Adaptive sorting algorithms for evaluation of automatic zoning

G. S Rajarathinam

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/551
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Adaptive Sorting Algorithms for Evaluation of Automatic
Zoning

by

G. S. Rajarathinam

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Electrical Engineering

Department of Electrical and Computer Engineering
University of Nevada, Las Vegas

December, 1995
The thesis of G. S. Rajarathinam for the degree of Master of Science in Electrical and Computer Engineering is approved.

Chairperson, Shahram Latifi, Ph.D

Examining Committee Member, Yafia Baghzouz, Ph.D

Examining Committee Member, Eugene E. McGAugh, Jr., Ph.D

Graduate Faculty Representative, Junichi Kanai, Ph.D

Graduate Dean, Cheryl L. Bowles, Ed.D

University of Nevada, Las Vegas

December, 1995
ABSTRACT

Optical Character Recognition (OCR) involves analysis of machine-printed and hand written document images. The first step in an OCR process is to locate the text to be recognized on a page. An OCR device tries to identify the characters in these text regions and outputs the characters in ASCII. To evaluate the performance of any OCR device, the ASCII output of the OCR device is compared with the ground truth text which is entered into the computer manually.

Some OCR devices provide the users with automatic zoning. The output of any automatic zoning algorithm has to be corrected manually to restore the correct reading order. This is done by elementary edit operations such as insertions, deletions and substitutions or by moving sub-strings of characters. The efficiency of an automatic zoning algorithm is measured by the cost of correcting the OCR generated text. The model for cost calculation requires movement of sub-strings in a particular fashion to ensure minimal cost. This problem has been modeled as sorting an arbitrary permutation. This thesis presents few adaptive sorting approaches which can be incorporated into the automatic zoning evaluation algorithm. These algorithms perform better than the existing algorithms used for this purpose. This thesis also presents more directions in which the problem can be pursued to achieve better performance.
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Shahram Latifi for his immense help and support extended to me during the course of my graduate program. Without his constant encouragement and guidance, completion of this thesis would have been impossible.

Special thanks to Dr. Yahia Baghzouz and Dr. Eugene E. McGaugh, Jr. for agreeing to be in my thesis committee. I would like to extend my gratitude to Dr. Junichi Kanai and Dr. Thomas A. Nartker for their support during the course of my graduate study. I would like to thank Mr. Stephen Rice for his excellent introduction to the problem and invaluable advice. I would like to thank my colleagues at Information Science Research Institute and my friends at UNLV for their support.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 What is Zoning?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Zoning Metric</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Motivation</td>
<td>4</td>
</tr>
<tr>
<td>2 Permutations: Background and Notations</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Background and Notations</td>
<td>6</td>
</tr>
<tr>
<td>2.2 The Model</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Terminology</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Reducible Permutations</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Irreducible Permutations</td>
<td>9</td>
</tr>
<tr>
<td>2.3.3 Inversion Table</td>
<td>10</td>
</tr>
<tr>
<td>2.3.4 Runs</td>
<td>11</td>
</tr>
<tr>
<td>2.3.5 Longest Ascending Sequence</td>
<td>12</td>
</tr>
<tr>
<td>2.3.6 Difference Vector</td>
<td>13</td>
</tr>
<tr>
<td>3 Conventional Sorting Methods: A Study</td>
<td>14</td>
</tr>
<tr>
<td>3.1 What is Optimal?</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Sorting by Insertion</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1 Straight Insertion Sort</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2 Shellsort</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Sorting by Exchanging</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1 Quicksort</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Sorting by Merging</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Sorting by Selection</td>
<td>21</td>
</tr>
<tr>
<td>4 Adaptive Sorting Algorithms: A Study</td>
<td>22</td>
</tr>
<tr>
<td>4.1 Sorting Presorted Sequences</td>
<td>22</td>
</tr>
<tr>
<td>4.2 Sorting Nearly Sorted Sequences</td>
<td>23</td>
</tr>
<tr>
<td>4.3 me/hort</td>
<td>25</td>
</tr>
<tr>
<td>4.4 Properties of Disorder Measures</td>
<td>27</td>
</tr>
<tr>
<td>4.5 Strip Sort</td>
<td>28</td>
</tr>
</tbody>
</table>
## 5 A Sub-Optimal Sorting Algorithm

