paper that breaks the two-phase algorithm by having the k_{sum} too large. Below is an example array that we assume to prove this.

$$\{160, 150, 140, 130, 120, 110, 90, 80, 70, 60, 50, 40, 30, 20, 10\}$$

For the above example, the k_{sum} metric adds to 128 and the algorithm goes beyond the bounds for it to converge.

The paper also gives a proposition by comparing the k_{sum} and k_{inv}. The metric k_{inv} is the total number of inversions defined like how we define it. The authors also define total adjacent inversions as k_{ainv}. The relationship established between k_{sum} and k_{inv} is given by proposition 13 as,

$$k_{inv} \leq k_{sum} \leq 2k_{inv}$$

The proof given takes two elements that get inverted and compares the displacements of the two elements which adds to the number of inversions needed

to completely sort the order. This as a metric is not possible to be used in algorithms or for providing bounds as it does not give any positional information about the given order.

The work from **(Kaligosi et al., 2005)**, gives a variation of the multiselection problem to select $R = r_1, r_2, ...r_k$ rank for a given unsorted array. They provide upper and probabilistic bounds for the number of comparisons ($T(n, R)$) needed for the algorithm using the median of median pivot choosing strategy. **(Cardinal et al., 2010)** also provides a generalized version of this for the partial ordering production problem.

Range search is used in a variety of applications as well. In the field of computational geometry, it has a multitude of applications in database searching and geographical databases **(Tao et al., 2007; Zhang et al., 2005; Asano et al., 1985)**. The idea of fractional cascading was introduced in two parts, **(Chazelle and Guibas, 1986a)** where the data structure was defined and **(Chazelle and Guibas, 1986b)** where the various applications were listed, including finding the intersection of a polygonal path with a line, computing locus functions and the most popular problem of orthogonal range search. This idea was made into a dynamic implementation by **(Mehlhorn and Näher, 1990)**. Some real-world applications of fractional cascading are in the problem of incoming packet classification in routers **(Buddhikot et al., 1999; Lakshman and Stiliadis, 1998)**. They are also used in the data storage of sensor networks that are studied in **(Gao et al., 2004)**.

This lays the foundation for the work done in this dissertation. We continue to the next chapter where we start exploring the quality metrics and approximate sorting algorithms. More survey has been discussed in each of the chapters as well.

Chapter 3

Sorting

In this chapter, the concept of measuring the quality of the sortedness of the output order will be introduced along with specific metrics that we use with examples. Provided is a simple algorithm to approximately sort a given array with a constraint on the factor of number of comparisons that can be performed. With this algorithm, we establish upper bounds for each of the metrics. We also provide a modified selection sort approach that completes sorting an approximately sorted order, given the metric.

3.1 Quality Metrics

We use $rank_\tau(x_i)$ (shortly $r_\tau(x_i)$) to be the unique position occupied by the element x_i in the fully sorted version of the input τ.

The first measure is the number of inversions. It is defined as the number of pairs (x_i, x_j) $(i < j)$ in the output that is out of order, having $x_i > x_j$. Each of these out-of-order elements is referred to as an inversion. We can represent the

quality of the output order of elements using the number of inversions, $inv(\tau)$ **(Knuth, 1997; Mannila, 1985; Wang et al., 2015; Disser and Kratsch, 2017)**.

The second metric is an extension of the first metric where we see look at the farthest inversion among all inversions present in the given order. Maximum distance ($dis(\tau)$), is defined as $\max\{j - i : i < j,\ x_i > x_j\}$, if there is at least one inversion. This is zero if there are no inversions present.

We first define what is displacement before we jump to the third metric. Displacement ($dp(\pi, i)$) of an element x_i in a given permutation (π) is $|k - i|$, where $k = r_\pi(x_i)$. The third measure of disorder here is the maximum displacement ($md(\pi)$), which is defined as $\max_{1 \le i \le n} dp(\pi, i)$. When restricted to permutations over $\{1, \ldots, n\}$, the maximum displacement becomes the *Chebyshev distance* measure defined in **(Wang et al., 2015)**.

All the measures described above are measures of disorder, meaning they measure how far away the given array is from a completely sorted order. Hence, all measures will be 0, when the array is completely sorted. This would be the lower bound. The upper bound for the number of inversions (inv), would be $\binom{n}{2}$ in the worst-case scenario when there the array is reverse sorted. The upper bound for the distance (dis) would be in the scenario when the greatest element in the array is in the first index position or vice versa - the smallest element in the last index position. This is the same for the maximum displacement (md) as well.

Following is an example (figure 3.1) of an array (similar to the example from **(Wang et al., 2015)**), that describes the maximum displacement and distance

visually. We represent the number of inversions as a set:

$$\{(50, 40), (50, 20), (50, 30), (40, 20), (40, 30)\} = 5$$

$$\implies inv(A) = 5$$

As seen in the figure, the farthest inversion is between the index positions 1 and 4, so $dis(A) = 3$.

To calculate the maximum displacement, we first need to calculate the displacement for each of the values. This is listed in the figure. For example, the displacement for 50 is 3 since its current position is index position 1, but when sorted (A_S), its rank is 4. Thus, we calculate the maximum displacement as the maximum value amongst all displacements of each of the values, which would give us $md(A) = 3$.

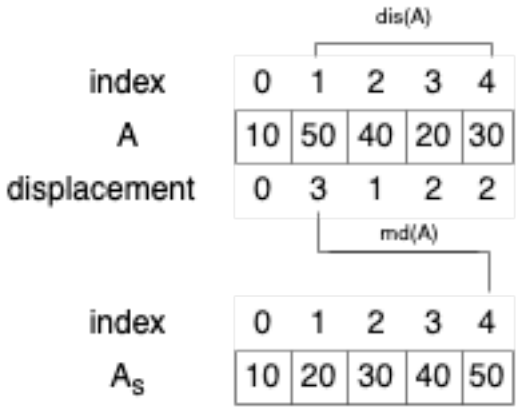


Figure 3.1: Example visualization of md and dis

The paradigm of approximate computing is taken to the space of sorting algorithms as the idea of approximate sorting (assumed to be increasing order without loss of generality).

3.2 Guaranteed bounds on various metrics for a given number of comparisons

For simplicity in describing the algorithm, we assume that n is a power of 2. Here, n denotes the number of packets in the input array $A[1..n]$. For $i \leq j$, we use $A[i, j]$ to denote the subarray $A[i..j]$ formed by the positions from i to j. Let $MedianRearrange(A[1..n])$ be a $O(n)$ time algorithm to find the median of $A[1..n]$ after which it rearranges $A[1..n]$ following (A2) given below. The median of $A[1..n]$ is the $\frac{n}{2}$-th smallest element of A. We assume the following about $MedianRearrange()$:

- A1: $MedianRearrange()$ is an in-place algorithm, that is, it does not use more than $O(1)$ additional storage.
- A2: When the execution of $MedianRearrange(A[i..j])$ (where $j - i + 1$ is even) is finished, it rearranges the packets of the array (buffer) $A[i..j]$ in such a way that any element x in the first half $A[i, \frac{j+i-1}{2}]$ and any element y in the second half $A[\frac{j+i+1}{2}, j]$ satisfy $x < y$.

Consider the following algorithm.

Algorithm 1: ParSort($A[1..n]$,D)

```
input: An array A[1..n], n is a power of 2.
for i ← 0 to D do
    for j ← 1 to 2^i do
        MedianRearrange(A[(j−1)n/2^i + 1, jn/2^i])
```

One can prove the following theorem/claim about ParSort().

Theorem 3.1. *Let π be the order of the packets in $A[1..n]$ when the execution of ParSort($A[], D$) ends. Then $inv(\pi) \leq \frac{n^2}{2^{D+2}}$ and $md(\pi), dis(\pi) \leq \frac{n}{2^{D+1}}$. Furthermore, PartSort() uses $O(Dn)$ comparisons to produce π.*

Proof. **Analysis of measure of disorder:**

An invocation of $MedianRearrange([A[i, j])$ uses at most $c(j - i + 1)$ comparisons for some absolute constant $c > 0$. There are well-known algorithms (such as the one based on the idea of "median of medians" discovered by **(Blum et al., 1973)**) that achieve this. The resulting array after the execution of $MedianRearrange()$ is that the smallest half of the packets in A are placed before the largest half of the packets in A. Using this fact, one can deduce the following claim.

For each $k \in \{0, 1, \ldots D\}$, the following is true: The entries of A just after the end of the iteration of the outer loop corresponding to $i = k$ are such that inversions can occur only between entries in each of the subarrays corresponding to $A[\frac{(j-1)n}{2^{k+1}} + 1, \frac{jn}{2^{k+1}}]$ for $j \in \{1, \ldots, 2^{k+1}\}$.

Proof. This claim can be established by inductive proof based on increasing values of i. The base case corresponding to $i = 0$ follows from the nature of MedianRearrange() explained above. For larger values of i, say $i = k \geq 1$, we know that induction up to $i \leq k - 1$ has established that after the first k rounds corresponding to $i = 0, \ldots k - 1$, we have obtained an array A in which the inductive hypothesis is true. By applying MedianRearrange() to each of these 2^k subarrays of size $\frac{n}{2^k}$ each, we further partition each of these subarrays into two smaller subarrays of size $\frac{n}{2^{k+1}}$ each and also satisfying the claim that inversions can occur only in these 2^{k+1} subarrays of size $\frac{n}{2^{k+1}}$ each. This establishes the claim.

Hence, after the end, for all $D + 1$ iterations of the outer loop, the array A is

such that inversions can occur only within the 2^{D+1} subarrays of size $\frac{n}{2^{D+1}}$ each. Therefore, the total number of inversions in π is at most

$$2^{D+1}\left(\frac{n(n-2^{D+1})}{2\cdot 2^{2D+2}}\right) \leq \frac{n^2}{2^{D+2}}.$$

This establishes the claimed upper bound on $inv(\pi)$. Furthermore, it follows immediately from the above claim that $md(\pi), dis(\pi) \leq \frac{n}{2^{D+1}}$.

Run-time analysis :

MedianRearrange($A[]$) uses at most cn comparisons (for some absolute constant $c > 0$) on inputs of size n. Therefore, the total number of comparisons used by ParSort($A[], D$) is bounded by

$$\sum_{0\leq i\leq D} 2^i \cdot c \cdot \frac{n}{2^i} \leq (D+1)cn.$$

3.3 Modified Selection Sort

In this section, we propose an algorithm to completely sort the given approximately sorted array with a corresponding L quality metric using a selection approach. We iterate through the array starting from the first position taking one position at a time to find the element that will have the rank as the index we currently are at.

Theorem 3.2. *Let π be the order of the elements when the execution of*

Algorithm 2: SortApproxArray($A[1..n], L$)

Input: An array $\tau = A[1..n]$ and $md(\tau) \leq L$
Output: Completely sorted array π
for $i \leftarrow 0$ ***to*** $n-1$ **do**
  $S \leftarrow \min\{i + 2L - 1, n\}$;
  $A[i]$ = find min within window $A[i : S]$ of length atmost $2L$
return $\pi = A$

SortApproxArray() ends. Then $inv(\pi), md(\pi), dis(\pi) \leq 0$*, using* $O(nL)$ *comparisons to produce* π*.*

The algorithm runs through every index position in the given approximately sorted order. With each element, we are able to determine the exact window $(i : S)$ within which the min element will occupy the current i^th index position. This window of elements is determined using the metric L that is given as an input. So the exact length of this window of elements is $2L$ until we reach the end of the array at which point the length becomes smaller as we approach the very last element. Finding the min element with this window $i : S$ will take $2L$ amount of comparisons for each index position. By repeatedly performing this operation of find the min element in each window for each index position (n), we will be able to completely sort the given approximately sorted array in $O(nL)$ number of comparisons.

We conclude this chapter as we have defined the basic quality metrics along with the ParSort sorting algorithm with which we establish the upper bounds for each of the metrics. We then provide a modified selection sort algorithm that takes in an approximately sorted order along with its metric and completely sorts it. The following chapter dives into analyzing the input array for sequences and patterns that will aid in the approximate sorting process.

Chapter 4

Sequence Analysis

In this chapter, we analyze the given input array for sequences that are consecutive and non consecutive. The goal with this knowledge of these patterns is that we will be able to get a better bound on the quality metrics for each of our approximately sorted output arrays. We look at consecutive increasing chains, local minima and non-consecutive chains along with some variations for each of these. There are also new quality metrics that will be introduced along the way.

4.1 Approximate Sorting when the input is a concatenation of Maximal Increasing Chains

Suppose we want to sort $\tau = A[1..n]$ into an increasing sequence. Write $\tau = \sigma_1\sigma_2\ldots\sigma_r$ as a concatenation of sequences where each σ_i is increasing and is maximal in the sense that the last number in σ_i is greater than the first number

of σ_{i+1} for each $i < r$. For every τ, the values of r and σ_is are uniquely defined. Define $cic(\tau)$ (concatenation of increasing chains) to be r. This is referred to as *Runs* in **(Mannila, 1985)** and as *Ascending Runs* in **(Barbay and Navarro, 2013)**. A simple scan of $A[1..n]$ from left-to-right determines r and also $\sigma_1, \ldots, \sigma_r$ using $n-1$ comparisons. r will be 1 if the order is already fully sorted and will be equal to n when the order is reverse fully sorted. Below, we present an algorithm Parsort(A, D), which produces an approximately sorted output using $O(Dn)$ comparisons.

Algorithm 3: Parsort($A[1..n]$,D)

input: An array $A[1..n]$, $n = 2^k$ and $D \leq k$.
Determine $\sigma_1, \ldots, \sigma_r$ satisfying $A[1..n] = \sigma_1 \ldots \sigma_r$. %% $r = cic(A[])$
Compute $M = \min\{2^D, r\}$
Rearrange the sequences (using an extra array if required) so that $|\sigma_i| \geq |\sigma_j|$ for every $1 \leq i \leq M,\ M+1 \leq j \leq r$
Merge the first M sequences $\sigma_1, \ldots, \sigma_M$ to get an increasing subsequence σ of $A[1..n]$ in the first $\sum_{i \leq M} |\sigma_i|$ positions
Output σ.

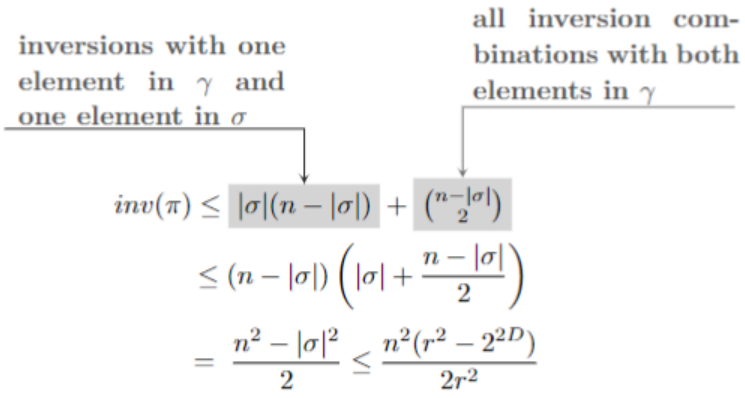
One can prove the following claim about Parsort(). For an ordering $\pi = (x_1, \ldots, x_n)$, let $lp(\pi)$ denote the longest increasing prefix of π. It is not a measure of disorder, but it measures the length of the already sorted and consecutive subsequence present in π. One can use this also as a measure of the quality of sortedness that an algorithm achieves. The above measure of $lp(\pi)$ is a variant of the measure of longest ascending subsequence discussed in **(Mannila, 1985)**, where $lp(\pi)$ is always the prefix of the order.

Theorem 4.1. *Let τ denote the ordering $A[1..n]$. Let r denote $cic(\tau)$. Let π be the ordering of elements in $A[1..n]$ when an execution of Parsort$(A[], D)$ ends. If $2^D \geq r$, π is a fully-sorted (in increasing order) version of $A[1..n]$. Otherwise,*

$lp(\pi) \geq \frac{n2^D}{r}$ and $inv(\pi) \leq \frac{n^2(r^2-2^{2D})}{2r^2}$. In addition, PartSort2() uses $O(Dn)$ comparisons to produce π.

Analysis of the number of comparisons :

Each of the Steps **1** and **2** can be implemented using $O(n)$ comparisons. For Step **3**, we find the M-th largest element among $\{|\sigma_1|, \ldots, |\sigma_r|\}$ using $O(n)$ comparisons. Once this is done, move the subsequences $\{\sigma_i\}_i$ within $A[1..n]$ so that the largest M subsequences occupy the first $\sum_{i \leq M} |\sigma_i|$ positions of A can be achieved using $O(n)$ comparisons (using possibly an extra array of length n).



Merging the first M sorted subsequences into a single sorted subsequence can be accomplished using $O(n(\log M)) = O(Dn)$ comparisons. It follows from the well-known fact that k sorted sequences of total length n can be increasingly sorted in $O(n(\log k))$ time by employing a min-heap structure. Thus, Parsort() performs $O(Dn)$ comparisons on the whole.

Analysis of the quality of the output :

Suppose $2^D \geq r$. Then, $M = r$ and hence all the r pre-sorted subsequences will be merged into a single sorted sequence of length n.

Suppose $2^D < r$. Then, $M = 2^D$. Let $n_i = |\sigma_i|$ for each i. We know $\sum_i n_i = n$. Hence, the M largest subsequences will have a total length $|\sigma| \geq \frac{nM}{r} = \frac{n2^D}{r}$.

Hence, inversion can occur only when either or both elements are part of the subsequences σ_j where $j > M$. Let us refer to the subsequences σ_j where $j > M$ is γ. $\gamma = \sigma_j \sigma_{j+1} \sigma_{j+2} ... \sigma_r$

Therefore, the total number of inversions is given by Equation 1, thereby completing the proof.

4.2 Sorting residual Gamma subarray

The following are consequences of the above theorem. From the above theorem, we can see that the subsequences γ are untouched and we are looking at them as randomly ordered residual numbers. Hence, in Equation 1, the second term counts all possible pair combinations possible for the number of inversions that can occur within γ. We can do better and reduce the value of this second term in Equation 1, by doing a ParSortMedianRearrange() algorithm.

For $i \leq j$, we use $A[i, j]$ to represent the elements in the subarray $A[i...j]$. *ParSortMedianRearrange(A[1..n])* can be designed as a $O(n)$ time algorithm (using **(Blum et al., 1973)**), which can find the median of $A[1..n]$ and rearranges $A[1..n]$ following (A2) given below. The median of $A[1..n]$ is the $\frac{n}{2}$-th smallest element of A when sorted. The following are assumptions about *ParSortMedian-Rearrange()*:

- A1: $ParSortMedianRearrange()$ uses $O(1)$ additional storage. Hence, it is an in-place algorithm.

- A2: One execution of $ParSortMedianRearrange(A[i..j])$ (where $j - i + 1$ is even), once it is done executing, the elements in the array $A[i..j]$ are rearranged such that, all the elements in the second half ($A[\frac{j+i+1}{2}, j]$) are

greater than any element in the first half ($A[i, \frac{j+i-1}{2}]$).

Consider the following algorithm.