5.1 Preliminaries ........................................................................................................................................ 31
  5.1.1 1-reduction .................................................................................................................................. 31
  5.1.2 2-reduction .................................................................................................................................. 32
  5.1.3 3-reduction .................................................................................................................................. 32

5.2 Bounds on Sorting Complexity ............................................................................................................ 32
  5.2.1 Upper Bound .............................................................................................................................. 32
  5.2.2 Lower Bound .............................................................................................................................. 33

5.3 The Algorithms ..................................................................................................................................... 33
  5.3.1 Algorithm Without Look Ahead .................................................................................................. 34
  5.3.2 Algorithm with 1-step look ahead .............................................................................................. 36

5.4 Observations ......................................................................................................................................... 38

## 6 New Adaptive Approaches

6.1 The $O(n!)$ Optimal Algorithm ........................................................................................................ 39

6.2 Basics of Sorting .................................................................................................................................. 41

6.3 Properties of Inversion Table ............................................................................................................ 46

6.4 Properties of Difference Vector ......................................................................................................... 48

6.5 Properties of Runs ................................................................................................................................ 50

6.6 New Approaches .................................................................................................................................. 51
  6.6.1 Inversion Vector Approach ($inv$) .............................................................................................. 51
  6.6.2 Difference Vector Approach 1 ($diff1$) .......................................................................................... 55
  6.6.3 Difference Vector Approach 1 - Non-Propagating ($diff1np$) ....................................................... 56
  6.6.4 Difference Vector Approach 2 ($diff2$) .......................................................................................... 58
  6.6.5 Difference Vector Approach 2 - Non-Propagating ($diff2np$) ....................................................... 60
  6.6.6 Runs Based Approach ($runs$) ....................................................................................................... 62
  6.6.7 Runs Based Approach - Look Ahead ($runsla$) .............................................................................. 63
  6.6.8 Negatives Based Approach ($negs$) ................................................................................................. 65
  6.6.9 Negatives Based Approach - Look Ahead ($negsla$) .................................................................... 65

6.7 Performance Comparison With Optimal Algorithm .......................................................................... 66
  6.7.1 $vola$ and $wla$ ............................................................................................................................. 66
  6.7.2 $inv$ ................................................................................................................................................ 67
  6.7.3 $diff1$ and $diff1np$ ........................................................................................................................ 67
  6.7.4 $diff2$ and $diff2np$ ........................................................................................................................ 68
  6.7.5 $runs$ and $runsla$ .......................................................................................................................... 68
  6.7.6 $negs$ and $negsla$ ........................................................................................................................ 69

6.8 Adaptive Bounds .................................................................................................................................. 70
  6.8.1 Lower Bound .................................................................................................................................. 70
  6.8.2 Upper Bound .................................................................................................................................. 71

6.9 Other Approaches .................................................................................................................................. 73
  6.9.1 Recursive Approach ......................................................................................................................... 73
  6.9.2 Longest Ascending Sequence Approach ......................................................................................... 78
7 Conclusions

BIBLIOGRAPHY
List of Figures

1.1 Zoning Error ................................................................. 2
3.1 Evaluation of Shellsort .................................................. 18
3.2 Example of Quicksort ................................................... 20
3.3 Merging $x_1 \leq x_2 \leq \ldots \leq x_m$ with $y_1 \leq y_2 \leq \ldots \leq y_n$ ................................................... 20
5.1 Flow chart of the sorting algorithm without look ahead .............. 34
5.2 Flow chart of the algorithm with 1-step look ahead ................... 37
6.1 The optimal sorting of $X = 3142$ ..................................... 40
6.2 Patterns to be examined for in Step 4 of Code 6.1 .................... 53
6.3 More patterns to be examined for in Step 4 of Code 6.1 ............. 54
6.4 Patterns which cannot occur in the inversion table ................... 54
List of Tables

6.1 Inversion table approach applied to 13628475 ................................................... 53
6.2 *diff1* applied to 16482753 ........................................................................... 57
6.3 *diff1np* applied to 16482753 ........................................................................ 57
6.4 *diff1* applied to 37158426 ........................................................................... 57
6.5 *diff1np* applied to 37158426 ........................................................................ 58
6.6 Performance comparison of *diff1* and *diff1np* ........................................... 58
6.7 *diff2* applied to 15372648 ........................................................................... 60
6.8 *diff2np* applied to 15372648 ....................................................................... 60
6.9 *diff2* applied to 486193725 ........................................................................ 61
6.10 *diff2np* applied to 486193725 .................................................................... 61
6.11 Performance comparison of *diff2* and *diff2np* ......................................... 61
6.12 *runs* applied to 36148275 .......................................................................... 64
6.13 *runsla* applied to 36148275 ....................................................................... 64
6.14 *runs* applied to 26483175 .......................................................................... 64
6.15 *runsla* applied to 26483175 ....................................................................... 65
6.16 Performance comparison of *runs* and *runsla* ............................................ 65
6.17 Performance comparison of *wola* and *wla* with optimal ......................... 66
6.18 Performance comparison of *inv* with optimal ............................................. 67
6.19 Performance comparison of *diff1*, *diff1np*, and both with optimal .......... 67
6.20 Performance comparison of *diff2*, *diff2np*, and both with optimal .......... 68
6.21 Performance comparison of *runs*, *runsla*, and both with optimal ............ 69
6.22 Performance comparison of *negs*, *negsla*, and both with optimal .......... 69
Chapter 1

Introduction

Optical Character Recognition (OCR) involves analysis of machine-printed and hand written document images. The first step in an OCR process is to locate the text to be recognized on a page. An OCR device tries to identify the characters in these text regions and outputs the characters in ASCII. To evaluate the performance of any OCR device, the ASCII output of the OCR device is compared with the ground truth text which is entered into the computer manually. Several Performance Metrics have been defined and used to evaluate OCR accuracy. These include character accuracy, word accuracy, etc. A detailed definition of all the metrics and experimental results can be found in [1].

1.1 What is Zoning?

Commercial OCR devices allow the user to select text regions, referred to as zones on the page by drawing rectangles around them. The order of these zones is significant and usually corresponds to the correct reading order of the page, specifying the order in which the generated text should appear. This process is known as manual zoning.

On the other hand, some of the OCR devices automatically identify the text regions
and their reading order, and generate text. The device recognizes columns of text and
defines a separate zone for each column of text. The device also identifies graphic regions
and exclude them. This process is known as automatic zoning.

1.2 Zoning Metric

The output generated by a zoning algorithm is normally different from the correct text due
to various reasons. Some of them include misclassification of a text region as a graphic
region or vice-versa, incorrect scanning order chosen by the zoning algorithm, etc. An

\[
\begin{array}{lll}
\text{Text-Line-1} & \text{Text-Line-3} & \text{Text-Line-1} \\
\text{Text-Line-2} & \text{Text-Line-4} & \text{Text-Line-2} \\
\end{array}
\]

\[
\begin{array}{lll}
\text{Text-Line-1} & \text{Text-Line-3} & \text{Text-Line-1} \\
\text{Text-Line-2} & \text{Text-Line-4} & \text{Text-Line-2} \\
\end{array}
\]

(a) Incorrectly Zoned Page          (b) Generated Text          (c) Correct Text

Figure 1.1: Zoning Error

instance of incorrect scanning order is shown in Figure 1.1, where a multi-column page has
been scanned in a row-wise reading order. To evaluate the performance of a given zoning
algorithm, it is of interest to find the cost of correcting the generated text to get the correct
text.

Zone representation schemes are not standardized. The following techniques are
commonly used by OCR devices: bounding rectangles, piece-wise rectangles, polygons, and
nested rectangles. Moreover, the results could be vastly different if a zoning algorithm pro­
cesses a page with and without a deskewing algorithm. These make geometric comparison
of zones infeasible.
Kanai et. al. introduced an automatic zoning metric based on finding all of the text in the text regions and its correct reading order. This metric reflects the deviation of the generated order from the correct one. The number of operations required to transform an OCR output to the correct text is used as the yardstick to measure the performance of a zoning algorithm.