Algorithm 4: ParSortMedianRearrange($A[1..n]$,D)

```
input: An array A[1..n], n is a power of 2.
for i ← 0 to D do
    for j ← 1 to 2^i do
        ParSortMedianRearrange(A[(j−1)n/2^i + 1, jn/2^i])
```

Theorem 4.2. *Let π be the ordering of elements in $A[1..n]$ when an execution of ParSortMedianRearrange($A[], D$) ends. Then, $inv(\pi) \leq \frac{n^2}{2^{D+2}}$ and $md(\pi), dis(\pi) \leq \frac{n}{2^{D+1}}$. Moreover, PartSortParSort() uses $O(Dn)$ comparisons to produce π.*

We now use this ParSortMedianRearrange() for the γ subsequence as shown in ParsortwithGammaRearrange(). The algorithm ParsortwithGammaRearrange() first calls ParSort($A[1..n]$) followed by ParSortMedianRearrange(γ, D).

Algorithm 5: ParSortwithGammaRearrange($A[1..n], D$)

```
input: An array A[1..n], n = 2^k and D ≤ k.
ParSort(A[1..n]).
ParSortMedianRearrange(γ, D).
```

The algorithm ParSortMedianRearrange() is called with the subsequence γ from Equation 1, which is of length $n - |\sigma|$. Once we call this algorithm with γ performed in $O(nD)$ time, we will be getting 2^D number of subsequences within γ, each of length $\frac{n-|\sigma|}{2^D}$. We can now rewrite theorem as the follows:

Theorem 4.3. *Let τ denote the ordering $A[1..n]$. Let r denote $cic(\tau)$. Let π be the ordering of elements in $A[1..n]$ when the execution of ParsortwithGammaRearrange(A[],D) ends. If $2^D \geq r$, π is a fully-sorted (in increasing order) version of $A[1..n]$. Otherwise,*

$$lp(\pi) \geq \frac{n2^D}{r};$$

$$inv(\pi) \leq \frac{n^2(r-2^D)(r+2^{4D+2}+2^D)}{2^{D+2}r^2};$$

$$md() \leq \frac{n-|\sigma|}{2^D} + |\sigma|$$

In addition, ParsortwithGammaRearrange() uses $O(Dn)$ comparisons.

The result of this theorem is changed with the second term of Equation 1 becoming more tight now that we have rearranged that with respect to the $ParSortMedianRearrange()$ in *ParsortwithGammaRearrange()*. This brings the number of inversions that occur within γ as $inv(\gamma) \leq \frac{(n-|\sigma|)^2}{2^{D+2}}$. Taking Equation 1 again and substituting this value to it, we get,

$$inv(\pi) \leq |\sigma|(n-|\sigma|) + \frac{(n-|\sigma|)^2}{2^{D+2}} \tag{2}$$

Substituting $|\sigma| \leq \frac{n2^D}{r}$,

$$\leq \frac{(nr-n2^D)}{r} \times \left(\frac{n(2^{4D+2}+r+2^D)}{r2^{D+2}}\right)$$

$$inv(\pi) \leq \frac{n^2(r-2^D)(r+2^{4D+2}+2^D)}{2^{D+2}r^2}$$

The above upper bound is for the number of inversions for π. ■

4.3 Upper bounds based on partition into decreasing subsequences

Instead of partitioning into disjoint increasing subsequences, one can consider partitioning into decreasing subsequences. The analogous Greedy-partitioning, when applied to a permutation τ, will partition into a number $nds(\tau)$ disjoint and decreasing subsequences. One can implement (as before) this greedy algorithm with an analogous PartDecSubseq() procedure which is very similar to PartIncSubseq() and it can be shown that it uses $O(n(\log nds(\tau)))$ comparisons on the whole. If we employ PartDecSubseq in place of PartIncSubseq in ParSort4(), we obtain ParSort4a() and also conclusions similar to Theorem **??** and Corollaries **??** and **??** which employ $nds(\tau)$ in place of $nis(\tau)$.

The reason why we study nds is: there are permutations that may have a high nis value but whose reversals will have a low nis value (for the decreasing ordering). It does not matter if we can sort in the decreasing order because by reversing this sorted order, we obtain (with no extra comparisons) the desired sorted ordering. For example, permutations π_1^R and π_s^R both have high (linear in n) nis values but have nds values 1 and 2 respectively.

4.4 Upper bounds based on partition into monotonic subsequences

Given a permutation $\tau = A[1..n]$ of $[n]$, consider the following algorithm PartMonSubseq() for obtaining a partition of $[n]$ into monotonic (increasing or decreasing) τ-subsequences. This is similar to the previous algorithm PartIncSub-

seq() for partitioning into increasing subsequences but then sometimes focusing on partitioning into monotonic subsequences will help us get partitions into a smaller number of increasing subsequences. The algorithm first partitions $[n]$ into a number of (not necessarily consecutive) monotonic subsequences and then places them in $A[]$ (after reversing decreasing subsequences) so that A becomes the concatenation of increasing subsequences.

4.5 Upper Bounds based on non-consecutive increasing subsequences

The first subsequence pattern that we analyzed was a concatenation of increasing chains where the elements in each subsequence were next to each other in the given array. However, what if we want to form an increasing subsequence where the elements are not necessarily next to each other. Let us take an example to explain this.

$$A = \{10, 20, 15, 5, 25, 17, 30, 8, 35, 13\}$$

The concatenation of increasing chains for the above array would be:

$$\sigma_1 = \{10, 20\}$$
$$\sigma_2 = \{15\}$$
$$\sigma_3 = \{5, 25\}$$
$$\sigma_4 = \{17, 30\}$$
$$\sigma_5 = \{8, 35\}$$
$$\sigma_6 = \{13\}$$

The non-consecutive increasing subsequence for the above array would be:

$$\sigma_1 = \{10, 20, 25, 30, 35\}$$

$$\sigma_2 = \{15, 17\}$$

$$\sigma_3 = \{5, 8, 13\}$$

The non-consecutive increasing sequences can be made of numbers that are not necessarily in adjacent index positions in the given array, A. The main reason for the non-consecutive approach is that we will be reducing the scenarios in which we would want to create a new subsequence, and instead concatenate the new element into an already existing subsequence. The idea would be to concatenate the new element with the subsequence having the greatest element lesser than the new element. If that does not exist, we create a new subsequence with the new element.

The following figure 4.1, visualizes the example from above showing the chains and the subsequences.

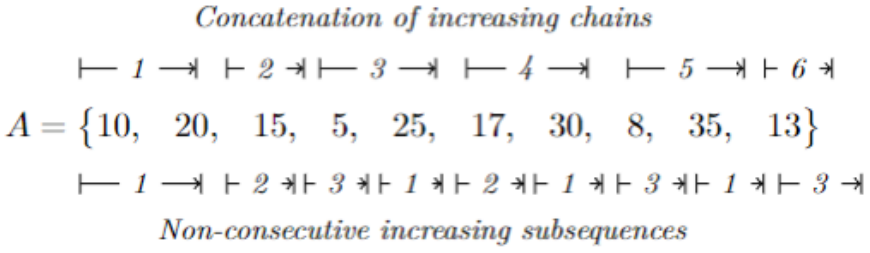


Figure 4.1: Comparison between concatenation of chains and non-consecutive increasing subsequences in an example - represented above and below are indices

The reasoning for finding a subsequence/pattern in the given array is so we have some amount of information prior to sorting so that we can do the sorting more efficiently with that knowledge. Moreover, in the paradigm of approximate sorting, there is a constraint (D) which is the factor of the number of comparisons that can be performed. This means that the sequence analyses must be done within $O(nD)$.

The algorithm explanation follows. We read one element at a time from the given array A. This element $A[1]$ is added to the first subsequence (σ_1). The next element $A[2]$, if greater than the element in σ_1, is concatenated to σ_1. If not, the $A[2]$ creates a new subsequence σ_2. This process keeps going, as we read in a new number ($A[i]$) from our input array, we search for the greatest element less than $A[i]$ from T_G (where, T_G is the BST that maintains the greatest elements from each of the subsequences). If found, we concatenate to the corresponding subsequence σ_x that contains the greatest element less than $A[i]$. If not, we create a new subsequence.

As we are using a BST to store and search the greatest elements from each of the subsequences, it is important for us to keep the time to search and insert into T_G to be less such that the time complexity of the entire algorithm stays below $O(nD)$. This means that the number of elements in T_G should be less than 2^D. Thus, the maximum number of subsequences that we can have is 2^D.

To overcome this issue of being able to store only 2^D elements, we store each subsequence as a BST, maintaining the length of each of them to 2^D, which means we can store at most 2^{2D} number of elements.

This leads us to three cases that we can come across with the number of elements (n) that are given:
Case 1: $n \leq 2^D$; Case 2: $n \leq 2^{2D}$;

Case 3: $n > 2^{2D}$

In the event of cases 1 and 2, all elements from A will be put into the subsequences and result in non-consecutive increasing subsequences. In the case of the third scenario, there will be $n - 2^{2D}$ number of elements that would not be put into subsequences. These residual elements are stored in an array γ as is. We now call the Parsort() algorithm on these subsequences.

Algorithm 6: Parsort3($A[1..n], D$)

input: An array $A[1..n]$, $n = 2^k$ and $D \leq k$.
ParBSTSort($A[1..n], D$).
Parsort($A[1..n], D$).

Theorem 4.4. *Let τ denote the ordering $A[1..n]$. Let π be the ordering of elements in $A[1..n]$ when an execution of ParSort3(A[],D) ends. If $n \leq 2^{2D}$, π is a fully-sorted (in increasing order) version of $A[1..n]$. Otherwise, $inv(\pi) < \frac{n^2-2^{4D}}{2}$. In addition, ParSort3() uses $O(Dn)$ comparisons to produce π.*

Once the algorithm is complete, the subsequences are all concatenated together ($\sigma\gamma$) where, $\sigma = \sigma_1\sigma_2\sigma_3...$ of length 2^{2D} (sorted) and γ holds all the residual values and is of length $n - 2^{2D}$ (not sorted). The inversions can only occur with either element in σ and γ or if both the elements are in γ.

inversions with one element in γ and one element in σ

all inversion combinations with both elements in γ of length $n - |2^{2D}|$

$$inv(\pi) \leq |2^{2D}|(n - |2^{2D}|) + \binom{n-|2^{2D}|}{2} \leq \frac{n^2 - 2^{4D}}{2} \quad (3)$$

Hence proving the number of inversions. We can also take the idea of ParSortMedianRearrange and get to the following algorithm ParBSTsortwithGammaRearrange(). This, as proved before, will tighten the number of inversions

Algorithm 7: ParBSTsort($A[1..n], D$)

input: An array $A[1..n]$, $n = 2^k$ and $D \leq k$.
Generate empty BSTs T_{σ_1}, T_{σ_2}, ... $T_{\sigma_{2^D}}$, to store the subsequences.
Generate empty BST T_G, to maintain the largest element of 2^D number of subsequences (σs). Generate γ empty array to add the elements as is if $n > 2^{2D}$. Insert $A[1]$ into T_G and T_{σ_1}.
for $i \leftarrow 2$ **to** n **do**
    if $i \geq 2^{2D}$ **then**
        break;
    Search T_G for $A[i]$ to obtain the closest element c to it and corresponding T_{σ_x}; $c = max(T_{\sigma_x})$
    if $c > A[i]$ **then**
        Insert $A[i]$ to T_{σ_x}; Balance T_{σ_x}; Update corresponding node at T_G with $max(T_{\sigma_x})$
        If numNodes at $T_{\sigma_x} \geq 2^D$, delete corresponding node at T_G
    else if $c < A[i]$ **then**
        if $\exists$ *an empty* T_{σ_y} **then**
            Insert $A[i]$ to T_{σ_y}; Balance T_{σ_y}; Update corresponding node at T_G with $max(T_{\sigma_y})$
            If numNodes at $T_{\sigma_y} \geq 2^D$, delete corresponding node at T_G
        else
            Insert $A[i]$ to T_{σ_x}; Balance T_{σ_x}; ; Update corresponding node at T_G with $max(T_{\sigma_x})$
            If numNodes at $T_{\sigma_x} \geq 2^D$, delete corresponding node at T_G
$\gamma = A[2^D + 1 : n]$ // copy $n - 2^D$ elements into γ; Output subsequences as inorder traversal of BSTs T_{σ_1}, T_{σ_2}, ... $T_{\sigma_{2^D}}$ and γ

and give us a maximum displacement bound. We will now have the function *ParBSTsortwithGammaRearrange*(), where we will call ParSort3() followed by ParSortMedianRearrange(γ, D).

Theorem 4.5. *Let τ denote the ordering $A[1..n]$. Let r denote $cic(\tau)$. Let π be the ordering of elements in $A[1..n]$ when execution of ParBSTsortwithGammaRearrange(A[],D) ends. If $2^D \geq r$, π is a fully-sorted (in increasing order) version of $A[1..n]$. Otherwise, $lp(\pi) \geq \frac{n2^D}{r}$ and $inv(\pi) \leq \frac{(n-2^{2D})(n+2^{3D+2}-2^{2D})}{2^{D+2}}$ and $md() \leq \frac{n-2^{2D}}{2^D} + 2^{2D}$. In addition, ParBSTsortwithGammaRearrange() uses*

Algorithm 8: ParBSTsortwithGammaRearrange($A[1..n], D$)

input: An array $A[1..n]$, $n = 2^k$ and $D \leq k$.
ParSort3($A[1..n]$).
ParSortMedianRearrange(γ, D).

$O(Dn)$ comparisons to produce π.

To make this bound of the maximum displacement tighter, we can perform a few steps before we do the algorithm above to reduce these upper bounds. It is important that these steps still stay within $O(nD)$. Consider the following algorithm, ParBSTSort2().

Algorithm 9: ParBSTSort2($A[1..n]$,D)

input: An array $A[1..n]$, $n = 2^k$ and $D \leq k$.
for $i \leftarrow 1$ **to** D **do**
  for $j \leftarrow 1$ **to** 2^i **do**
    ParSortMedianRearrange($A[\frac{(j-1)n}{2^i} + 1, \frac{jn}{2^i}]$)
for $j \leftarrow 0$ **to** 2^D **do**
  ParBSTSort($A[\frac{(j-1)n}{2^D} + 1, \frac{jn}{2^D}]$)

$ParBSTSort2()$ algorithm takes at most $c \times numberofelements$ number of comparisons to swap elements such that all elements lesser than the median are in the lower half and all elements greater than the median are in the upper half. c is the constant corresponding to D for the outer loop. The median of medians **(Blum et al., 1973)** is a very popular algorithm that can be used here, having a time complexity of $O(n)$. At each of the D number of steps that we perform, we invoke $ParSortMedianRearrange()$ for each of the partitions and call them recursively. This can be proved similarly to our previous proofs. This means that at the end of D steps, we will have $2D$ number of partitions each of length $\frac{n}{2^D}$. We now call the ParBSTSort() algorithm for each of the 2^D number of partitions.

Each partition is of length $\frac{n}{2^D}$ and we call ParBSTSort() 2^D times, and so each invocation of ParBSTSort() takes $O(\frac{n}{2^D}D)$ time. Therefore, ParBSTSort2() will take $O(nD) + O(\frac{n}{2^D}D2^D)$, which reduces to $O(nD)$.

Theorem 4.6. *Let τ denote the ordering $A[1..n]$. Let π be the ordering of elements in $A[1..n]$ when an execution of ParBSTSort2$(A[], D)$ ends. If $\frac{n}{2^d} < 2^{2D}$, π is a fully-sorted (in increasing order) version of $A[1..n]$. Otherwise, $inv(\pi) \leq \frac{n^2-2^{6D}}{2^{D+1}}$. In addition, ParBSTSort2() uses $O(Dn)$ comparisons to produce π.*

Claim: For each $k \in \{0, 1, \ldots D\}$, the following holds true: the entries of A just after the end of the iteration of the outer loop corresponding to $i = k$ is such that maximum displacement of A can be at most $\frac{n}{2^D}$ (before we call ParBSTSort()), which is the length of the subarray corresponding to $A[\frac{(j-1)n}{2^{k+1}} + 1, \frac{jn}{2^{k+1}}]$ for $j \in \{1, \ldots, 2^{k+1}\}$.

Proof. First, we prove the claim by using an inductive approach to increasing the value of i. For the outermost for loop in $ParBSTSort2()$, the value $i = 0$ will be the base case. This is proven to split the given array into two halves with any element x in the first half lesser than any element y in the second half by definition of the $ParSortMedianRearrange()$ algorithm. The maximum displacement will also $\frac{n}{2}$ as per the definition previously. When we extend the proof to when $i \leq k$, we will be having 2^k number of partitions that would have followed the $ParSortMedianRearrange()$ assumptions with maximum displacement $\frac{n}{2}$, as per the inductive hypothesis. Calling the $ParSortMedianRearrange()$ algorithm for each of these 2^k number of partitions each of length n, splitting each of them into two sub partitions. We will end up with 2^{k+1} number of partitions each following the $ParSortMedianRearrange()$ assumptions each of length $\frac{n}{2^{k+1}}$.

Now, continuing with the second part of the algorithm, we call the ParBSTSort() algorithm for each of the 2^D partitions individually. One call to the ParBSTSort() algorithm on an array of length $\frac{n}{2^D}$ would sort the entire subarray if $\frac{n}{2^D} \leq 2^{2D}$. This is proved using theorem 4.5. However, in the case that if $\frac{n}{2^D} > 2^{2D}$, the ParBSTSort() algorithm will completely sort 2^{2D} number of elements and have $\frac{n}{2^D} - 2^{2D}$ number of residual elements in this partition unsorted.

This will be the case in each of the 2^D numbers of partitions/subarrays in A. Hence, the displacement for each of the elements in each of the partitions would be at most $\frac{n}{2^D} - 2^{2D}$. This means that the maximum displacement would also be $\frac{n}{2^D} - 2^{2D}$. This proves the theorem. □

We can call ParSortMedianRearrange() algorithm for the residual elements as well and tighten the bounds even more, yet having $O(nD)$. Having reduced $A[1..n]$ as non-consecutive increasing subsequences, we apply Parsort() to obtain the Parsort4() algorithm that calls ParBSTSort2() and then ParSort().

Algorithm 10: Parsort4($A[1..n], D$)

input: An array $A[1..n]$, $n = 2^k$ and $D \leq k$.
ParBSTSort2($A[1..n], D$)
Parsort($A[1..n], D$)

4.6 Upper bounds based on number of local optima

For a sequence $\pi = (x_1, \ldots, x_n)$, an element x_i is a local optimum if *either* $x_{i-1} < x_i > x_{i+1}$ *or* $x_{i-1} > x_i < x_{i+1}$. Consider another measure $lopt(\pi)$ which is

the number of local optima. That is,

$$lopt(\pi) = \#\{i : 2 \leq i \leq n-1,\ x_i \text{ is a local optimum}\}.$$

If $x_{i_1}, \ldots, x_{i_s}$ (for $2 \leq i_1 < \ldots < i_s \leq n-1$) form the local optima, then π gets split into at most $s+1$ maximal and monotone (ascending or descending) subsequences determined by the following procedure.

ParSortLocOpt($A[1..n]$**)**

Input : An array $A[1..n]$, $n = 2^k$.

1 $L \leftarrow 1$; $beg[L] \leftarrow 1$;

2 for $i \leftarrow 2$ **to** $n-1$ **do**

2a – **If** $A[i]$ is a local optimum **then**

2b —- $end[L] \leftarrow i$; $L \leftarrow L+1$; $beg[L] \leftarrow i+1$. **endif endfor**

3 $end[L] \leftarrow n$.

4 for $i \leftarrow 1$ **to** L **do**

5 —- **if** $A[beg[i]..end[i]]$ is a decreasing sequence **then** reverse it. **endfor**

end

The following claim can be easily seen to be true. **Claim:** Each of the collection of consecutive subsequences determined by $\{A[beg[i]..end[i]]\}_{i \leq L}$ is a maximal and increasing subsequence whose concatenation forms the sequence $\tau = A[1..n]$ after Step 5. Also, $L = lopt(\tau) + 1$. Also, ParSortLocOpt() performs at most $O(n)$ comparisons.

Having reduced $A[1..n]$ as a concatenation of maximal increasing subsequences, we apply ParSort2() to obtain the following algorithm.

ParSort3($A[1..n], D$**)**

Input : An array $A[1..n]$, $n = 2^k$ and $D \leq k$.

1 ParSortLocOpt($A[1..n]$).

2 ParSort2($A[1..n], D$).

end

One can prove the following claim about ParSort3().

Theorem 4.7. *Let τ denote the ordering $A[1..n]$. Let L denote $lopt(\tau) + 1$. Let π be the ordering of elements in $A[1..n]$ when execution of ParSort3($A[], D$) ends. If $2^D \geq L$, π is a fully-sorted (in increasing order) version of $A[1..n]$. Otherwise, $lp(\pi) \geq \frac{n2^D}{L}$ and $inv(\pi) \leq \frac{n^2(L^2-2^{2D})}{2L^2}$. Also, PartSort3() uses $O(Dn)$ comparisons to produce π.*

This follows from the fact that ParSorLocOpt($A[]$) produces $\tau = A[]$ as a concatenation of at most $L \leq lopt(\tau) + 1$ maximal and increasing subsequences.

As a consequence, we obtain the following corollary. **Corollary:** The following is true :

(a) Any input τ of n elements is sorted by ParSort3(A, D) using $O(n(\log(lopt(\tau)))$ comparisons when $2^D \geq lopt(\tau) + 1$.

(b) Given any $D = D(n)$ such that $1 \leq D(n) \leq \log n$, ParSort3(A, D) produces a sorted ordering of any input τ satisfying $lopt(\tau) + 1 \leq 2^D$ using $O(Dn)$ comparisons.

To conclude this chapter, we analyzed the input array for various types of sequences and patterns. This led to the introduction of new quality metrics (longest increasing prefix). We established bounds for this new quality metric along with the previous ones. There were new approaches that were defined for

when there are residual elements left after approximately sorting which resulted in a better bound for the quality metrics. In the next chapter, we explore searching these approximately sorted orders.

Chapter 5

Search

This chapter explores searching the approximately sorted order given the metric as an input. We start by providing modified binary search approaches with the maximum displacement or distance metrics as input. There are binary search approaches provided that need not have the metric as an input.

5.1 Modified Binary Search with known metrics

The well-known binary search problem reduces the search space by half at each comparison. Given a sorted array $A[1..n] = \tau$ of n distinct elements from a totally ordered set U, searching A for the presence of an element x (and also finding its location in A) can be done using $\lfloor \log_2 n \rfloor + 1$ comparisons. We assume the availability of access to an oracle which given two $x, y \in U$, determines if x $(<, =, >)$ y. It also assumes the availability of access to a membership oracle for U.

5.1.1 With maximum displacement

What happens when A is not sorted but is only almost sorted in the sense that $md(\tau) \leq L$. In this scenario, one can design a search algorithm (which is similar to the binary search procedure) but has to take into account that an element in A may not be in its corrected sorted position since $md(\tau) \neq 0$. When $L = 0$, it reduces to the standard binary search. It is described below and its analysis is given afterwards. We invoke $modBST(A[1..n], L, x)$ to search for the presence of x in A. The following theorem analyzes modBST(). We assume that a single comparison query x $(<, =, >)?$ y returns the correct answer depending on x and y.

Algorithm 11: modBST$(A[i..j], L, x)$

input: An array $\tau = A[i..j]$ and $md(\tau) \leq L$. The presence of x is to be searched.

if $j - i + 1 \leq 4L$ **then**
  search for x in A by a direct search procedure and output either "$x \notin A[i..j]$" or output s such that $A[s] = x$
$r \leftarrow \lceil \frac{j+i-1}{2} \rceil$
if $A[r] = x$ **then**
  OUTPUT r
if $A[r] < x$ **then**
  OUTPUT modBST$(A[r - 2L + 1, j], L, x)$
else
  OUTPUT modBST$(A[i, r + 2L - 1], L, x)$

Theorem 5.1. *$modBST(A[i..j], L, x)$ correctly determines if $x \in A[i..j]$ and outputs its position in A if the answer is YES. For $L \geq 0$, it uses at most $\log_2 \frac{m}{4L} + 4L + 1$ comparisons to correctly determine the answer. Here, $m = j - i + 1$ is the number of elements in $A[i..j]$.*

Correctness :

It is clear from its description that modBST() outputs $x \notin A[i..j]$ when that is the case. Hence, we assume that $x \in A[i..j]$ and prove that it correctly outputs s such that $A[s] = x$. For ease of explanation, we assume, without loss of generality, that $i = 1, j = n$. The arguments for the more general case of $A[i..j]$ are obtained by redefining $i = 1$ and $n = j - i + 1$. Denote $A[1..n]$ by τ. Step 5 correctly determines the position of x in A when $A[r] = x$. We first observe the following claims.

Claim A : Let $\tau_1 = A[1..r + 2L]$. We have $md(\tau_1) \leq L$.

Claim B : Let $\tau_2 = A[r - 2L..n]$. We have $md(\tau_2) \leq L$.

Let $y = A[r]$. We have $r - L \leq r_\tau(y) \leq r + L$. If $A[r] < x$, then $r_\tau(x) \geq r_\tau(y) + 1 \geq r - L + 1$. Hence, we need to search for x among elements with $r_\tau(z) \geq r - L + 1$. Since such elements are are all available in $A[r - 2L + 1..n]$, we have modBST() recursing on $\tau_2 = A[r - 2L + 1..n]$. We also have (by Claim B) that md of this subarray is at most L. This explains steps 8-9.

If $A[r] > x$, then $r_\tau(x) \leq r_\tau(y) - 1 \leq r + L - 1$.

Hence, we need to search for x among elements with $r_\tau(z) \leq r + L - 1$. Such elements are all available in $A[1..r + 2L - 1]$, we have modBST() running on $\tau_1 = A[1..r + 2L - 1]$. We also have (by Claim A) that md of this subarray is at most L. This explains steps 10-11.

Time complexity :

Let $t\{n, L\}$ denote the worst-case number of element comparisons performed by modBST() on inputs of length n and L. From its description, it follows that (after extending the definition of $t()$ to real values of n)

$$t\{n, L\} = 1 + t\Big\{\lfloor \frac{n}{2} \rfloor + 2L, L\Big\} \text{ if } n > 4L$$

$$= n \text{ otherwise.}$$

Extending $t(,)$ to nonnegative real values and expanding the recurrence iteratively, we obtain,

$$t\{n, L\} \leq 1 + t\Big\{\frac{n}{2} + 2L, L\Big\}$$

$$\leq 2 + t\Big\{\frac{n}{4} + L + 2L, L\Big\}$$

$$\leq ....$$

$$\leq k + t\Big\{\frac{n}{2^k} + 4L - \frac{2L}{2^{k-1}}, L\Big\} \tag{5.1}$$

$$\leq \log_2 \frac{n}{4L} + 1 + t\{4L, L\}$$

$$\leq \log_2 \frac{n}{4L} + 1 + 4L \tag{5.2}$$

We substitute $k = \log_2 \frac{n}{4L}$ because of the stopping criteria that we have in place in our algorithm. This completes the proof for the case $i = 1$ and $j = n$. For arbitrary $i \leq j$, the proof is a generalization of the above arguments.

Proof: (of Claim A) For any element y of τ_1, its τ_1-rank is at most its τ-rank, that is, $r_{\tau_1}(y) \leq r_\tau(y)$. All those elements y with τ-rank $r_\tau(y) \leq r + L$ are surely in $A[1..r + 2L]$. We call each of these a *small* element of τ_1 and refer to each of the remaining as a *large* element of τ_1. Each large element y of $A[1..r + 2L]$ has its τ-rank $r_\tau(y)$ lying between $r + L + 1$ and $r + 3L$. Moreover, since exactly L elements of τ with τ-ranks between $r + L + 1$ and $r + 3L$ are not present in τ_1, it follows that $r_{\tau_1}(y) \geq r_\tau(y) - L$ for any such y. Each small element of τ_1 has the

same value for its τ-rank and its τ_1-rank and has same positions with respect to both τ and τ_1 and hence its displacement remains unaffected. For each large y of τ_1, its position (i such that $\tau(y) = \tau_1[i] = y$) remains the same. For a large y if its τ-position i is such that $r+L < r_{\tau_1}(y) \leq r_\tau(y) \leq i \leq r+2L$, then its displacement in τ_1 is at most L. For a large y if its τ-position i is such that $0 \leq r_\tau(y) - i \leq L$, then its displacement in τ_1 satisfies $L \geq r_{\tau_1}(y) - i \geq r_\tau(y) - L - i \geq -L$. Hence, the displacement of every y in τ_1 is at most L and hence $md(\tau_1) \leq L$. ■

Proof: (of Claim B) This case can be proved using analogous arguments as for the case of τ_1. One can work out such detailed proof arguments similar to those worked out in the proof of Claim A. However, a simpler approach would be to reverse the total order and the array. Suppose we reverse the array A as $B[i] = A[n - i + 1]$. r in Step 2 is redefined as $n - \lceil \frac{n}{2} \rceil + 1$. Then B can be considered as an almost sorted (in descending) order with $md(B) \leq L$. The rank of an element in $\mu = B[1..n]$ is given by $r_\mu(y) = n - r_\tau(y) + 1$. Thus, when we go from τ to μ, both rank and position go from a to $n - a + 1$ (a is the value) for any y. Hence, the displacement remains the same for any y in both τ and μ. The case $A[r] < x$ corresponds to $B[r] > x$ and hence we can apply Claim A to μ to infer that displacement remains at most L. When we translate this to A, we get Claim B. ■

The extra additive factor $4L + 1$ cannot be brought down asymptotically (in L) even if the other factor $\log_2 n$ is allowed to be replaced by any function $f(n)$ satisfying $f(n) = o(n)$. The following result shows that the above search algorithm is optimal (up to constant multiplicative factors in the additive term $4L + 1$) with respect to the number of comparison queries made.

Theorem 5.2. *There does* not *exist any deterministic search algorithm which, given a permutation τ of n distinct elements and an element x and also the value of $L = md(\tau)$, correctly determines whether $x \in \tau$ and if so, outputs a s such that $\tau(s) = x$ and makes $o(n) + o(L)$ comparisons. Here, L stands for $md(\tau)$.*

Note: The two asymptotics are with respect to their respective variables n and L.

Suppose there is such an algorithm A. This means that, given arbitrary τ on n elements from an ordered universe U, A correctly determines the presence and position of x in A by making at most $o(n) + o(L)$ comparisons. Since $L \leq n$ always, this implies that one can search for the presence of an arbitrary x in an arbitrary permutation stored in an array of n distinct elements by making $o(n)$ comparisons, which is well-known to be impossible. This contradiction establishes the theorem.

The following example (figure 5.1), shows the first partition step that takes the additional elements from the median index position (r) to the index position $2 \times md$ away. The next iteration will take in the subarray $B[1 : r + 2 \times md]$. In that iteration, we shall find the median index of $B[1 : r + 2 \times md]$ and based on the left or the right partition we go would need to go to, we would either add to the left or the right, $2 * md()$.

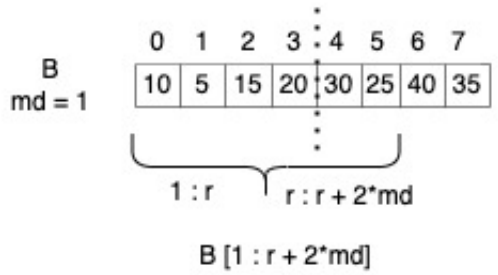


Figure 5.1: Visualization of first partition of modBST()

We also have the following lower bound which shows that the additive term

$\log_2 n$ cannot be brought down asymptotically even if we are allowed to replace $4L + 1$ by any $f(L)$.

Theorem 5.3. *For every $f : \mathcal{N} \to \mathcal{N}$, there does not exist any deterministic algorithm which, given a permutation τ of n distinct elements and an x and also the value of $L = md(\tau)$, correctly determines whether $x \in \tau$ and if so, outputs a s such that $\tau(s) = x$ and makes $o(\log_2 n) + f(L)$ comparisons. Here, $L = md(\tau)$. The conclusion holds even if τ is required to satisfy $md(\tau) \leq L$, for every fixed L.*

Suppose there is such an algorithm A which makes $o(\log_2 n) + f(L)$ comparisons to correctly answer the search query. In particular, this implies that one can answer the search query using $o(\log_2 n) + f(0) = o(\log_2 n)$ comparisons for the fully-sorted permutation τ (corresponding to $L = 0$). This is not possible since it is well-known that any search algorithm (for sorted arrays) is required to make $\lfloor \log_2 n \rfloor$ comparison queries in the worst case. This establishes the claim.

5.1.2 With distance

What happens when A is not sorted but is only almost sorted in the sense that $dis(\tau) \leq L$. Recall that $dis(\tau)$ is the maximum separation distance achieved by any inversion in τ. The search procedure is described below and its analysis is given afterwards. We invoke $modBSTdis(A[1..n], L, x)$ to search for the presence of x in $\tau = A[1..n]$ with $dis(\tau) \leq L$. When $L = 0$, it reduces to the standard binary search.

The following theorem analyzes modBSTdis().

Theorem 5.4. *$modBSTdis(A[i..j], L, x)$ correctly determines if $x \in A[i..j]$ and outputs its position in A if the answer is YES. For $L \geq 0$, it uses at most*

Algorithm 12: modBSTdis($A[i..j], L, x$)

input: An array $\tau = A[i..j]$ and $dis(\tau) \leq L$. The presence of x is to be searched.

if $j - i + 1 \leq 2L + 3$ **then**
  search for x in A by a direct search procedure and output either "$x \notin A[i..j]$" or output s such that $A[s] = x$
$r \leftarrow \lceil \frac{j+i-1}{2} \rceil$
if $A[r] = x$ **then**
  OUTPUT r
if $A[r] < x$ **then**
  OUTPUT modBSTdis($A[r - L, j], L, x$)
else
  OUTPUT modBSTdis($A[i, r + L], L, x$)

$\log_2 \frac{m}{2L} + 2L + 3$ *comparisons to correctly determine the answer. Here,* $m = j - i + 1$ *is the number of elements in* $A[i..j]$.

Correctness : It is clear from its description that modBSTdis() outputs $x \notin A[i..j]$ when that is the case. Hence, we assume that $x \in A[i..j]$ and prove that it correctly outputs s such that $A[s] = x$. For ease of explanation, we assume, without loss of generality, that $i = 1, j = n$. The arguments for the more general case of $A[i..j]$ are obtained by redefining $i = 1$ and $n = j - i + 1$. Denote $A[1..n]$ by τ. Step 5 correctly determines the position of x in A. We first observe the following immediate claims.

Claim C : Let $\tau_1 = A[1..r + L]$. We have $dis(\tau_1) \leq L$.

Claim D : Let $\tau_2 = A[r - L..n]$. We have $dis(\tau_2) \leq L$.

Let $y = A[r]$. If $A[r] < x$, then, we should have $x \in A[r - L..n]$. Otherwise, $dis(\tau) > L$ contradicting our assumption about $dis(\tau) \leq L$. We also have (by Claim C) that $dis(\tau_1) \leq L$. This explains Step 4. Similarly, if $A[r] > x$, then we should have $x \in A[1..r + L]$ by our assumption about τ. In addition, $dis(\tau_2) \leq L$ (by Claim D).

The time complexity can also be proved similarly. Let $t\{n, L\}$ denote the worst-case number of element comparisons performed by modBSTdis() on inputs of length n and L. From its description, it follows that (after extending the definition of t to real values of n)

$$t\{n, L\} = 1 + t\Big\{\lfloor\frac{n}{2}\rfloor + L + 1, L\Big\} \text{ if } n > 2L + 3$$

$$= n \text{ otherwise.}$$

Extending $t\{,\}$ to nonnegative real values and expanding the recurrence iteratively, we obtain that

$$t\{n, L\} \leq 1 + t\Big\{\frac{n}{2} + L + 1, L\Big\}$$

$$\leq 2 + t\Big\{\frac{n}{4} + \frac{3(L+1)}{2}, L\Big\}$$

$$\leq 3 + t\left(\frac{n}{8} + \frac{L}{2} + L + 2L, L\right)$$

$$\leq 4 + t\left(\frac{n}{16} + 4L - \frac{2L}{8}, L\right)$$

$$\leq ...$$

$$\leq k + t\Big\{\frac{n}{2^k} + 2(L+1) - \frac{L+1}{2^{k-1}}, L\Big\}$$

$$\leq \log_2 \frac{n}{2L} + t\{2L + 3, L\}$$

$$t(2L + 2L + L + \frac{L}{2} + \ldots + \frac{8L^2}{n}, L)$$

$$\leq \log_2 \left(\frac{n}{2L}\right) + t(6L, L)$$

$$t\{n, L\} \leq \log_2 \frac{n}{2L} + 2L + 3 \tag{5.3}$$

This completes the proof for the case $i = 1$ and $j = n$. For arbitrary $i \leq j$, the proof is a generalization of the above arguments. ■

We have the following lower bounds on searching permutations with small $dis()$ values, similar to those presented before for the case of small $md()$ values. The proofs are identical and we present only the lower bound results.

Theorem 5.5. *There does* not *exist any deterministic search algorithm which, given a permutation τ of n distinct elements and an element x and also the value of $L = dis(\tau)$, correctly determines whether $x \in \tau$ and if so, outputs a s such that $\tau(s) = x$ and makes $o(n) + o(L)$ comparisons. Here, L stands for $dis(\tau)$.*

Theorem 5.6. *For every $f : \mathcal{N} \to \mathcal{N}$, there does not exist any deterministic algorithm which, given a permutation τ of n distinct elements and an element x and also the value of $L = dis(\tau)$, correctly determines whether $x \in \tau$ and if so, outputs a s such that $\tau(s) = x$ and makes $o(\log_2 n) + f(L)$ comparisons. Here, $L = dis(\tau)$. The conclusion holds even if τ is required to satisfy $dis(\tau) \leq L$, for every fixed L.*

Similar to algorithm *modBST()*, this algorithm *modBSTdis()* can also be replaced with an index range instead of L and the algorithm will perform the same. We can also design algorithms when the metrics are not input to the algorithm. This means that we can start with an estimation of the metric and perform the algorithm with a little overhead which would be a function of the metric itself.