Correcting the OCR generated text has to be done manually. A human editor utilizes three kinds of operations to do this: insertion, deletion, and move. If a zoning algorithm fails to recognize a text region or misclassifies a text region as a graphic region and excludes it, the characters in the text region will be missing from the OCR output. Thus, the editor must insert (type) the missing characters into the OCR output. Alternatively, a graphic region may be misclassified as a text region and the algorithm will output characters in that region. Furthermore, graphic objects could be converted into a set of characters. For example, the vertical axis of a graph might become I's. Unnecessary characters must be deleted from the OCR output.

When a multi-column page is incorrectly zoned as shown in Figure 1.1, Text-Line-3 must be moved between Text-Line-2 and Text-Line-4. This is a move operation. A move operation can be either cut and paste or delete and re-type. The human editor will likely make use of a cut and paste capability to move a block of $n$ characters to its correct location, for $n$ greater than some threshold $T$. But for $n$ less than $T$, it is easier (and less costly) to perform $n$ insertions and $n$ deletions. The value of $T$ varies considerably depending on the skills of the human editor and the editing tools at hand, but is most likely to be in the range of 10 to 100 [2].

If $S_c$ is the string of characters corresponding to correct text of a page and $S_o$, the
string of characters generated by the automatic zoning option of any OCR algorithm, then the cost of transforming the OCR output to correct output is given by:

\[
\text{Cost}(S_0, S_c, T) = W_i \times \text{NumberOfInsertions} + W_d \times \text{NumberOfDeletions} + (W_i + W_d) \times T \times \text{NumberOfMoves}
\]

where \( W_i \) is the cost of correcting an insertion error and \( W_d \) is the cost of correcting a deletion error. In this model, each move is charged a cost of \( (W_i + W_d) \times T \) irrespective of the length of the string being moved and the distance it moves. This value of \( \text{Cost}(S_0, S_c, T) \) also includes cost of correcting character recognition errors which should be eliminated from automatic zoning metric. To eliminate the effects of OCR errors and isolate errors caused by automatic zoning, the calibrated cost is defined as follows:

\[
\text{CalibratedCost}(S_0, S_m, S_c, T) = \text{Cost}(S_0, S_c, T) - \text{Cost}(S_m, S_c, T)
\]

where \( S_m \) is the character string generated from a manually zoned page. The cost of transforming \( S_m \) into the correct text \( S_c \) is equal to \( \text{Cost}(S_m, S_c, T) \) and results from OCR errors only. This calibrated cost effectively isolates the cost of zoning the page automatically.

1.3 Motivation

Kanai et. al. [2] conducted experiments to evaluate the performance of automatic zoning algorithms on six different OCR devices. A set of 460 pages randomly selected from a set of 2,500 documents containing a total of 100,000 pages was used in this experiment. The test
set was divided into three distinct classes namely, *Single column*, *Table* and *Multi-column* pages. *Single column* pages did not produce any significant error due to automatic zoning. But, the authors have reported problems in de-columnization of *Multi-column* and *Table* pages. These problems coupled with the complexity of the page layout give rise to many transposed matches between the generated text and the correct text. This necessitates moving string of characters in order to restore the correct reading order.

It can be seen from the expression for $\text{Cost}(S_o, S_c, T)$ that each move is charged a cost of $(W_t + W_d) \times T$ by the automatic zoning evaluation algorithm. Therefore, it is desirable to make minimum number of moves in correcting the generated text. An algorithm is optimal if it guarantees minimum number of moves in correcting the OCR output. A sub-optimal algorithm has been developed by Latifi [3] and is the base for the automatic zoning evaluation algorithm of [2]. The objective of this work is to develop an optimal algorithm to evaluate the performance of various automatic zoning algorithms.
Chapter 2

Permutations: Background and Notations

As seen in the previous chapter, de-columnization of Multi-column and Table pages is known to be the biggest bottleneck in the performance of any automatic zoning algorithm. The efficiency of an automatic zoning algorithm is judged by the cost of transforming the generated text into correct text. We want to correct the generated text by making minimum number of moves so that the cost remains minimum in majority of the cases. This problem has been modeled as sorting a given permutation in [3].

2.1 Background and Notations

A permutation of a finite set is an arrangement of its elements in all the possible ways. For instance, if $X = \{1, 2, 3\}$ is the finite set, then the permutations of $X$ is given by $S_X = \{(123), (132), (213), (231), (312), (321)\}$.