5.2 Modified Binary Search with unknown metrics

Both algorithms modBST() and modBSTdis() assume that the respective values (or an upper bound) of $md(\tau)$ $dis(\tau)$ are known apriori and they are part of the input. This assumption plays a crucial role in establishing the correctness of the algorithm. However, if we only have an estimate L such that $L < md(\tau)$ (or $< dis(\tau)$), the algorithm may give a wrong answer "$x \notin \tau$" when actually $x \in \tau$. For those x such that $x \notin \tau$, it will never give a wrong answer.

This leads us to the following question : How does one search if this knowledge (of md, dis or inv) is missing? If the searched element is present in the array, one can still search and find its position, but at a cost of increased running time. However, if an element is not in the array, then one cannot do better than m comparisons when no apriori knowledge of L is available. Consider the following algorithm for searching with no apriori knowledge of L.

5.2.1 With distance

We present the algorithm only for the case of unknown $dis()$ value. The analogous algorithm for the case of unknown $md()$ value is the same except that one applies modBST() in Step 3 instead of modBSTdis().

Theorem 5.7. *modBSTdisUnKn*$(A[i..j], x)$ *correctly determines if* $x \in \tau = A[i..j]$ *and outputs its position in* A *if the answer is YES. Let* L^* *denotes the smallest nonnegative power of 2 larger or equal to* $dis(\tau)$*. Let* $m = j - i + 1$ *be the number of elements in* $A[i..j]$*. Moreover,*

Algorithm 13: modBSTdisUnKn($A[i..j], x$)

input: An array $\tau = A[i..j]$ and $dis(\tau) = L$. L is unknown. The presence of x is to be searched.

$L \leftarrow 1$; $found \leftarrow FALSE$
while $L \leq (j - i + 1)/4$ **and** $NOT(found)$ **do**
  $ans \leftarrow modBSTdis(A[i..j], L, x)$
  if $ans =$ "$x \notin A[i..j]$" **then**
    $L \leftarrow 2L$
  else
    $found \leftarrow TRUE$
if $NOT(found)$ **then**
  compare x with every member of τ to determine ans
Return ans

- *If $x \in \tau$, then it uses at most $(\log_2 2L^*)(\log_2 \frac{4m}{L}) + 4L^*$ comparisons.*
- *If $x \notin \tau$, then modBSTdisUnKn() will make at least m comparisons. Moreover, no deterministic algorithm can correctly certify this fact with less than m comparisons.*

5.2.2 With Maximum Displacement

Algorithm 14: modBSTmdUnKn($A[i..j], x$)

input: An array $\tau = A[i..j]$ and $dis(\tau) = L$. L is unknown. The presence of x is to be searched.

$L \leftarrow 1$; $found \leftarrow FALSE$
while $L \leq (j - i + 1)/4$ **and** $NOT(found)$ **do**
  $ans \leftarrow modBST(A[i..j], L, x)$
  if $ans =$ "$x \notin A[i..j]$" **then**
    $L \leftarrow 4L$
  else
    $found \leftarrow TRUE$
if $NOT(found)$ **then**
  compare x with every member of τ to determine ans
Return ans

One can design an analogous algorithm modBSTmdUnKn() which is the same

as modBSTdisUnKn() except that in Step 3, we invoke modBST() (instead of modBSTdis()).

Theorem 5.8. *modBSTmdUnKn*$(A[i..j], x)$ *correctly determines if* $x \in \tau = A[i..j]$ *and outputs its position in* A *if the answer is YES. Let* L^* *denotes the smallest nonnegative power of 2 larger or equal to* $md(\tau)$*. Let* $m = j - i + 1$ *be the number of elements in* $A[i..j]$*. Moreover,*

- *If* $x \in \tau$*, then it uses at most* $(\log_2 2L^*)(\log_2 \frac{m}{L}) + 8L^*$ *comparisons.*
- *If* $x \notin \tau$*, then modBSTmdUnKn() will make at least* m *comparisons. Moreover, no deterministic algorithm can correctly certify this fact with less than* m *comparisons.*

We conclude this chapter as we have provided the algorithms for searching for an element and its existence in an approximately sorted array. The binary search algorithms provided have complexity that is comparable to the original binary search algorithms that search a completely sorted order. The binary search algorithms that search without the metric as an input use an approach that estimates the metric for the input approximately sorted array. In the next chapter, we continue to explore searching with range search.

Chapter 6

Range Search

This chapter explores range search on an approximately sorted array. We first establish a 1D range search to find the number of elements within a given range and then also give an algorithm that returns all the elements that are within the given range in an approximately sorted order. This idea is extended to a two-dimensional space and later generalized to a multidimensional space. For the 2D algorithm, we also use a modified fractional cascading approach that is explained in this chapter.

6.1 1D range search

Given a completely sorted array A of n elements and $x, y \in A, x < y$, do a range search to determine all elements in A between x and y. This would involve doing a binary search to find the element x and iterating through all elements greater than x until we find y. The time taken would be $(\log_2 n) + k$, where k is the output length (the number of elements within the given range, including x and

y).

A simpler query would be to find the number of elements that fall within a given range x and y. This would involve searching for each of these two values, $2\log_2 n$ and then subtracting the index positions of the found elements' positions.

6.1.1 With maximum displacement

What if the given array $\tau = A[1..n]$ is not completely sorted but only approximately sorted with an $md(\tau) \leq L$. We can still perform a range search on this approximately sorted order and return a precise list of elements within the given range by modifying the original range search algorithm. The given modified range search algorithm will reduce to the normal range search algorithm when $L = 0$. We give two algorithms.

Find the number of elements within the range

The first one, $modRSNumElements\ (A[i..j], L, [x, y])$ computes the precise number of elements in A which lie in the range $[x, y]$ within an additive error of $4L$. The $modBST$ function call used here is a modified binary search algorithm that is presented in **(Narasimhan et al., 2022c)(Narasimhan et al., 2022b)**. $modBST()$ algorithm performs at most $\log_2 \frac{m}{4L} + 4L + 1$ comparisons. This is a novel algorithm that searches for the existence of a given value x in an approximately sorted array $A[i..j]$ with a corresponding quality metric L.

Lemma 1: $k - 1 - 2L \leq q \leq k - 1 + 2L$.

Proof. First, we prove the lower bound. From Steps 1 and 2, we have $A[lRange] = x$ and $A[rRange] = y$. But since τ is an approximately sorted order with $md(\tau) \leq L$, we know that $lRange - L \leq r_\tau(x) \leq lRange + L$ and $rRange - L \leq r_\tau(y) \leq$

Algorithm 15: modRSNumELements($A[i..j], L, [x, y]$)

Input:$\tau = A[i..j]$, $md(\tau) \leq L$ and a range $[x, y]$ where $x, y \in A[i..j]$ and $x \leq y$.
q – an approximation of the number of elements in A within the range $[x, y]$.
Output: Number of elements in A which lie in the range $[x, y]$ within an additive error of $4L$.

1: $lRange = modBST(A[i..j], L, x)$
2: $rRange = modBST(A[i..j], L, y)$
3: return $(q = rRange - lRange)$

$rRange + L$, where $r_\tau(x)$ represents the $rank_\tau(x)$ and similarly for y. Moreover, the actual number of elements within the range $[x, y]$ is $k = r_\tau(y) - r_\tau(x) + 1$ in the completely sorted order μ of τ. It follows that

$$q = rRange - lRange$$

$$\geq r_\tau(y) - L - (r_\tau(x) + L)$$

$$= r_\tau(y) - r_\tau(x) - 2L$$

$$= k - 1 - 2L$$

For the upper bound, we have

$$q = rRange - lRange$$

$$\leq r_\tau(y) + L - (r_\tau(x) - L)$$

$$= r_\tau(y) - r_\tau(x) + 2L$$

$$= k - 1 + 2L$$

□

Theorem 6.1. *The algorithm modRSNumELements($A[i..j], L, [x, y]$) determines the required number within an additive error of* $4L$ *using at most* $2(\log_2 \frac{m}{4L}+4L+1)$ *comparisons, where* m *is the number of elements in* A*,* $m = j - i + 1, L = md()$*.*

Proof. Lemma 1 establishes the additive approximation within an additive error of $4L$. Each of the two calls to the *modBST()* function in Steps 1 and 2 takes $(\log_2 m+4L+1)$ comparisons as shown by *algorithm 1.* Thus, a total of $2(\log_2 \frac{m}{4L}+ 4L + 1)$ comparisons occur. This concludes our proof. □

An analogous algorithm using *modBSTdis()* **(Narasimhan et al., 2022c,b)** instead of *modBST()* in Step numbers 1 and 2 would result in Theorem 2.

Theorem 6.2. *The algorithm modRSNumELementsdis($A[i..j], L, [x, y]$) determines the required number within an additive error of* $4L$ *using at most* $2(\log_2 \frac{m}{2L}+ 2L+3)$ *comparisons, where* m *is the number of elements in* A*,* $m = j-i+1, L = dis()$*.*

Find the set of elements within the range

The second algorithm $modRangeSearch(A[i..j], L, [x, y])$ is the modified range search algorithm that computes and returns exactly the set of elements of A lying in the range $[x, y]$.

Theorem 6.3. *modRangeSearch($A[i..j], L, [x, y]$) determines the exact list of elements of* $A[i..j]$ *that are within the given range* $[x, y]$ *and performs a total of at most* $\log_2 \frac{m}{4L} + 10L + k$ *comparisons, where* m *is the number of elements in* A*,* $m = j - i + 1$*,* $L = md()$*.*

Algorithm 16: modRangeSearch($A[i..j], L, [x, y]$)

Input:$\tau = A[i..j]$, $md(\tau) \leq L$ and a range $[x, y]$ where $x, y \in A[i..j]$ and $x \leq y$.

Output: Z - set of all elements of A lying within $[x, y]$ is to be determined.

1: $lRange = modBST(A[i..j], L, x)$
2: **for** loop through $z = A[id]$ starting from $id = lRange - 2L + 1$ in A: **do**
3: **if** $x \leq z \leq y$ **then**
Append z to Z
4: **else if** $z < x$ **then**
Ignore z
5: **else if** $z > y$ **then**
6: **for** loop through from current id position to $id + 4L$ in A: **do**
7: Append $z = A[id]$ to Z if $z \in [x, y]$ else ignore z
8: **end for**
9: break;
10: **end if**
11: **end for**
12: Output Z

Proof. Step 1 which is a call to the *modBST()* function, takes $\log_2 \frac{m}{4L} + 4L + 1$ comparisons **(Narasimhan et al., 2022b,c)** to determine the position of the given value x in the order $\tau = A[i..j]$. This yields us $A[lRange] = x$. But since $md(\tau) \leq L$, we have $lRange - L \leq r_\tau(x) \leq lRange + L$. For any $z \in A[i..j]$ such that $z > x$, we have $r_\tau(z) > r_\tau(x) \geq lRange - L$. This implies that all such z's should only lie in $A[lRange - 2L + 1..j]$.

Let $id = i^*$ be the first value for which $z = A[id] > y$. Then, $r_\tau(z) > r_\tau(y)$. Moreover, $i^* - L \leq r_\tau(z) \leq i^* + L$. Combining, we obtain that $r_\tau(y) \leq i^* + L - 1$. This, in turn, implies that any $w \in A[i..j]$ such that $w \leq y$ must be in $A[i..i^* + 2L - 1]$. Thus, it suffices to check all elements in $A[lRange - 2L + 1..i^* + 2L - 1]$ to identify all elements of $A[i..j]$ lying in $[x, y]$.

Denote $lRange - 2L + 1$ and $i^* + 2L - 1$ respectively by X and Y. The number of elements in $A[X..Y]$ is exactly $Y - X + 1 = i^* + 2L - 1 - (lRange - 2L + 1) + 1 =$

$i^* - lRange + 4L - 1$. We claim that $i^* - lRange \leq k + 2L$ where k is the actual number of elements of A in the range $[x, y]$. Otherwise, since i^* is the first index position where we have an element greater than y, this implies that there are more than $2L$ elements smaller than x in $A[lRange..i^*]$. This, in turn, implies that we have an element z such that $r_\tau(z) \leq r_\tau(x) - 2L - 1 \leq lRange - L - 1$ but $z = A[id]$ for some $id \in [lRange..i^*]$. This contradicts our assumption that $md(\tau) \leq L$. Combining all, we deduce that the algorithm performs a total of at most $(\log_2 m) + 10L + k$ comparisons.

□

6.1.2 With distance

This idea of doing a 1D range search on an approximately sorted array can be extended by using the metric as the distance as well. We can substitute step 1 with *modBSTdis()* with some minor changes that will result in an analogous theorem.

Theorem 6.4. *modRangeSearchdis($A[i..j], L, [x, y]$) determines the exact list of elements of $A[i..j]$ that are within the given range $[x, y]$ and performs a total of at most $\log_2 \frac{m}{2L} + 5L + k$ comparisons, where m is the number of elements in A, $m = j - i + 1, L = dis()$.*

6.2 Fractional Cascading on a set of approximately sorted catalogs

Fractional Cascading or Catalog Search is a technique used to search multiple sorted arrays for the same single value. These multiple-sorted arrays are referred

to as catalogs. A simple approach would be to perform a binary search to find the same element in all catalogs. This would take $\log n \times K$, where K is the number of catalogs present. This approach of fractional cascading avoids the multiple binary searches performed. The same idea can be extended to perform multiple-range searches as well. This idea reduces the complexity of multiple range searches that need to be performed on a list of catalogs as well. We shall see how it works below.

Let us take an example of the following two catalogs:

$$A_1 = \{10, 20, 30, 40, 50, 60, 70\}$$

$$A_2 = \{30, 40, 60, 70\}$$

Let us assume that we are given a query to range search all catalogs for the values between $x : [20, 50]$. A naive approach would be to do a binary search followed by traversing the output falling within the range on A_1 and A_2 which would take $\log n_1 + k_1 + \log n_2 + k_2$ time, where k_1 is the numbers to be reported in A_1 of length n_1 and k_2 is the numbers to be reported in A_2 of length n_2, and let $k = k_1 + k_2$.

A better approach would be to build a data structure that has pointers from the elements in the ith catalog to the elements in the $i + 1th$ catalog, such that each element in A_i is pointing to the smallest element greater than or equal to than itself in A_{i+1}.

An example of this representation for the above example is shown in Figure 6.1. Highlighted are the output elements (k of them) that do lie within the given range.

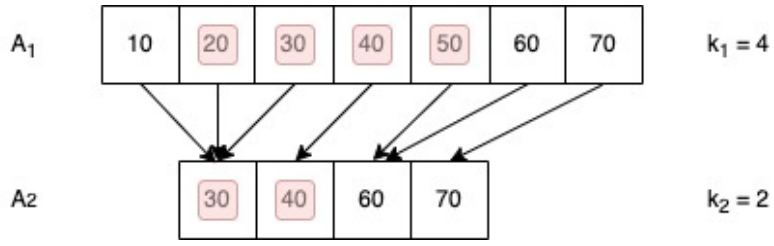


Figure 6.1: Fractional Cascading on completely sorted catalogs

6.2.1 Construction of a Fractional Cascading Structure

The construction of these fractional cascading pointers takes Kn comparisons **(Chazelle and Guibas, 1986a)**. Here, we provide a theorem claiming the construction cost for constructing the fractional cascading data structure for an approximately sorted array.

Theorem 6.5. *Construction of the fractional cascading data structure on an approximately sorted set of K catalogs will take $Kn * 4L$ number of comparisons, where n is the total number of elements and $L = md()$.*

6.2.2 Fractional Cascading using maximum displacement

We consider the quality metric of *maximum displacement (md())* for the analyses and assumptions. The input catalogs are not approximately sorted independently. Assume that each of the catalogs has its own individual *md()* metric that represents the specific maximum displacement.

We recall the *modBST (A[i:j], L, x)* algorithm where A is the input array/-subarray, L is the quality metric and x is the element to be searched for in A. A query to *modBST()* takes $\log n + 4L + 1$ comparisons **(Narasimhan et al., 2022c,b)**.

Let us consider the following catalogs, for example:

$$S_1 = \{30, 20, 10, 50, 40, 80, 110, 60, 100, 90, 70\}$$

$$S_2 = \{30, 20, 70, 110, 90\}$$

$$S_3 = \{60, 110, 30\}$$

where $md(S_1) = 4$, $md(S_2) = 1$ and $md(S_3) = 2$.

In this modified fractional cascading algorithm, we construct the pointer between the approximately sorted catalogs in a slightly different way to compensate for the elements not being in their rank position. The pointer from $S_i[u]$, will be pointing to the $rank(S_{i+1}[v]) - 2L$, where v is the smallest element greater than u and $L = md(S_{i+1})$. This will make sure that when we are moving from $S_i[u]$ to $S_{i+1}[v]$, we do not miss the element v since we do take into account the $md(S_{i+1})$. Figure 6.2 shows the pointers going from S_1 to S_2 and from S_2 to S_3 considering the above example.

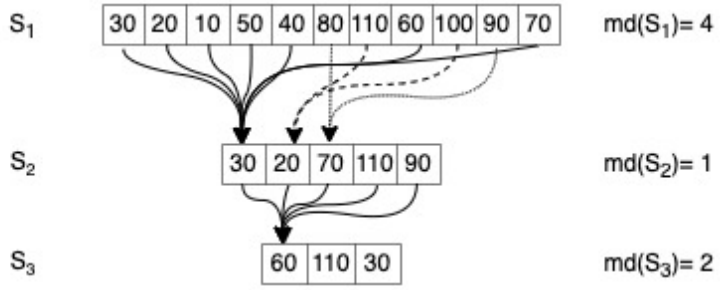


Figure 6.2: Fractional Cascading on an approximately sorted catalogs

In the above example, we see that if the pointer has to point to index positions beyond the range of the catalog, we make it point to the front of the catalog. Another edge case to look into is if the min and max ranges of a catalog S_i are

not present in the next catalog S_{i+1}. In the above example, we have taken it such that all catalogs are a subset in terms of the same min and max range of S_1. Otherwise, we can still make sure that the pointer to the max range is out of bounds by making it point to the last element of the catalog.

Theorem 6.6. *A range search query to find all the elements between a given range* $[x_1, x_2]$ *among* K *number of approximately sorted catalogs/arrays built using fractional cascading, will return the correct output of all the elements that lie between* $[x_1, x_2]$*, in* $\log_2 \frac{n}{4L} + 1 + 4KL + k$*, where* n *is the length of the longest catalog,* L *is the upper bound to the maximum displacement of the catalogs and* k *is the output size.*

Proof. Let us start this proof by considering the time taken for the same if we are going to take a naive approach to do a range query on such approximately sorted catalogs. Our *modBST()* algorithm that would perform a binary search on an approximately sorted array would return the position of the element that we are searching for or the element closest to it. This takes $\log_2 \frac{n}{4L} + 4L + 1$ comparisons **(Narasimhan et al., 2022c,b)**. Therefore, if the output length of the first catalog is k_1, it would take $\log_2 \frac{n_1}{4L_1} + 4L_1 + 1 + (4L_1 + k_1)$. Where, $4L_1$ constitutes to the excess elements the algorithm would need to check if they fall within the range. We would have to repeat this step for all K catalogs. This would become,

$$\sum_{i=1}^{K} \log_2 \frac{n_i}{4L_i} + 1 + 4L_i + (4L_i + k_i) \tag{6.1}$$

$$= (K \times (\log_2 \frac{n}{4L} + 1 + 8L)) + \sum_{i=1}^{K} k_i \tag{6.2}$$

Now considering the fractional cascading approach, we draw pointers for every element (v) from an approximately sorted array $S_i[v]$ to $rank(S_{i+1}[v] - 2L$, where

v is the smallest element greater than u and $L = md(S_{i+1})$, as mentioned earlier.