By a permutation of an arbitrary set $X$ we shall mean a bijection from $X$ to itself. The collection of all permutations of $X$ forms a group $S_X$ under composition functions. If $\alpha : X \rightarrow X$ and $\beta : X \rightarrow X$ are permutations, the composite function $\alpha \beta : X \rightarrow X$ defined by $\alpha \beta(x) = \alpha(\beta(x))$ is also a permutation. Composition of functions is associative, and
the special permutation $I$ which leaves every point of $X$ fixed clearly acts as an identity. Finally, each permutation $\alpha$ is a bijection and therefore has an inverse $\alpha^{-1}: X \to X$, which is also a permutation and which satisfies $\alpha^{-1}\alpha = I = \alpha\alpha^{-1}$. When $X$ consists of the first $n$ positive integers, then $S_X$ is written $S_n$ and called the symmetric group of degree $n$. The order of $S_n$ is thus $n!$.

A permutation can be described by a given order of the $n$ integers enclosed in [ ] with the identity permutation $I$ as the reference.

**Examples:**

\[
(i) \quad I = \begin{bmatrix} 123456789 \\ 123456789 \end{bmatrix}, \quad (ii) \quad \alpha = \begin{bmatrix} 123456789 \\ 174695382 \end{bmatrix}
\]

Note that in $\alpha$ digits 1 and 8 are invariant and map to themselves. For simplicity, we will adopt a cyclic representation for permutation; for instance $\alpha$ can be represented as $(1)(2734659)(8)$.

**Example of an inverse:** The inverse of 591826473 is 359716842, since

\[
\begin{bmatrix} 591826473 \\ 123456789 \end{bmatrix} = \begin{bmatrix} 123456789 \\ 359716842 \end{bmatrix}
\]
2.2 The Model

The following assumptions are made:

1. The correct order of substrings is known.

2. The time to move a substring is independent of the length and location of the substring.

3. A valid move is a deletion followed by an insertion of a single substring.

4. Any two adjacent substrings in the generated string are not adjacent in the correct version.

5. All substrings are distinct.

Under the above assumptions, a text string containing \( n \) substrings can be modeled as a permutation on \( n \) integers \{1,2,\ldots,n\}, where the generated string is represented by \( \pi_G = [a_1 a_2 \ldots a_n] \), \( 1 \leq a_i \leq n \), \( a_i \neq a_j \). The correct string will be the identity permutation or \( \pi_c = I = [12\ldots n] \).

Note that in \( \pi_G \), \( a_{i+1} \neq a_i + 1 \) by assumption 4. If two substrings in \( \pi_G \) are adjacent and are also in the correct reading order, they can be combined and treated as a single entity, justifying assumption 4.

2.3 Terminology

This section discusses few interesting concepts and properties of permutations which will be used in subsequent chapters. For the sake of simplicity, from here on it will be assumed that if \( X \) is a finite set of \( n \) elements, the elements will be the digits 1,2 \ldots, n.

Let \( \alpha = (a_1, a_2, \ldots, a_n) \) be a permutation.
2.3.1 Reducible Permutations

If in $\alpha$, $a_{i+1} = a_i + 1$, where $1 \leq i \leq (n - 1)$, for at least one value of $i$, then $\alpha$ is called a reducible permutation.

Example of a reducible permutation: 24531678.

2.3.2 Irreducible Permutations

$\alpha$ is said to be irreducible if $a_{i+1} \neq a_i + 1$, for $1 \leq i \leq (n - 1)$.

Example of an irreducible permutation: 24351687.

From the description of the model and assumption 4, it is evident that we are interested in irreducible permutations only. Latifi in [3] has derived a formula to find out the number of irreducible permutations for any $n$ using the Principle of Inclusion and Exclusion [4].

$$N_{\text{irreducible}} = \sum_{j=0}^{n-1} (-1)^j \binom{n-1}{j} (n - j)!$$

(2.1)

For $n \gg 1$, the Taylor expansion can be used to find the value of $N_{\text{irreducible}}$ as:

$$N_{\text{irreducible}} = n! \times e^{-1} = 0.36n!$$

(2.2)

The above equation implies that for large $n$, only 36% of the permutations are irreducible. The corresponding proof can be found in [3].