This would mean that we perform a binary search using modBST() for the first catalog and continue to search the rest of the catalogs by following the pointers and reading the k_i output length for each of them.

$$\log_2 \frac{n_1}{4L_1} + 1 + 4L_1 + k_1 + \sum_{i=2}^{K} 4L_i + k_i \tag{6.3}$$

$$\leq \log_2 \frac{n}{4L} + 1 + 4KL + \sum_{i=1}^{K} k_i \tag{6.4}$$

Comparing the above two equations with approaches with and without using fractional cascading, we can clearly see that the latter approach while we use fractional cascading is going to take lesser time compared to the naive approach, thus proving the theorem.

The equations above can be reduced to the following

$$= \log_2 \frac{n}{4L} + 1 + 4KL + k \tag{6.5}$$

where k is the output size of all catalogs put together. □

6.2.3 Fractional Cascading using distance

An analogous algorithm using the distance metric can also be written.

Theorem 6.7. *A range search query to find all the elements between a given range $[x_1, x_2]$ among K number of approximately sorted catalogs/arrays built using fractional cascading, will return the correct output of all the elements that lie between $[x_1, x_2]$, in $\log_2 \frac{n}{4L} + 1 + 2KL + k$, where n is the length of the longest*

catalog, L is the upper bound to the maximum distance of the catalogs and k is the output size.

6.3 2D range search

A very popular structure that is used for a two-dimensional range search are range trees. A range tree is a balanced binary tree where we essentially keep track of the range of values that are in the leaves of the node's sub-tree. In a 2D range tree structure, there are y-trees attached to each of the intermediate nodes. These y-trees are also range trees containing the y-values of the corresponding x-values in the leaves of the intermediate node's sub-tree.

When performing a 2D query $[(x_1, x_2), (y_1, y_2)]$, we would first have to perform a binary search in the x-tree with x_1. After which we will also have to do the same thing for x_2. Once this is done, there will be intermediate numbers within this range of indices where we would need to search the corresponding y-values to see if they fall within the given $[y_1, y_2]$. This y-value check has to be done at every step of the traversal that leads to multiple binary searches in the corresponding intermediate y-trees. To avoid this, we can store the y-values of each of the intermediate nodes as catalogs **(Chazelle and Guibas, 1986a,b)**. Each intermediate nodes will have a catalog/array of corresponding y-values.

This idea reduces the multiple binary searches that would have to be performed in the y-trees using the fractional cascading approach. We combine this idea with our fractional cascading approach on approximately sorted arrays to perform a 2D range search.

Let us take the following example (figure 6.3) where A is a set of coordinates and we split and write the x and y coordinates separately as A_x and A_y

respectively.

$$A = \{(3,4),(2,7),(1,5),(5,3),(4,10),(7,13),(6,12),(8,11)\}$$

$$A_x = \{3,2,1,5,6,7,6,8\} => md(A_x) = 2$$

$$A_y = \{4,7,5,3,10,13,12,11\} => md(A_y) = 3$$

6.3.1 With Maximum Displacement

Now getting into how we can split the main/first y coordinate catalog, we need to consider the $md(A_y)$. So, the left child elements will be from 1 through $\frac{n_{1y}}{2} + md(A_y)$. Similarly for the right child, elements will be from $\frac{n_{1y}}{2} + md(A_y) + 1$ to n_{1y}.

At every level, there will be duplicate y values stored between siblings/cousins.

Claim 1: Every leaf node u will be of length $< 2md(A_u)$, where A_u is the subarray of A.

Claim 2: $md(A_d) \geq md(A_{d+1})$ at depth d.

From the example figure 6.3, we can see that for every x coordinate value that is approximately sorted, the corresponding y co-ordinate can be found at the respective leaf u subarray in the y range tree, where we can do a linear search of maximum $2md(A_u)$ number of elements. The dotted arrows depict the fractional cascading pointers going from one level of catalogs to a lower level.

We can now find the parent nodes that are completely within the ranges of the index positions. We can find the intermediate nodes that completely have the range within. This will enable us to do a 1D range search on the corresponding y-coordinate values that are attached to that node. The construction of the

fractional cascading data structure for the second dimension will take $2Kn^2 * 4L$ a number of comparisons (from theorem 6.5). This is still lesser than completely sorting all the catalogs. The following theorem analyses the number of comparisons for a 2D range search on an approximately sorted array.

Theorem 6.8. *2D modified range search algorithm determines the exact list of elements of* $A[(a_i, b_i)..(a_j, b_j)]$ *that are within the given range* $[(x_1, x_2), (y_1, y_2)]$ *and performs a total of at most* $5 \log_2 \frac{m}{4L} + 4L(2+3K) + k + 3$ *comparisons, where* k *is the number of elements in* A, $m = j - i + 1$ *and* K *is the number of catalogs to traverse* ($K \leq log_2 m$).

Proof. The x_1 range value is first searched in the given approximately sorted array. We use the modBST() algorithm that we have discussed previously in **(Narasimhan et al., 2022c)**. This will be taking $\log \frac{n}{4L} + 4L + 1$ time to find x_1. Where L is the quality metric (consider it as the maximum displacement).

Once we find this number, we take the index position of this value as i. The values from $i - 2md$ to $i + 2md$ have to be linearly search to check if the x value lies within our x range first and then search for the corresponding y value to check if that lies within the given range. If they both are satisfied, then we are going to be adding value to the output. This will take $4md$ amount of time.

We repeat the same for the x_2 range as well. Let us refer to the position of x_2 as j. So, in total, we have now taken $(\log \frac{n}{4md} + 4md + 1) + (\log \frac{n}{4md} + 4md + 1) + 8md$. We will also have to perform fractional cascading for the branch that we traverse when searching for the x_1 and x_2 ranges. Let us assume that we have identified k' values in the above step.

The rest of the numbers that are certainly within the x range are going to be within the index positions, $i + 2md$ and $j - 2md$. The values between these

two ranges are all going to be within the given x range x_1 and x_2. We now perform fractional cascading for the y values within this range. This will yield the remaining elements in our output, $k - k'$ number of elements. All these together yield $5 \log_2 \frac{m}{4L} + 4L(2 + 3K) + k + 3$ comparisons, where $L = md()$. $\square$

6.4 Multidimensional range search on an Approximately Sorted d dimensional input

The approach taken for the 2D range search can be extended and generalized to a multi-dimensional range search. Let us assume that we are given a d dimensional input data to be searched given a d dimensional query. All the $d - 1$ dimensions are approximately sorted and stored as the range trees. The last dimension is stored as catalogs using fractional cascading.

Multidimensional range search algorithm determines the exact list of elements that are within the given range $[(x_1, x_2), (y_1, y_2), ..(d_1, d_2),]$, where d is the number of dimensions and performs a total of at most $(2 \log_2 \frac{n}{4L} + 8L + 2)^{d-1} + 3(\log_2 \frac{n}{4L} + 4KL + k + 1)$ comparisons, where n is the number of elements of the input data, k is the output size and K is the number of catalogs to traverse ($K \leq log_2 m$).

We conclude this chapter as we have defined range search approaches that give precise output on an approximately sorted order of multiple dimensions. A modified fractional cascading approach was also defined for an approximately sorted set of catalogs that aids in the multidimensional range search reducing the complexity of the search. The following chapter dives into some parallel sorting and searching algorithms that can be performed on an approximately sorted array.

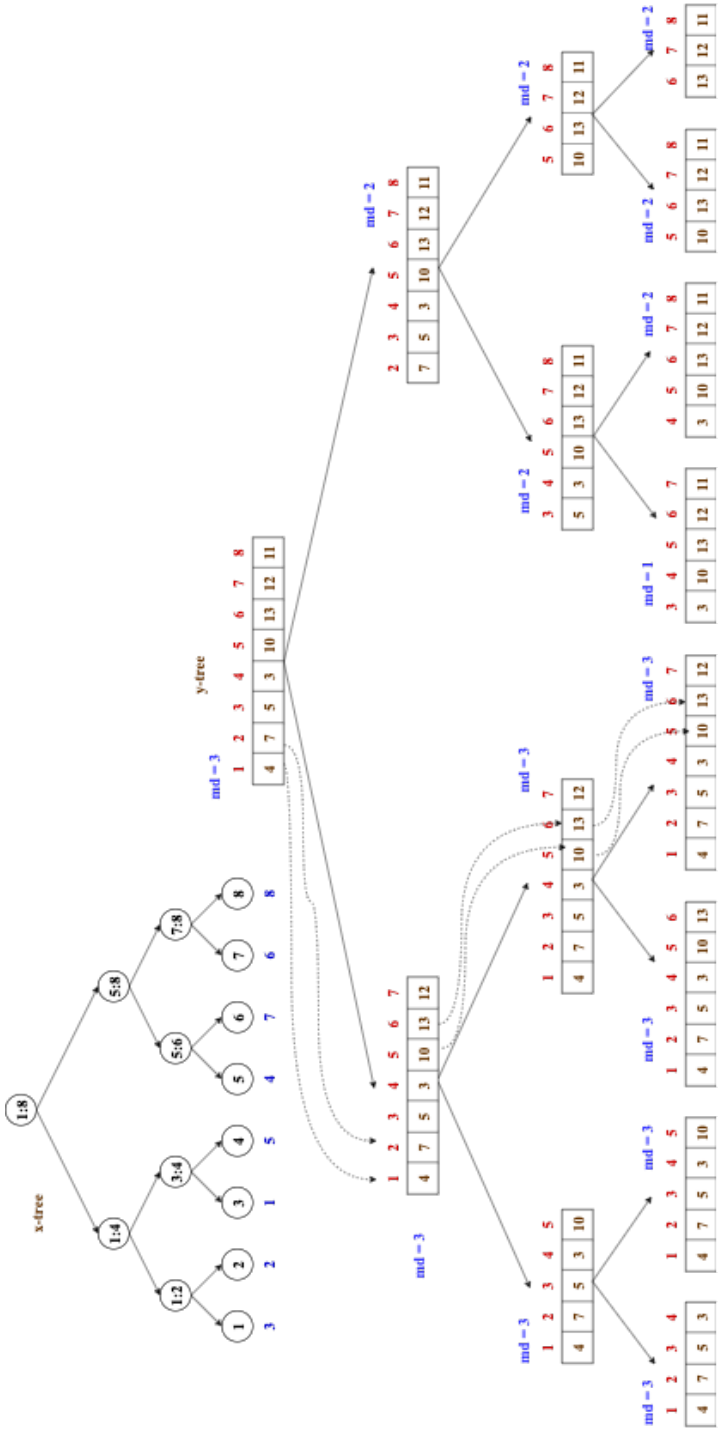


Figure 6.3: Shows the x range tree and y-values stored as catalogs

Chapter 7

Parallel Sorting and Searching

In this chapter, we explore the parallel space where we can apply the approximate sorting idea with the given constraint split between multiple processors. We provide parallel algorithms for this and also provide algorithms to sequentially and parallely search the rank of a given element in a position and vice versa. We also provide multi-select algorithms that work on an approximately sorted array.

7.1 Multi-core Approximate Sorting

The algorithm $ApproxSort()$ being a sequential algorithm results in an approximately sorted output depending on the input factor of the comparisons D. This can be used and extended to a parallel setting, where we use some special techniques to parallelize the approximate sorting.

Algorithm $IncreasingChainSort()$ gives an approach to parallelize the approximate sorting technique, where a matrix A and a constraint D are given as a factor of the number of comparisons that can be made as total work across the

number of processors p. The constraint with the number of comparisons is split between all p processors.

Algorithm 17: IncreasingChainSort($A[1..n]$, D, p)

Input: An array $A[1..n]$, of length $n = 2^q$, $D \leq q$, p processors
Output: Output array τ'
do in parallel:
- τ_{p_i} = ApproxSort($A[i..j]$, $\frac{D}{p}$)
- $L = inv(\tau_{p_i}) \leq \frac{(\frac{n}{p})^2}{2^{\frac{D}{p}+2}}$ upper bound to $inv(\tau_{p_i})$
- In one sweep, find the set of concatenated increasing chains - S_{p_i}
- Each processor will calculate the $minVal_{p_i}$ and $maxVal_{p_i}$ among all increasing chains in $|S_{p_i}|$ time.
- All $minVal$ and $maxVal$ values from all p processors will go through a tournament to fill the *result* output array from both ends - repeat steps 5 and 6.

return $\tau' = result$

Theorem 7.1. *Let τ_{p_i} be the order of the elements of the processor p_i when the execution of ApproxSort() ends. Then $inv(\tau_{p_i}) \leq \frac{(\frac{n}{p})^2}{2^{\frac{D}{p}+2}}$, with each processor using $O(\frac{nD}{p})$ comparisons to produce τ_{p_i}.*

Invocation of *IncreasingChainSort*() starts by splitting the input array $A[1..n]$ into p processors. Each processor p_i gets n/p number of elements. Once the function *ApproxSort*() is called, we refer to this returned subarray as τ_{p_i}. Invocation by a single processor to *ApproxSort*() takes $O(\frac{nD}{p})$ comparisons. The number of inversions $inv(\tau_{p_i})$ will have an upper bound based on the length of the subarray and the factor of comparisons given to the processor p_i.

Theorem 7.2. *Let τ_{p_i} be the order of elements in processor p_i when the execution of ApproxSort() ends. Then the maximum number of increasing chains $|S_{p_i}| \leq 2(inv(\tau_{p_i})) + 1$.*

Once the call to *ApproxSort* returns with τ_{p_i}, we now count the number of increasing chains. We write $S_i = S_i^1 S_i^2 S_i^3 ... S_i^m$ as a concatenation of sequences **(Narasimhan et al., 2022b)** where each S_i^j is increasing and is maximal in the sense that the last number in S_i^j is greater than the first number in S_i^{j+1} for each $j < m$. We define a new metric here $cic(\tau_{p_i})$, concatenation of increasing chains, m **(Mannila, 1985; Barbay and Navarro, 2013; Narasimhan et al., 2022b)**. One scan of the n/p elements in τ_{p_i} will determine $S_i = S_i^1 S_i^2 S_i^3 ... S_i^m$. This uses the $\frac{n}{p} - 1$ number of comparisons. The lower bound for m will be 1 if the subarray in p_i is already sorted. The upper bound will be $m = n/p$ if the subarray is reverse sorted. The number of inversions will dictate the number of increasing subsequences. For every inversion, m will increase by a factor of 2. We also define the measure of the longest increasing prefix $lp(\tau_{p_i})$. Measures the length of the already sorted and consecutive subsequence present in τ_{p_i}. This can also be used as a measure of the quality of sortedness that an algorithm achieves. If $lp(\tau_{p_i}) = n/p$, then the subarray is completely sorted **(Narasimhan et al., 2022b)**. The above measure of $lp(\tau_{p_i})$ is a variant of the measure of the longest ascending subsequence discussed in **(Mannila, 1985)**, where $lp(\tau_{p_i})$ is always the prefix of order.

We introduce an extension of the longest increasing prefix which is defined here. We define the longest increasing affix ($la(\tau)$), which is a combination of the longest increasing prefix and longest increasing suffix. The upper bound for $la(\tau)$ will be $n/2$ when the array is completely sorted.

Theorem 7.3. *For each processor p_i, given an array $B[i..j]$ of size n_i, and a set of increasing chains S_{p_i}, steps 5 and 6 from IncreasingChainSort() will approximately sort B in $O(n_i * D_i)$ where $D \leq |S_{p_i}|/2$. Furthermore, if $D =$*

$|S_{p_i}|/2$, *we will completely sort* B *and* $k = la(B) = n_i/2$). *If* $D = 0$, *then we end up with an unsorted subarray* B *and* $k = la(B) = 0$.

Taking a single processor p_i, we have a $B[i..j]$ subarray of size n_i. In B there are $|S_{p_i}$ number of increasing chains. Each of the chains has a min and a max value. In the $|S_{p_i}$ number of comparisons, we can find the absolute min and max value among all increasing chains. If we continue to do this for $n_i/2$ times, we will end up with a completely sorted version of B. The reason being that we are filling the output array from both ends with the min and max values.

The longest increasing affix (la) will increase by a factor of 2 at each pass to find the absolute min and max values. The constraint D when less than $|S_{p_i}|/2$ means that all the min and max values have not yet been analyzed. If $D = 0$, no comparisons are made and we do not have any of the min and max values of S_{p_i}.

Theorem 7.4. *Let* $\tau'[1..n]$ *be the output array when* $IncreasingChainSort()$ *completes execution. The total work done in Steps 5 and 6 will be* $O(n * D)$ *where* $D \leq (|S_p| * log_2 p)/2$. *If* $D = (|S_p| * log_2 p)/2$, *then* τ' *is completely sorted with* $k = la(\tau') = n/2$. *If* $D = 0$, *then* $k = la(\tau') = 0$. *Furthermore,* $md(\tau'), inv(\tau') \leq n - 2k$.

When every processor gives a min and a max value, it will take $log_2 p$ work in a tournament-style work to determine the min and max in all processors at that iteration. Assuming that the lengths of all subsequences are the same, it will take $(n/2) * |S| * log_2 p$ work to completely sort τ', where $|S|$ is the total number of chains across all processors. But when the constraint is $D \leq (|S| * log_2 p)/2$, then the output array τ' will be approximately sorted. We would be filling the output array from both ends with the min and max values after each tournament outcome, but we will not go all the way, leaving a gap of $n - 2k$, where $k = la(\tau')$.

This means that we would have to copy all the values as such in the middle of the output array ($n - 2k$ window) from each processor. The elements to the left and to the right that were filled with the min and max values are already in their original correct position (rank). Due to this, all inversions or maximum displacement would occur only within the window $n - 2k$.

This is visually represented in the following figure 7.1.