All instances of reducible permutations will be hitherto reduced to form an equivalent irreducible permutation for further processing. For example, if $\alpha = 12435687$, then it will be reduced to 132465.
2.3.3 Inversion Table

Given a permutation \( \alpha = (a_1, a_2, \ldots, a_n) \), an inversion is an instance in \( \alpha \) where \( a_i > a_j \) and \( i < j \).

The inversion table \((b_1, b_2, \ldots, b_n)\) of the permutation \((a_1, a_2, \ldots, a_n)\) is obtained by letting \( b_j \) be the number of \( a_i \)'s to the left of \( j \) such that \( a_i > j \) [5].

For example, the permutation

\[
4261375
\]

has the inversion table

\[
3120200
\]

Properties of the Inversion Table

The inversion table has the following interesting properties:

- By definition, we will always have

\[
0 \leq b_1 \leq n - 1, \quad 0 \leq b_2 \leq n - 2, \ldots, 0 \leq b_{n-1} \leq 1, \quad b_n = 0 \quad (2.3)
\]

- An inversion table uniquely determines the corresponding permutation.

For example, if the inversion table is 40210, then the corresponding permutation can be constructed by starting from left to right. As \( b_1 = 4 \), \( a_5 = 1 \). \( b_2 = 0 \implies a_1 = 2 \).

Inserting the other digits similarly, we get 25431.

- Total number of inversions in \((a_1, a_2, \ldots, a_n)\) = \( \sum_{i=1}^{n} b_i \quad (2.4) \)
If we interchange two adjacent elements of a permutation, the total number of inversions increases or decreases by unity and the reason for this is obvious.

- The inverse of a permutation has exactly as many inversions as the permutation itself.

The proof of this can be found in [5].

Alternatively, the inversion table \((b_1, b_2, \ldots, b_n)\) of the permutation \((a_1, a_2, \ldots, a_n)\) is obtained by letting \(b_j\) be the number of \(a_i\)s to the right of \(j\) such that \(a_i < j\).

For example, the permutation

\[
4261375
\]

has the inversion table

\[
0103031
\]

In this case, we will have

\[
b_1 = 0, \ 0 \leq b_2 \leq 1, \ldots, 0 \leq b_{n-1} \leq n - 2, \ 0 \leq b_n \leq n - 1
\]  \quad (2.5)

All the above properties remain the same. This definition of the inversion table will be used in chapter 6.

2.3.4 Runs

Ascending runs or "runs up" in a permutation is used as a method of testing the randomness or pre-sortedness of a sequence [5]. If we place a vertical line at both ends of a permutation \((a_1, a_2, \ldots, a_n)\) and also between \(a_j\) and \(a_{j+1}\) whenever \(a_j > a_{j+1}\), the runs are the segments between pairs of lines.
For example the permutation

\[4|26|137|5\]

has four runs.

We can use any given permutation on \{1, 2, \ldots, n - 1\} to form \(n - 1\) new irreducible permutations, by inserting the element \(n\) in all possible places barring the one after \((n - 1)\), as this will give rise to a reducible permutation. If the original permutation has \(k\) runs, exactly \((k - 1)\) of these new permutations will have \(k\) runs; the remaining \((n - k)\) will have \((k + 1)\) runs, since we increase the number of runs unless we place the element \(n\) at the end of an existing run. For example, consider the case of inserting 7 in 426135 which has three runs. This will give rise to

7426135, 4726135, 4276135, 4261735, 4261375, 4261357

It is clear that all the permutations but the second and sixth have four runs instead of three.

2.3.5 Longest Ascending Sequence

A longest ascending sequence of a permutation \((a_1, a_2, \ldots, a_n)\) is a sequence constructed by accumulating the elements from the original sequence such that \(a_i < a_j\), for \(j > i\) and \(1 \leq i \leq n - 1\), \(i + 1 \leq j \leq n\) \([9]\).