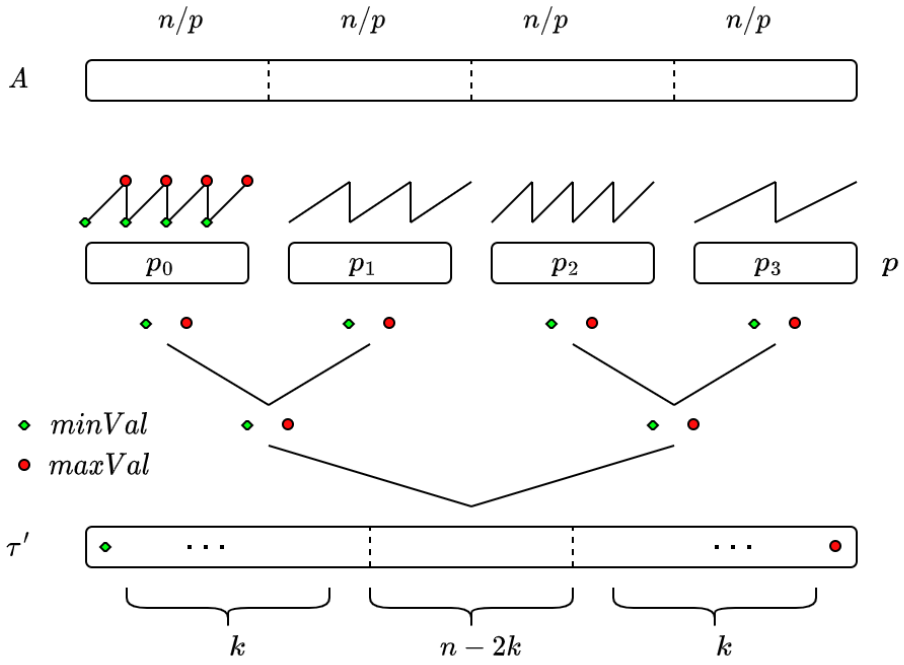


Figure 7.1: Visualization of the sorting process across p processors.

Theorem 7.5. *Let $\tau'[1..n]$ be the output array when IncreasingChainSort() completes execution. The total time will be $O(c * n * D)$, where $c << n$.*

At each phase of the sorting process, we can see that the number of comparisons used in terms of time does not exceed $O(nD)$. The constant factor of c is introduced to show the use of this constraint in multiple phases. But c is a very small value, much smaller than n.

7.2 Determining Rank given Position

Theorem 7.6. *Given an array $\tau = A[1..n]$ of distinct elements and also the value of $L = md(\tau)$ or $L = dis(\tau)$ (or an upper bound) and also a $1 \leq I \leq n$, we want to determine the $rank(A[I])$ in the sorted version of τ. As usual, $rank(x)$ is the number of elements in τ that are at most x.*

When τ is known to be sorted, the answer is I without comparison. When τ is arbitrary, one cannot determine rank with comparisons of less than n comparisons in the worst case. To the best of our knowledge, this problem has not been studied in the literature.

7.2.1 Case of known bound on $md(\tau)$

Consider the following algorithm for the **RfromP** problem with a known bound on $md(\tau)$.

Algorithm 18: RfromPmd($A[1..n], L, I$)

Input: An array $\tau = A[1..n]$ and $md(\tau) \leq L$; $1 \leq I \leq n$.
Output: $rank(A[I])$
$s \leftarrow \max\{1, I - 2L + 1\}$
$S \leftarrow \min\{I + 2L - 1, n\}$;
$rank \leftarrow s - 1$;
for $q \leftarrow s$ ***to*** S **do**
  if $A[q] \leq A[I]$ **then**
    $rank \leftarrow rank + 1$
return $rank$

We have the following claim on RfromPmd().

Theorem 7.7. ***(i)*** *$RfromPmd(A, L, I)$ makes at most $4L - 1$ comparisons to correctly determine the rank of $A[I]$ in $A[1..n]$.* ***(ii)*** *There exists no deterministic algorithm for* ***RfromP*** *which makes at most L comparisons where $L = md(\tau)$.*

(i) Let $x = A[I]$. From the assumption about A, we have $I - L \leq rank(x) \leq I + L$. This, in turn, implies that $A[I] < A[I]$ for each $q \leq I - 2L$ and also that $A[q] > A[I]$ for each $q \geq I + 2L$. Hence $rank(A[I])$ is exactly $s - 1$ plus the number of entries in $A[s..S]$, which is at most $A[I]$. This number is computed by Step 2 and the **for** loop of Steps 3 and 4. The number of comparisons made is at most $4L - 2$.

(ii) It is not hard to see that there is no deterministic algorithm for **RfromP** that always makes fewer comparisons than n. Since $md(\tau) \leq n - 1$ always, this implies the non-existence of an algorithm making at most L comparisons.

Statement **(ii)** establishes that the algorithm RfromPmd() is optimal (with respect to the number of comparisons in the worst-case) up to a multiplicative factor of four. We can extend this to using the $dis()$ metric which will result in the following theorem.

Theorem 7.8. *(i)* $RfromPdis(A, L, I)$ *makes at most* $2L + 1$ *comparisons to correctly determine the rank of* $A[I]$ *in* $A[1..n]$. ***(ii)*** *There exists no deterministic algorithm for* ***RfromP*** *which makes at most* L *comparisons where* $L = dis(\tau)$.

7.2.2 Case of known bound on $dis(\tau)$

Consider the following algorithm for the **RfromP** problem with a known bound on $dis(\tau)$.

We have the following claim on RfromPdis(). Using arguments similar to those employed in the proof of Theorem 7.7, one can establish the claim.

Theorem 7.9. *(i)* $RfromPdis(A, L, p)$ *makes at most* $2L + 1$ *comparisons to correctly determine the rank of* $A[p]$ *in* $A[1..n]$. ***(ii)*** *There exists no deterministic algorithm for* ***RfromP*** *which makes at most* L *comparisons where* $L = dis(\tau)$.

Algorithm 19: RfromPdis($A[1..n], L, p$)

Input: An array $\tau = A[1..n]$ and $dis(\tau) \leq L$; $1 \leq p \leq n$.
Output: $rank(A[p])$
$s \leftarrow \max\{1, p - L\}$; $S \leftarrow \min\{p + L, n\}$;
$rank \leftarrow s - 1$;
for $q \leftarrow s$ ***to*** S **do**
  if $A[q] \leq A[p]$ **then**
    $rank \leftarrow rank + 1$
return $rank$

The statement **(ii)** establishes that Algorithm RfromPdis() is optimal (with respect to several comparisons in the worst-case) up to a multiplicative factor of two.

When no knowledge of the exact (or a bound) value of the two parameters is available, one can still try to determine the rank of an element in a given position by using the increasingly doubled estimates of the values of these parameters. While this will determine $rank(A[p])$ when the estimate on $L = md(\tau)$ becomes equal to L^* defined in **(Narasimhan et al., 2022c)**, there is no known rank verification algorithm which makes $O(L)$ comparisons. This would be interesting to design an algorithm for determining (while making $O(L)$, $L = md(\tau)$ comparisons) the rank with no knowledge of $md()$. The situation is similar with respect to $dis(\tau)$ knowledge.

7.3 Parallel Rank given Position

The natural next step to improve the performance of the $RfromPmd$ algorithm is to parallelize it.

Theorem 7.10. *Given an array $\tau = A[1..n]$ of distinct elements and also the value of $L = md(\tau)$ or $L = dis(\tau)$ (or an upper bound) and also a $1 \leq I \leq n$*

with p processors, we want to determine rank given the position I (and also the element A[I]) in τ. Note that we do not allow write access for the array, but only read access.

The following algorithm shows the parallel approach that can be taken to find the *rank*.

Algorithm 20: RfromPmdParallel($A[1..n], L, I, p$)

Input: An array $\tau = A[1..n]$ and $md(\tau) \leq L$; $1 \leq I \leq n$.
Output: $rank(A[I])$
$s \leftarrow \max\{1, I - 2L + 1\}$
$S \leftarrow \min\{I + 2L - 1, n\}$;
$n_P = S - s + 1$
$rank \leftarrow s - 1$;
$rank_{P_i} = 0$
do in parallel: for each chunk of size n_P/p
  $q = i \times n_P/p$
  if $A[q] \leq A[I]$ **then**
    $rank_{P_i}++$
  q++
$rank+ =$ sum of $rank_{P_i}$
return $rank$

Theorem 7.11. *The RfromPmdParallel algorithm finds the rank of the given element A[I] in position I in τ = A[1..n]. Also given are L = md(τ) and 1 ≤ I ≤ n with p processors, in O(n_P) number of comparisons in time and work performed O(L).*

The algorithm first splits the window of $A[s..S]$ where we know that *rank* of $A[I]$ exists into equal chunks of n_P among p processors. We also initialize the *rank* to the index position pointing to the left more index of the window. Each processor has an individual variable $rank_{P_i}$ that keeps track of the number of elements that are less than $A[I]$. As these processors calculate $rank_{P_i}$, in

parallel, we count the total number of elements that are less than $A[I]$ in each chunk. Once all processors are completed counting, we add them and sum them with the original *rank*. This will give us *rank* of $A[I]$ in τ. This algorithm is also naturally load balanced as every processor performs the same number of comparisons.

7.4 Determining position given rank

Consider the following dual computational problem of determining the position of an element with a given rank.

Theorem 7.12. *Given an array $\tau = A[1..n]$ of distinct elements and also the value of $L = md(\tau)$ or $L = dis(\tau)$ (or an upper bound) and also a $1 \leq r \leq n$, we want to determine the position s (and also the element) in τ of the unique element whose τ-rank is r. Note that we do not allow write access for the array, but only read access.*

This problem is the same as the famous search problem of finding the **k-smallest element** in an array except that we ask for extra information on L and make use of it to determine the answer using fewer comparisons. The very popular algorithm for finding the **k-smallest element** is quick select **(?)**.

When τ is known to be sorted, the answer is $(r, A[r])$ without comparison. When τ is arbitrary, one cannot determine the position with fewer than n comparisons in the worst case. For arbitrary τ (without knowledge of L), there are known algorithms (like the one based on finding the median of the medians) that determine the position and the element by making $O(n)$ comparisons. For those τ whose values $md()$ or $dis()$ are small, we present an algorithm that makes far

fewer comparisons. To the best of our knowledge, this problem has not been studied in the literature.

7.4.1 Case of known bound on $md(\tau)$

Consider the following algorithm for the **PfromR** problem with a known bound on $md(\tau)$. Below, $Find(B[i..j], r)$ is an algorithm to find, given an array $\mu = B[i..j]$ and a $r \geq 1$, the position (and also the element) whose μ-rank is r. Again, we do not allow write access to the array, but only read access on B. It is known that the best-known candidates (such as the median of medians algorithm) **(Blum et al., 1973)**) for $Find()$ make at most $O(j - i + 1)$ comparisons.

Algorithm 21: PfromRmd$(A[1..n], L, r)$

Input: An array $\tau = A[1..n]$ and $md(\tau) \leq L$; $1 \leq r \leq n$.
Output: Position I of element x
$s \leftarrow \max\{1, r - L + 1\}$
$S \leftarrow \min\{r + L - 1, n\}$;
$(I, x) \leftarrow Find(A[s..S], L)$;
return (I, x).

We have the following claim for PfromRmd().

Theorem 7.13. *(i)* $PfromRmd(A, L, r)$ *makes at most* $O(L)$ *comparisons to correctly determine the element whose* τ*-rank is* r *and also its position in* A. ***(ii)*** *There exists no deterministic algorithm for* ***PfromR*** *which makes at most* L *comparisons where* $L = md(\tau)$.

(i) Let $x = A[I]$ be the unique element whose τ-rank is r. From the assumption about A, we have $A[q] < x$ for every $q < s$. Similarly, $A[I] > x$ for every $q > S$. Hence $x \in A[s..S]$. In addition, x is the L-th smallest element in the subarray $A[s..S]$. Its value and also its position are found in Step 2 by an invocation of

Find($A[s..S], L$) and returned in Step 3. The number of comparisons is essentially determined by the number of comparisons made by Find() which is $O(L)$.

(ii) It is not hard to see that there is no deterministic algorithm for **PfromR** that always makes fewer comparisons than n. Since $md(\tau) \leq n-1$ always, this implies the non-existence of an algorithm making at most L comparisons.

The statement **(ii)** establishes that Algorithm PfromRmd() is optimal (with respect to the number of comparisons in the worst-case) up to a constant multiplicative factor. We can extend this to using the $dis()$ metric which will result in the following theorem.

7.4.2 Case of known bound on $dis(\tau)$

Consider the following algorithm for the **PfromR** problem with a known bound on $dis(\tau)$.

Algorithm 22: PfromRdis($A[1..n], L, r$)

Input: An array $\tau = A[1..n]$ and $dis(\tau) \leq L$; $1 \leq r \leq n$.
Output: Position p of element x
$s \leftarrow \max\{1, r-2L\}$; $S \leftarrow \min\{r+2L, n\}$;
$(p, x) \leftarrow Find(A[s..S], 2L)$;
return (p, x).

We have the following claim on PfromRdis(). Using arguments similar to those employed in the proof of Theorem 7.7, one can establish the claim.

Theorem 7.14. **(i)** *$PfromRdis(A, L, p)$ makes at most $O(L)$ comparisons to correctly determine the rank of $A[p]$ in $A[1..n]$.* **(ii)** *There exists no deterministic algorithm for* **PfromR** *which makes at most L comparisons where $L = dis(\tau)$.*

(i) Let $x = A[p]$ be the unique element of τ-rank r. We claim that $p \geq r - L$. Otherwise, some element whose τ-rank is at most $r-1$ should occupy a position

$s \geq r$. This gives rise to an inversion $(A[p], A[s])$ with a separation at least $s - p \geq r - (r - L - 1) = L + 1$ violating our assumption about $dis(\tau) \leq L$. Similarly, $p \leq r + L$. As a consequence, it follows that $A[q] < x$ for every $q < r - 2L$ and also that $A[q] > x$ for every $q > r + 2L$. Hence, x is precisely the $(2L + 1)$-th smallest element in the subarray $A[s..S]$ and its value and position are found by the invocation of Find() in Step 2. The number of comparisons made is $O(L)$.

(ii) Follows from arguments employed in Statement **(ii)** of Theorem 7.13 and from $L = dis(\tau)$.

Statement **(ii)** establishes that Algorithm PfromRdis() is optimal (with respect to several comparisons in the worst-case) up to a constant multiplicative factor.

7.5 Parallel Position given Rank

This approach to finding the position of elements that have a rank r in an approximately sorted array can be extended to be implemented in a parallel setting to improve the performance of the query.

Theorem 7.15. *Given an array $\tau = A[1..n]$ of distinct elements and also the value of $L = md(\tau)$ or $L = dis(\tau)$ (or an upper bound) and also a $1 \leq r \leq n$ with p processors, we want to determine the position s (and also the element) in τ of the unique element whose τ-rank is r. Note that we do not allow write access for the array, but only read access.*

The following algorithm shows the parallel approach that can be taken to find the position s along with the element in τ with rank r.

Algorithm 23: PfromRmdParallel($A[1..n], L, r, p$)

Input: An array $\tau = A[1..n]$ and $md(\tau) \leq L$; $1 \leq r \leq n$; p number of processors.

Output: Position I of element x

do in parallel:

- $s_i \leftarrow \max\{1, r - L + 1\}$
- $S_i \leftarrow \min\{r + L - 1, n\}$
- $(I, x) \leftarrow FindParallel(A[s_i..S_i], L)$

return (I, x).

Theorem 7.16. *The $PfromRmdParallel$ algorithm finds the position I (and also the element whose τ-rank is r) in $\tau = A[1..n]$ given $L = md(\tau)$ and also a $1 \leq r \leq n$ with p processors, in $O(L/p + p)$ number of comparisons in time and work performed $O(L)$.*

The $FindParallel$ algorithm used here takes the subarray $B[i..j]$ of length $t = j - 1 + 1$, and divides it into a t/p number of elements per processor P_i. Each processor will perform the best known median-of-medians algorithm **(Blum et al., 1973)**) and send it to the master where the algorithm is performed again to find the position p and the element s. As is known, the median-of-medians algorithm finds the median of the subarray in a single processor P_i in $O(t/p)$ comparisons. And the next phase of this algorithm will take place $O(p)$ to find the median among all the medians of each processor P_i. This results in a total of $O(t/p + p)$ number of comparisons where $t = 2L$. One may also use more sophisticated approaches to find the median that involves multilevel parallelism **(??)**.

7.6 Multi Select in an approximate sorted array

Assuming that these values in the given array K all have a window independent of each other (without overlaps) in the given approximately sorted array A, we can come up with a simple approach to parallelize this. We present the following algorithm.

Algorithm 24: $k_i$thPfromRmdParallel$(A[1..n], L, K[1..q], p)$

Input: An array $\tau = A[1..n]$, $md(\tau) \leq L$, array $K = [1..q]$ with indices and $1 \leq q \leq n$.
Output: x_i and its position I_i in τ.
do in parallel: for each element in K
- $s \leftarrow \max\{1, k_i - L + 1\}$
- $S \leftarrow \min\{k_i + L - 1, n\}$
- $(I_i, x_i) \leftarrow PfromRmd(A[s..S], L, r)$
- output (I_i, x_i)

Theorem 7.17. *$k_i thPfromRmdParallel()$ does at most $O(qL)$ work and $O(L)$ comparisons in time to correctly determine the element whose τ-ranks are k_i from $K[1..q]$ and also their corresponding positions in A, where $1 \leq i \leq q \leq n$.*

Each rank K_i is assigned to a processor of its own to perform the $PfromRmd$ algorithm. Every processor will receive a subarray of size $2L$. As the total work is divided between p processors, the time taken becomes $O(L)$ to find each element x_i in rank k_i.

This approach can be extended to multiple levels of parallelism. We could use the $PfromRmdParallel$ algorithm in line 4 which would reduce the time even further.

7.7 Parallel Multi Select in an approximate sorted array

Assuming that these values in the given array K all have a window independent of each other (without overlaps) in the given approximately sorted array A, we can come up with a simple approach to parallelize this. We present the following algorithm.

Algorithm 25: $k_i$thPfromRmdParallel$(A[1..n], L, K[1..q], p)$

Input: An array $\tau = A[1..n]$, $md(\tau) \leq L$, array $K = [1..q]$ with indices and $1 \leq q \leq n$.
Output: x_i and its position I_i in τ.
do in parallel: for each element in K
  $s \leftarrow \max\{1, k_i - L + 1\}$
  $S \leftarrow \min\{k_i + L - 1, n\}$
  $(I_i, x_i) \leftarrow PfromRmd(A[s..S], L, r)$
  output (I_i, x_i)

Theorem 7.18. *$k_i thPfromRmdParallel()$ does at most $O(qL)$ work and $O(L)$ comparisons in time to correctly determine the element whose τ-ranks are k_i from $K[1..q]$ and also their corresponding positions in A, where $1 \leq i \leq q \leq n$.*

Each rank K_i is assigned to a processor of its own to perform the $PfromRmd$ algorithm. Every processor will receive a subarray of size $2L$. As the total work is divided between p processors, the time taken becomes $O(L)$ to find each element x_i in rank k_i.

This approach can be extended to multiple levels of parallelism. We could use the $PfromRmdParallel$ algorithm in line 4 which would reduce the time even further.

We conclude this chapter with the observation and bounds for the various metrics that we defined earlier in the parallel space when the constraint of the number of comparisons is split between various processors. In the next chapter, we dive into one of the applications of approximate sorting, packet classification in software defined networks.

Chapter 8

Network Packet Classification with Approximate Sorting

In this chapter, we explore one of the applications of approximate sorting, packet classification in software defined networks. We define what comparing a packet means and provide algorithms to approximately sort a set of packets in a buffer, that leads to gains in the speed of classification. We also provide analysis by comparing fully sorted and unsorter approaches.