The longest ascending sequence for any given permutation is not unique. There may be many longest ascending sequences.
For example, if the permutation is 26418375, the longest ascending sequences are

268, 267, 248, 247, 245, 137, 135

2.3.6 Difference Vector

The difference vector \( (d_{12}, d_{23}, \ldots, d_{(n-2)(n-1)}, d_{(n-1)n}) \) of the permutation \( (a_1, a_2, \ldots, a_n) \) is constructed as follows:

\( d_{ij} \) is the number of elements between \( i \) and \( j \). In the permutation, if \( i \) follows \( j \), then \( d_{ij} \) has a negative value. If \( i \) immediately follows \( j \), \( d_{ij} \) is denoted by \( 0^- \). If \( j \) immediately follows \( i \), \( d_{ij} \) is denoted by \( 0^+ \). For simplicity, \( d_{i(i+1)} \) will be referred to as \( d_i \) and is the number of elements between \( i \) and \( (i + 1) \) in the permutation.

If the given permutation is 4261375, the difference vector is (-1,2,-3,5,-3,2).

Properties of Difference Vector

- The length of the difference vector is \( (n - 1) \).
- A negative or \( 0^- \) entity in the difference vector corresponds to an inversion in the permutation.
- Given the position of any element, it is possible to uniquely determine the permutation from the difference vector.
Chapter 3

Conventional Sorting Methods: A Study

For application in automatic zoning evaluation algorithm, a sorting algorithm is said to be optimal if it can sort the given permutation in minimum number of moves. For example, if 153624 is the given permutation, an optimal sorting algorithm should be able to sort it in 2 moves as follows:

\[
153624 \xrightarrow{\text{move } 3 \text{ after } 2} 156234 \xrightarrow{\text{move } 234 \text{ after } 1} 123456
\]

With this notion of optimality in mind, let us take a look at the conventional sorting methods. The sorting methods have been divided into five classes according to the nature of their approach. The only sorting method which is relevant to the application is the straight insertion sort [5]. This algorithm will be quantified in terms of the number of moves required to sort a given permutation. Examples will be given to prove its non-optimality. Other popular sorting methods will be reviewed in brief.
3.1 What is Optimal?

The model presented in chapter 2 will form the basis while sorting a permutation using any approach. A Move is defined as removing a single digit and inserting the same digit in some other place. After every move, the permutation will be reduced to form an equivalent permutation of shorter length. The sequence of moves which reduces the length of the permutation to 1 or sorts the permutation in minimum number of moves is the optimal solution.

3.2 Sorting by Insertion

3.2.1 Straight Insertion Sort

Given a permutation \((a_1, a_2, \ldots, a_n)\), straight insertion sort compares \(a_i, 2 \leq i \leq n\), with \(a_1\) through \(a_{i-1}\) and inserts \(a_i\) in its appropriate place [5]. The pseudo code for this is given below:

```plaintext
for (i = 2; i \leq n; i + +) {
    for (j = 1; j \leq (i - 1); j + +)
        if (a_i < a_j) {
            insert a_i before a_j;
            break;
        }
}
```
If \((b_1, b_2, \ldots, b_n)\) is the inversion table of \((a_1, a_2, \ldots, a_n)\), then straight insertion sort algorithm requires

\[
\text{Number Of Moves} = \text{number of non-zero entries in the inversion table}
\]

The reason for the above is obvious as with each move we eliminate all the inversions corresponding to a particular digit. In this definition of inversion table, we are counting the number of elements greater than and to the left of \(a_i\), for \(1 \leq i \leq n\).

**Example:** The inversion table of 24153 is 20200. The straight insertion sort algorithm corrects the order in two moves as follows:

\[
24153 \rightarrow 12435 \rightarrow 12345
\]

On the other hand, the algorithm can be modified to insert numbers from the right end starting with \(n\), for which the pseudo code is shown below:

\[
\text{for } (i = (n - 1); i \geq 1; i --) \{ \\
    \text{for } (j = n; j \geq (i + 1); j --) \\
    \quad \text{if } (a_i > a_j) \{ \\
        \text{insert } a_i \text{ after } a_j; \\
        \text{break;}
    \}
\} 
\]
The number of moves required by the algorithm is again given by

\[ \text{Number. Of. Moves} = \text{number of non-zero entries in the inversion table} \]

The definition of inversion table is to count the number of elements lesser than and to the right of \( a_i \), for \( 1 \leq i \leq n \).

Example: Considering the same permutation given above, we have the inversion table for 24153 as 01021, and this can be corrected in three moves as follows:

\[
\text{24153} \rightarrow 24135 \rightarrow 21345 \rightarrow 12345
\]

This example shows that there could be a difference in the performance of straight insertion sort that depends on the direction in which it progresses.