8.1 Introduction

Packet buffers are part-and-parcel of any packet switching network. With increases in the availability of 100 Gbit networks or more and with a plethora of available applications, the traffic volume on the Internet has exceeded 130K petabytes in some estimates **(Clement, Feb 28, 2020)**. A buffer is allocated for each of the router interfaces. The router CPU processes each of these queues

simultaneously or sequentially, depending on the number of processors available. This processing involves determining which outgoing queue a packet from the input queue must be placed in or filtering to remove packets altogether. This may involve some packet classification.

Routers employ a set of rules that each packet is subject to **(Fuchino et al., 2021a; Daly et al., 2019)**. For example, a simple rule would be to search the routing table based on the destination address of each incoming packet. Of course, rules are usually more complicated, such as those deployed for discarding to prevent the spreading of worms. Many modern routers allow for new network paradigms, such as software-defined networking (SDN) through OpenFlow **(Hakiri et al., 2014)** and virtualization of network functions (NFV) **(Han et al., 2015)**. These paradigms are highly dependent on packet classification, which is performed by executing each of the rules (*classifier*) on the information contained in the header of each packet and sometimes in the payload as well **(Daly et al., 2019; Gupta and McKeown, 2001)**.

Packet classification algorithms often preprocess the set of rules by ordering rules **(Hamed and Al-Shaer, 2006; Fuchino et al., 2021b)**, decision tree approaches **(Li et al., 2018b)**, or partition methods such as **(Daly et al., 2019; Yingchareonthawornchai et al., 2018)**. Each packet, one by one, is processed through all rules after the rules have been preprocessed. Processing packets one by one; no matter how the rules are organized, this will be a slow process. If the packets in the buffer are 'similar', we can execute the rule on the first, obtain the result and use it for all the others. Say that we have k similar groups of packets in a buffer with n packets, then we need to execute the rules only on k packets.

Taking into account constant traffic, the number of packets n in a single input

interface buffer depends on the input line rate. Based on our earlier observations, one can assume that there will be many packets in each of the interface buffers. These observations on buffer sizes have previously been thoroughly examined **(Abbas et al., 2016; Prasad et al., 2009)**.

8.1.1 Decision Tree Based Packet Classification

Let R be a set of rules, commonly known as a set of classes. Let $A = \{a_1, a_2, ..., a_k\}$ be the set of attributes of the packet headers referred to in the rules. A packet could satisfy multiple rules, in which case it is not uniquely classified. Rather than assigning multiple classes, we can uniquely assign one class (the first rule that satisfies).

Often a decision tree-based packet classifier is used, where the root node covers the entire search space that contains all R rules. The space corresponding to the node (root, for example) is partitioned into two smaller subspaces, assuming that we are constructing a decision tree that is a binary tree. This is often the case, and therefore the height of the decision tree is $O(\log R)$. Recursive partitioning of the rule space continues until we have a leaf node that contains a single rule. In cases where the subspace split overlaps because of a rule(s), the rule(s) are replicated (certainly undesirable). A packet traverses this decision tree until it reaches a leaf node where it satisfies the single rule that it is the leaf node.

Numerous approaches have been used to create decision tree-based packet classification, including well-studied CutSplits **(Li et al., 2018a)** and HiCuts **(Gupta and McKeown, 2000)** and its improved version, HyperCuts **(Singh et al., 2003)**. PartitionSort **(Yingchareonthawornchai et al., 2018)** is an approach that sorts the ruleset, which we assume and use in our paper. Parti-

tionSort defines the comparison of packets as follows: Consider $\leq$ which defines the total ordering of a set of n given rules $r_1...r_n$. To classify a packet P_x, we compare P_x with rule r_i and determine $P_x > r_i$. This means that the packet P_x may match some rules among $r_{i+1}...r_n$ and will not match rules from r_1 through r_i. In our paper, we take a similar approach to the comparison of two packets, elaborated more in Section 8.1.2. This sorted list of rules becomes the leaf nodes on which a decision tree is built. We have illustrated the decision tree and the path the packet takes in Fig. 8.1.

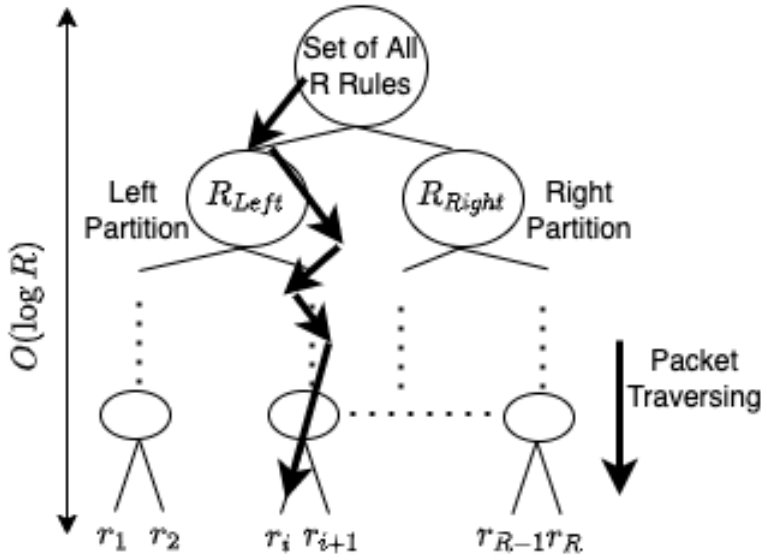


Figure 8.1: The decision tree and R rules as leaf nodes.

Each packet that needs to be classified has to spend $O(\log R)$ time traversing the binary decision tree. Sorting can help reduce the traversal time by looking at the classification of the previous packet. Once this node is determined, we move from that node to a leaf node that applies to the packet. Given that sorting is a time-consuming step, we have developed algorithms for approximate sorting and then provide techniques for packet classification based on the metrics on

approximate sorting.

- This section proposes an approximate sorting technique that requires $O(\delta \times D \times n)$ time to complete the sorting, where D is the number of allowed comparisons and $\delta > 0$ is a constant, and is the cost of comparing a packet pair. Keeping $D \ll \log n$ will reduce the overall time to preprocess the packet, but at the expense of not being sorted. To measure *sortedness* of the output as a result of the algorithm, we discuss some metrics. We show that as D increases, the metrics we measure decrease and move closer to sorted. The discussion of approximate sorting is provided in Section II.

- Once the set of packets is approximately sorted, we show how to classify the packets without having to subject the packet to the entire decision tree from the root. We show the overall effectiveness of our approximate sorting methods for packet classification and examine them by varying the values of D, n, and δ. We can determine the optimal values for which the approximate sorting will be helpful through these evaluations. These are discussed in Section III.

- We relate the bursty throughput time to one of the metrics that we define later (maximum displacement - md). Our analysis provides us with a clear proportionality of md with the duration of the bursty traffic. We explain this correlation in Section 8.3.

8.1.2 Comparing Two Packets

Let P_i and P_j be two packets containing fields $P_i = (a_{i1}, a_{i2}, ...a_{im})$ and $P_j = (a_{j1}, a_{j2}, ...a_{jm})$. When we compare lexicographically packets P_i and P_j, we can

say that packets $P_i = P_j$, if for all r, $1 \leq r \leq m$, $a_{ir} = a_{jr}$. We can say $P_i > P_j$, if for some r, $1 \leq r \leq m$, $a_{ir} > a_{jr}$ and for all s, $1 < s < r$, $a_{is} = a_{js}$. The operator less than $<$ ($P_i < P_j$) can be defined similarly.

One can define more complicated relational operators with an eye toward the rules for classification. Let us say a compound classification rule that involves more than one packet attribute, for example, destination address and port number. More precisely, we say that a packet belongs to class C_1 if its destination address is in the set S of the destination address and its destination port is the given range of port numbers. The packets can be presorted only by destination address. Here we will say that two packets are the same if both the packet's destination addresses are in the set S. The relational operators $>$ and $<$ can be defined by comparing the destination address of the packets with the smallest and the largest addresses in the set S. In general, we can presort (or approximate sort) based on the partial set of attributes and with the corresponding logic to determine the results of the relational operators **(Narasimhan et al., 2022a)**.

8.2 Relationship between quality metrics and Traversing the Decision-Tree

The packets that arrive at a router belong to many flows, whereas a flow is a sequence of packets that belong to a source and a destination pair. All packets that belong to a particular flow do not need to be consecutive in the input buffer. Packets belonging to a particular flow may also not be classified similarly.

When we completely sort the packets in the input buffer, adjacent packets in the sorted list could belong to the same class or a class that is closest to it. Here

is the reason. Consider the decision tree in which the leaves are the classification of the packets and the path from the root to the leaf places the conditions on the attributes $(a_1, a_2, ...a_m)$ in order $1, 2, ...m$. We need to sort each packet P_i based on attributes $(a_{i1}, a_{i2}, ...a_{im})$ (again in that order). Two packets P_i and P_j are closer in the sorted list if $P_i = (a_{i1} = a_{j1}, a_{i2} = a_{j2}, ...a_{ik} = a_{jk}$ or P_i and P_j agree on most of the attributes starting from a_1 (in the order) until a_k. The packets P_i and P_j will be next to each other in the completely sorted array if there exists no other packet P_s that agrees on more attributes $(a_1, a_2, ...a_m)$ of P_i in comparison with the same attributes of P_j. The basic idea of our approach is to determine the packet classification for some of the packets in the input buffer and to use the sortedness of the set to reduce the search space. This is shown below for all three metrics.

As defined above, $dis(\pi)$ is the maximum distance (in array locations) between the locations of the packet pairs that are out of order. Given a list of n packets, the maximum distance is $n - 1$. In this case, an approach to packet classification would be to find the classification for the first packet. Let us say that this packet P_i belongs to class r, that is, it is the rth leaf node in the decision tree. To find the class for the second packet P_{i+1}, we climb the tree from the leaf node r, until we find the node k that matches the rules for that node. Once it matches, we traverse the tree from node k. The process continues for all other packets, where we start climbing the tree for the next packet based on the leaf position (packet classification) of the previous packet. The maximum climb to the tree is given by $dis(\pi)$ and the sum value of all the distances between inversions can be shown to be $n \times (n - 1)/2$. This will be valid in the case where the number of packets in the buffer is less than the number of rules for packet classification.

For the case of $md(\pi)$, consider this. Let a packet P_i be classified as i, that

is, in position i among the leaf nodes when read from left to right. The packet P_j that follows P_i in the approximately sorted list could be in the wrong place (with respect to total sortedness) by at most $md(\pi)$ to the right and to the left by the same distance. Therefore, the packet P_j could belong to any class between $i - md(\pi)$ and $i + md(\pi)$. Assuming that the decision is a binary tree, we need to start the search for the P_j class from a subtree where the number of leaf nodes is $2 \times md(\pi) + 1$. The minimum height of a binary tree with $2 \times md(\pi) + 1$ leaf nodes is $h = \log(2 \times md(\pi) + 1)$. This implies that we need to search for the height tree h, which could be considerably smaller than the binary tree of minimum height with all leaf nodes (equal to the number of packet classifications).

8.3 Relationship between Bursty Throughput and maximum displacement

In a network, there may be an unexpected or sudden surge in the number of packets received. This may be due to various seasonal factors. This can contribute to the packet not being received in a sorted or expected manner. Some packets may not be arriving in the sequences that they are expected to arrive in and will only be off by a certain amount of packets as long as the burstiness is prolonged. Let us assume that bursty throughput occurs and is present for a time t_b. This means that packets (n_b number of packets) that arrive in this time frame may not be ordered or arrive in the expected order.

All packets that are out of order will occur within these n_b packets and will not go beyond the burst time frame. This makes the maximum displacement (md) of this set of packets at most n_b. This observation shows that md is directly

proportional to t_b.

8.4 Approximate Sorting and Packet Classification

We will consider a list of packets n in the buffer B and let C be the number of packet classes. We will also assume that there are $R = C$ rules, which implies that each rule produces a unique class for the packet. Each of these rulesets has been chosen from the simulated sets created by **(Taylor and Turner, 2007)**. We test on three types of rulesets: ACL (Access Control Lists), IPC (IP Chain) and FW (Firewall), each of sizes ranging from 100 to 1 million rules from **(Taylor and Turner, 2007)**. These have been widely accepted as a tool for simulating benchmark rulesets. When we do not sort the packets (unsorted case), each packet will have to go through the decision tree, and the total time consumed will be $O(n \times \log R)$. We set this as a baseline calculation and compare the results with this.

When we fully sort the packets, we will consume $O(n \times \log n)$ time to sort. After sorting, take the first packet and we will take $O(\log R)$ time to find its classification. If the first packet is classified at l (l the leaf node), then the next packet will be classified on or after the l classification. In this case, the tree for which we need to search will have only $R - l$ leaf nodes, giving us a height of $\log(R - l)$. But in the worst case, l could be as small as 1. Given this approach, the worst-case number of computations after sorting will be $\Sigma_{i=0}^{n-1}(\log(R - i))$, given a total time of

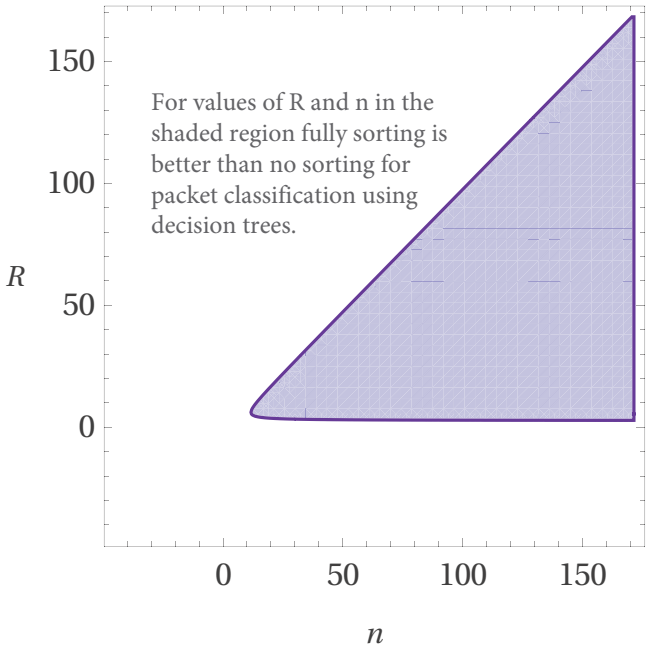


Figure 8.2: Packet Classification with fully sorted versus unsorted packets list.

$$C = \sum_{i=0}^{n-1} (\log(R-i)) + n \times \log n \quad [FullySorted]$$

For the case of fully sorted packets, we need to show $C < n \times \log R$. This equation can be rewritten as $\log(\frac{R!}{(n-2)!}) + n \times \log n < n \times \log R$. In further rewriting, we will have $\frac{R!}{(n-2)!} < (\frac{R}{n})^n$. The graph in Fig. 8.2 shows the values for R and n that this condition will be true. As you can see, for example, when $n = 100$ and $R = 50$ the fully sorted ones perform between the unsorted ones. Note that these can be scaled proportionally and that the condition will still hold.

Next, we focus our attention on the case of approximately sorted packets A. The Algorithm 2 presented is explained below. In Step 1 we perform the approximate sorting of the packets in A. This consumes $O(D \times n)$ time. We assume that we have $|R| = m$ rules and that $m > n$. We further assume that these rules are in sorted order on the decision tree left-to-right as leaf nodes **(Yingchareontha-**

worchai et al., 2018). In line 3 of the algorithm, we determine the position of the first packet $sPos$. The second packet could be in position $sPos - md$ to some position $k > sPos$ that is based on the result of the approximate sorting of the packets. If the second packet is less than $R[sPos + md]$ then this packet is between $sPos - md$ and $sPos + md$, the decision tree is to search within this window in line 16. On the other hand, if the second packet is greater than $R[sPos + md]$ we move the search window by $2 \times md$ from our current $sPos$. We compare the third packet with the rule at the new position $sPos$ and continue the processing similar to the one we did for the second packet. The SearchDecisionTree at any given time looks at a window of $2 \times md$ rules to search (leaf nodes). Each of these searches costs $\log(2 \times md)$ time. We need to do this for $n - 1$ packets. The searches on the decision tree are skipped if the condition in line 10 is true. The total of such comparisons is at most $n - 1$. The initial search for the first packet will consume $\log |R|$ time. The total time for the approximate sorting-based approach is given below.

$$T_1 = n \times D + \log R + n - 1 + \log(2 \times \frac{n}{2^{D+1}}) \ [ApproxSorted]$$

Next, we focus on comparing the approximate sorting with the unsorted case. We can show that $T_1 < T_2 = n \times \log R$.

By comparing T_1 with T_2 we have

$$\begin{aligned} T_1 - T_2 &= nD + \frac{R}{2^{D+1}} + \log R + (n-1)\log\frac{n}{2^D} - n\log R \\ &= \frac{R}{2^{D+1}} + (n-1)\log n + D - (n-1)\log R \\ &= \frac{R}{2^{D+1}} + D - (n-1)\log\frac{R}{n} \end{aligned}$$

Algorithm 26: PacketClassify ($A[1..n]$, $R[1..m]$, D)

Input: An array of packets $P[1..n]$, n is a power of 2, an array of rules $R[1..m]$ that are assumed to be already sorted, D being the factor for the number of comparisons

```
ParSort (P[1..n], D)
md = n / 2^(D+1)
sPos = SearchDecisionTree (R[1..m], m − 1, P[1])
i = 2
while i < n do
    if (sPos + md) > m then
        ePos = m − 1
    else
        ePos = sPos + md
    if P[i] > R[ePos] then
        sPos = sPos + 2 × md
    else
        if (sPos − md) < 0 then
            sPos = SearchDecisionTree(R, 0, sPos + md, P[i])
        else
            sPos = SearchDecisionTree(R, sPos − md, sPos + md, P[i])
        sPos = sPos + (2 × md)
        i++
```

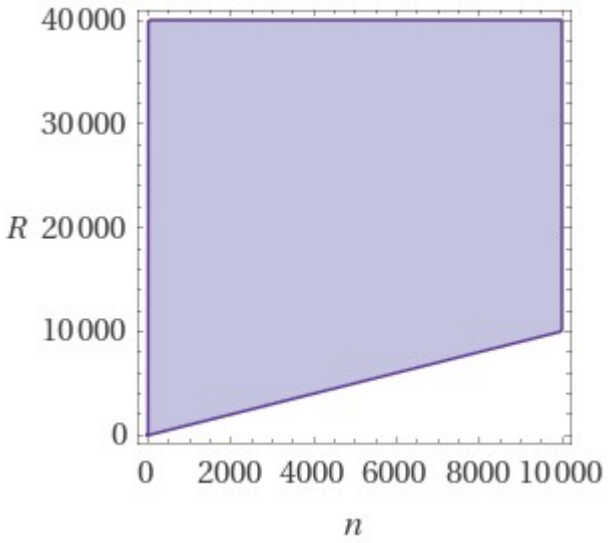


Figure 8.3: Conditions under which the approximate sorting is better than fully sortedness

If $T_1 < T_2$, then $\frac{R}{2^{D+1}} + D < (n-1)\log\frac{R}{n}$.

Furthermore, if we check the improvement of T_1 in percentage compared to T_2, we have the following.

$$1 - \frac{T_1}{T_2} = \frac{(n-1)\log\frac{R}{n} - \frac{R}{2^{D+1}} - D}{n\log R}$$

$$= \frac{n-1}{n} - \frac{(n-1)\log n + \frac{R}{2^{D+1}} + D}{n\log R}$$

The graph in Fig. 8.3 shows areas in which the values of $|R|$ and n could improve the approximate sorting. In the graph in Fig. 8.4 we have provided a comparison between unsorted and approximate sorted methods in terms of actual values (in time units) for the 1 million number of rules and the percentage of improvement for various values of R, n and D.

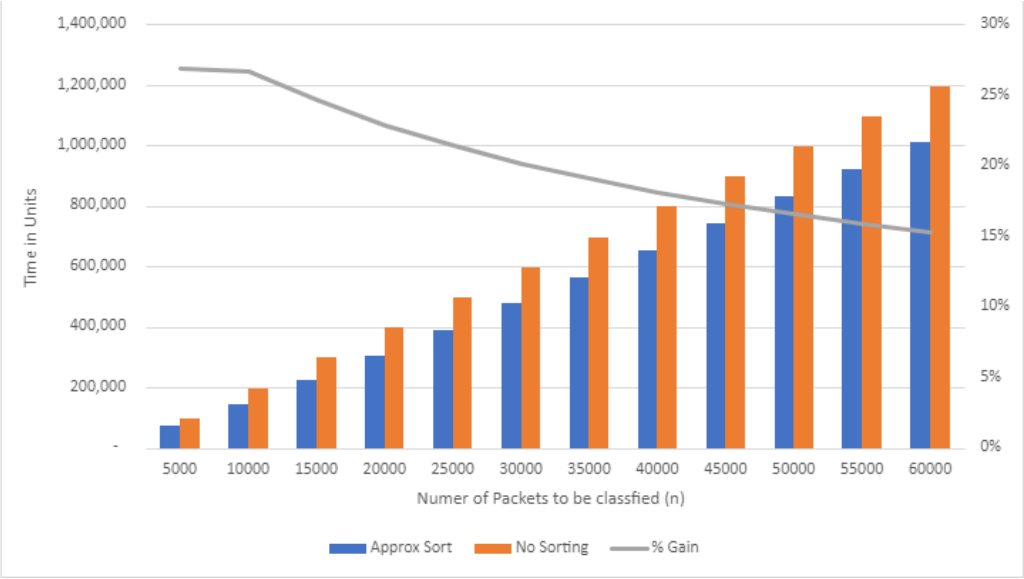
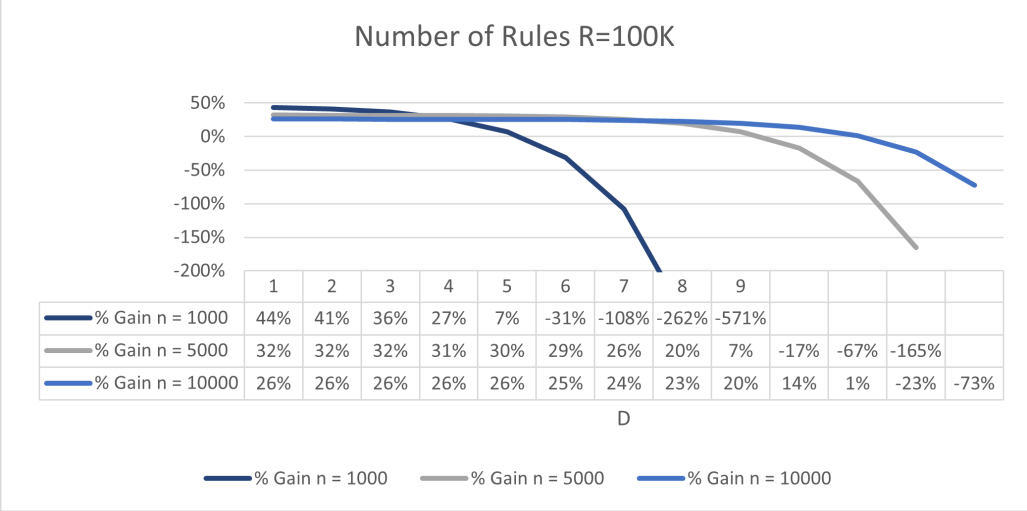


Figure 8.4: Comparing Approximate Sort and Unsorted Methods for 1M number of rules for ACL/IPC/FW. The value of $D = 4$



	1	2	3	4	5	6	7	8	9				
% Gain n = 1000	44%	41%	36%	27%	7%	-31%	-108%	-262%	-571%				
% Gain n = 5000	32%	32%	32%	31%	30%	29%	26%	20%	7%	-17%	-67%	-165%	
% Gain n = 10000	26%	26%	26%	26%	26%	25%	24%	23%	20%	14%	1%	-23%	-73%

Figure 8.5: Varying the value of D and showing the impact on gain in performance. After certain values of D there are no gains

8.5 Improved Packet Classification using a Cache approach

In the previous section, we provided a packet classification algorithm that utilized the maximum displacement metric as an input to classify the incoming packets in the buffer. One of the issues with this approach was that the packets in the buffers were approximately sorted and this maximum displacement was used in the rule tree to categorize the packets. This led to some issues with the classification. Hence in this section, we provide an improved approach to packet classification using an approximate sorting approach of packets in the buffer.

To establish the premise, we analyze the complexity of classifying a packet when the packets are not sorted in the buffer of the router. This is observed to be the following.

$$T_1 = n \times \log_2^d r$$

The above means that each packet (n of them) has to go through the entire rule tree of r rules and d dimensions to get classified. We now present a simple algorithm that sorts the packets fully.

In the PacketClassifyFullSort algorithm, P are all the packets that have to be classified and R are all the rules that are existent. These rules are maintained as a range tree with d dimensions.

We start by sorting the packets fully in the buffer. We assume that all the packets fit in the buffer. If the buffer is too small to fit all the packets, another approach would be to take chunks of packets and perform the same algorithm. $R_{current}$ maintains the most recent rule into which a packet was classified into.

Algorithm 27: PacketClassifyFullSort ($P[1..n]$, $R[1..r]$)

```
Input: An array of packets P[1..n], n is a power of 2, an array of rules
       R[1..r] built as a range tree with d dimension attributes
Sort (P[1..n])
R_current = SearchRangeTree (R[1..r], pathMarkers[1..d], P[1])
i = 2
while i < n do
    if P[i] gets classified into R_current then
        i++
    else
        j = d
        while j > 1 do
            R_current = SearchRangeTreeBottomUP(R[1..r],
             pathMarkers[1..j], P[i])
            if R_current > −1 then
                break;
            else
                j−−
```

So, we first classify the first packet running through the entire rule tree.

Inside the while loop, we start with the second packet and before we proceed to immediately run through the rule tree, we first check if the current packet gets classified into the previous rule into which the previous packet was classified into. Since these packets are all sorted in the buffer based on the same set of attributes/dimensions as the rules, if at all the packets are getting classified into the same rule, it will happen at this step.

If they do not match, then the packet goes through the rule tree using a bottom-up approach to classify itself into the new rule. This new rule is then updated with $R_{current}$. A bottom-up approach is used utilizing the $pathMarkers$ that maintains the root of the tree at each dimension. Starting from the innermost dimension where the previous $R_{current}$ was present makes the average case of finding the new rule much faster.

The complexity of this approach is analyzed to be the following.

$$T_2 = dn \log_2 n + s \log_2^d r + nd$$

The first component in the above equation is the time taken to table-sort the packets in the buffer. The second component is the time taken for the s set of packets that get classified into a new unique rule. This is the worst-case time taken for any packet to run through the entire tree to get classified into a rule. The third component is the time taken to check each packet to see if it matches the rule ($R_{current}$) into which the previous packet was classified into. One way to think about this approach is as if there is a cache of size one that is maintained with only $R_{current}$.

In the next algorithm, we provide a case for approximately sorting the packets in the buffer and still being able to precisely classify them into their corresponding rules.

The algorithm starts with approximately sorting the packets in the buffer with the factor of a number of comparisons D as an input as well. We also immediately calculate the upper bound for the maximum displacement metric that is dependent on D. In this algorithm, we maintain the $R_{current}$ as a cache of recent rules into which packets are classified into. The number of rules in this cache will only be $2md$. This is proved to be enough as the packets that are approximately sorted in the buffer would not differ from an adjacent packet by more than $2md$ along with the corresponding rule. The cache is also maintained as a d dimensional tree for easier searching, but not to forget the additional insertion cost that this structure comes with.

Similar to the previous algorithm, we classify the first packet and add the

Algorithm 28: PacketClassifyApproxSort ($P[1..n]$, $R[1..r]$, D)

Input: An array of packets $P[1..n]$, n is a power of 2, an array of rules $R[1..r]$ built as a range tree with d dimension attributes, D being the factor for the number of comparisons

```
ParSort (P[1..n], D)
md = n / 2^(D+1)
R_current = [ ] - cache memory to store most recent 2md packets and their
 classified rules as a d dimensional tree
R_current = SearchRangeTree (R[1..r], pathMarkers[1..d], P[1])
i = 2
while i < n do
    if P[i] gets classified into a rule in the cache (R_current) then
        i++
    else
        j = d
        while j > 1 do
            R_current = SearchRangeTreeBottomUP(R[1..r],
             pathMarkers[1..j], P[i])
            if R_current > −1 then
                break;
            else
                j−−
```

rule to the cache. As we run through all the packets, we first check the cache to see if the packet gets classified into an existing rule into which another packet was classified into. If not, we run through the entire rule tree with the bottom-up approach to classify the new packet into a new rule. This new rule that is nonexistent in the cache will now be added and swapped into the cache with the oldest rule that was in the cache.

The following is an analysis of the complexity of this algorithm.

$$T_3 = dnD + s\log_2^d r + sd\log_2(2md) + n\log_2^d(2md)$$

The first component is the time taken to approximately sort the d dimensions of packets. The second component is the same as T_2 which is the time taken to classify the new packets into new unique rules. The third component here is the insertion cost of inserting a new rule into the cache. The final component is the time taken to search for a matching rule for a new packet in the cache of $2md$ rules. This has to be done for every packet.

When we compare T_2 and T_3, we derive the following optimality and observe that approach of approximately sorting the packets is better than fully sorting them.

$$T_2 > T_3$$

$$dn\log_2 n + s\log_2^d r + nd > dnD + s\log_2^d r + sd\log_2(2md) + n\log_2^d(2md)$$

$$dn\log_2 n + nd > dnD + sd\log_2(2md) + n\log_2^d(2md)$$

When D approaches $\log_2 n$ for optimality, at first the sorting times get canceled,

$$nd > sd \log_2(2md) + n \log_2^d(2md)$$

But now the maximum displacement md becomes 1 and the log components on the right-hand side are completely gone and vanish. This makes the left-hand side with the nd component that proves that the fully sorting approach is indeed more time taking than the approximate sorting approach.

We conclude this chapter by revisiting the gain in resources that we have when approximately sorting compared to fully sorting or no sorting. The improved approach proves this with the analyses that have been performed. We proceed to conclude the dissertation in the next chapter.

Chapter 9

Conclusion

In this dissertation, we embarked on a comprehensive exploration of approximate sorting algorithms and their applications. We began by defining fundamental quality metrics and establishing upper bounds for each metric using the ParSort sorting algorithm. Additionally, we introduced a modified selection sort algorithm capable of completely sorting an approximately ordered array based on its metric. Building upon these foundations, we delved into analyzing the input arrays for sequences and patterns, leading to the identification of novel quality metrics, such as the longest increasing prefix. Through rigorous analysis, we derived bounds for both new and existing metrics, enhancing our understanding of the approximate sorting process.

Our investigation into searching algorithms for approximately sorted arrays yielded promising results. We developed efficient binary search algorithms that rival the complexity of traditional binary searches on fully sorted arrays. Additionally, we proposed a novel approach for estimating metrics in an approximately sorted array, enabling the application of binary search without explicitly provid-

ing the metric as an input. Furthermore, we extended our exploration to include range search in multidimensional approximately sorted orders. The introduction of modified fractional cascading proved valuable in reducing the complexity of this multidimensional range search, demonstrating the practicality of our findings.

To expand the applicability of approximate sorting, we investigated parallel sorting and searching algorithms. Through careful observation and analysis, we derived valuable insights into the behavior of various metrics in parallel processing environments with constraints on the number of comparisons allocated to each processor. Our work in this area lays the groundwork for future developments in parallel approximate sorting and searching algorithms.

Lastly, we showcased a real-world application of approximate sorting in the context of packet classification for software-defined networks (SDNs). Our improved approach in approximate sorting revealed significant resource savings compared to fully sorting or performing no sorting, as supported by the thorough analyses conducted.

In conclusion, this dissertation presented a comprehensive study of approximate sorting algorithms and their applications. We introduced novel quality metrics, developed efficient and optimal searching and range search algorithms, and explored parallel sorting approaches. Our work demonstrates the potential of approximate sorting in various scenarios, offering resource-efficient solutions while maintaining reasonable acceptable performance. The findings from this research contribute to the advancement of sorting and searching techniques and open up new possibilities for optimizing various applications, including those in SDNs. As we move forward, this work lays the foundation for further research and advancements in the field of approximate sorting and its diverse applications.

Bibliography

Ghulam Abbas, Zahid Halim, and Ziaul Haq Abbas. Fairness-driven queue management: A survey and taxonomy. *IEEE Communications Surveys Tutorials*, 18(1):324–367, 2016. doi: 10.1109/COMST.2015.2463121.

Takao Asano, Masato Edahuroe, Hiroshi Imai, Masao Iri, and Kazuo Murota. Practical use of bucketing techniques in computational geometry. In Godfried T. TOUSSAINT, editor, *Computational Geometry*, volume 2 of *Machine Intelligence and Pattern Recognition*, pages 153–195. North-Holland, 1985. doi: https://doi.org/10.1016/B978-0-444-87806-9.50011-6.

Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7 (4):448–461, 1973.

Milind M Buddhikot, Subhash Suri, and Marcel Waldvogel. Space decomposition techniques for fast layer-4 switching. In *International Workshop on Protocols for High Speed Networks*, pages 25–41. Springer, 1999.

Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M Jungers, and J Ian Munro. An efficient algorithm for partial order production. *SIAM journal on computing*, 39(7):2927–2940, 2010.

Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1):133–162, 1986a.

Bernard Chazelle and Leonidas J Guibas. Fractional cascading: Ii. applications. *Algorithmica*, 1(1):163–191, 1986b.

J. Clement. Global data volume of consumer ip traffic 2017-2022. *Statista, Retreived, April 17, 2022*, Feb 28, 2020.

James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. Tuplemerge:

Fast software packet processing for online packet classification. *IEEE/ACM Transactions on Networking*, 27(4):1417–1431, 2019. doi: 10.1109/TNET.2019.2920718.

Persi Diaconis. Group representations in probability and statistics. *Lecture notes-monograph series*, 11:i–192, 1988.

Persi Diaconis and Ronald L Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(2):262–268, 1977.

Y. Disser and S. Kratsch. Robust and adaptive search. In *Proceedings of the 34th International Symposium on Theoretical Aspects of Computer Science (STACS)*, page 26(14), 2017.

Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

Takashi Fuchino, Takashi Harada, and Ken Tanaka. Accelerating packet classification via direct dependent rules. In *2021 12th International Conference on Network of the Future (NoF)*, pages 1–8, 2021a. doi: 10.1109/NoF52522.2021.9609820.

Takashi Fuchino, Takashi Harada, and Ken Tanaka. Accelerating packet classification via direct dependent rules. In *2021 12th International Conference on Network of the Future (NoF)*, pages 1–8, 2021b. doi: 10.1109/NoF52522.2021.9609820.

Jie Gao, Leonidas J Guibas, John Hershberger, and Li Zhang. Fractionally cascaded information in a sensor network. In *Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 311–319, 2004.

Joachim Giesen, Eva Schuberth, and Miloš Stojaković. Approximate sorting. In *Latin American Symposium on Theoretical Informatics*, pages 524–531. Springer, 2006.

P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000. doi: 10.1109/40.820051.

P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001. doi: 10.1109/65.912717.

Akram Hakiri, Aniruddha S. Gokhale, Pascal Berthou, Douglas C. Schmidt, and Thierry Gayraud. Software-defined networking: Challenges and research opportunities for future internet. *Comput. Networks*, 75:453–471, 2014.

Hazem Hamed and Ehab Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 332–342, 2006.

Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015. doi: 10.1109/MCOM.2015.7045396.

Kanela Kaligosi, Kurt Mehlhorn, J Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *International Colloquium on Automata, Languages, and Programming*, pages 103–114. Springer, 2005.

Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1997. ISBN 0201896834.

Anna Korba. *Learning from ranking data: theory and methods*. PhD thesis, Université Paris-Saclay (ComUE), 2018.

T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28(4):203â214, oct 1998. ISSN 0146-4833. doi: 10.1145/285243.285283. URL `https://doi.org/10.1145/285243.285283`.

Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2645–2653. IEEE, 2018a.

Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2645–2653. IEEE, 2018b.

H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985. doi: 10.1109/TC.1985.5009382.

Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(1):215–241, 1990.

Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

Aditya Narasimhan, Sridhar Radhakrishnan, Mohammed Atiquzzaman, and C. R Subramanian. High-speed packet classification: A case for approximate sorting. *IEEE GLOBECOM*, 2022a.

Aditya Narasimhan, Sridhar Radhakrishnan, and C. R Subramanian. Approximate sorting and sequence analysis. *18th International Conference on Fundamentals of Computer Science*, 2022b.

Aditya Narasimhan, Sridhar Radhakrishnan, and C. R Subramanian. On searching an approximately sorted array. *21st International Conference on Information and Knowledge Engineering*, 2022c.

Ravi S. Prasad, Constantine Dovrolis, and Marina Thottan. Router buffer sizing for tcp traffic and the role of the output/input capacity ratio. *IEEE/ACM Transactions on Networking*, 17(5):1645–1658, 2009. doi: 10.1109/TNET.2009.2014686.

Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224, 2003.

Yufei Tao, Xiaokui Xiao, and Reynold Cheng. Range search on multidimensional uncertain data. *ACM Transactions on Database Systems (TODS)*, 32(3):15–es, 2007.

David E Taylor and Jonathan S Turner. Classbench: A packet classification benchmark. *IEEE/ACM transactions on networking*, 15(3):499–511, 2007.

Da Wang, Arya Mazumdar, and Gregory W Wornell. Compression in the space of permutations. *IEEE Transactions on Information Theory*, 61(12):6417–6431, 2015.

Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, and Eric Torng. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking*, 26(4):1907–1920, 2018. doi: 10.1109/TNET.2018.2852710.

Jun Zhang, Dimitris Papadias, Kyriakos Mouratidis, and Zhu Manli. Query processing in spatial databases containing obstacles. *International Journal of Geographical Information Science*, 19(10):1091–1111, 2005.