Let us take a look at an example in which none of these is optimal. Consider the permutation 4261375 for which the inversion tables are 3120200 and 0103031. Using either of the algorithms discussed above, we need four moves to order the permutation whereas the optimal algorithm sorts the sequence in three moves as shown below:

\[
\text{move 2 after 1} \rightarrow \text{move 5 after 4} \rightarrow \text{move 456 before 7}
\]

\[
\text{4261375} \rightarrow 4612375 \rightarrow 4561237 \rightarrow 1234567
\]

This example clearly proves that straight insertion sort is not the solution to the problem and cannot be used in automatic zoning evaluation.
3.2.2 Shellsort

Another popular method which is based on sorting by insertion is called *diminishing increment sort* or *Shellsort* [5]. This algorithm uses a sequence of increments \( h_t, h_{t-1}, \ldots, h_1 \), where \( h_t > h_{t-1} > h_{t-2} > \ldots > h_1 \) and \( h_1 = 1 \). Effectively, during pass \( i, t \geq i \geq 1 \), shellsort uses straight insertion on sequences formed by the elements which are \( h_i \) apart in the original sequence. To count the number of moves, we write the whole sequence at the end of each pass and count the number of positions in which the present sequence differs from the one before the iteration. This is equal to the number of moves except when \( h_i = 1 \). When \( h_i = 1 \), this quantity should be divided by 2 to get the actual number of moves. The choice of \( h_t, h_{t-1}, \ldots, h_1 \) is not unique and a judicious choice can reduce the number of exchanges required to sort a sequence.

Let us consider the example in [5] with the same increment values and calculate the number of moves. The sequence is 83572461 and the increment values are 3,2,1. Figure 3.1 shows the sequence at the end of each iteration and also shows the number of moves required during that iteration. As is obvious from the figure, shellsort requires 14 moves to sort the sequence.

![Figure 3.1: Evaluation of Shellsort](image-url)
correct the sequence, whereas the sequence can be optimally sorted in 5 moves.

3.3 Sorting by Exchanging

Bubble sort is the simplest sorting algorithm which is based on exchanging adjacent elements. Bubble sort exchanges adjacent elements if they are out of order [5]. Therefore, the number of moves required by bubble sort is equal to the total number of inversions in the permutation. This is not optimal as can be seen in 13524. Bubble sort requires three moves, whereas the optimal value is two.

3.3.1 Quicksort

The basic idea of this method is to choose an element called the pivot element from the input permutation \( a_1, a_2, \ldots, a_n \) and move it to the final position it should occupy in the sorted sequence, say position \( s \). While determining the final position, the other elements are rearranged in such a way that there will be no element with a greater value than the pivot to the left of \( s \) and none with a smaller value than the pivot to the right of \( s \). Then \( a_1, a_2, \ldots, a_{s-1} \) and \( a_{s+1}, a_{s+2}, \ldots, a_n \) are sorted independently using the same procedure. The algorithm terminates when there are no more sequences to sort. Quicksort uses the first element \( (a_1) \) as the pivot. A variation of quicksort called Quikersort uses the median as the pivot.

An example of quicksort is shown in Figure 3.2. This example shows that quicksort is not optimal.
3.4 Sorting by Merging

Merging means combination of two or more ordered sequences into one ordered sequence.

The simplest method of doing this is called *straight merge sort* [5]. Straight merge sort on two ordered sequences is illustrated in Figure 3.4. Example of straight merge sort:

Let 35142 be the input permutation. This can be split into three ordered sequences as 35, 14, and 2. The process is shown below. The sequence is written down after each move. If consecutive integers are in adjacent positions, they are treated as a single entity as usual.

\[
\begin{align*}
35142 & \rightarrow 13542 \rightarrow 12354 \rightarrow 12345 \\
\end{align*}
\]
This is not optimal as the sequence can be sorted in two moves optimally.

3.5 Sorting by Selection

Heapsort is the most popular sorting algorithm which is based on sorting by selection. It constructs a heap during every iteration. The sequence \(a_1, a_2, \ldots, a_n\) is a "heap" if

\[
a_{\lfloor i/2 \rfloor} \geq a_i \quad \text{for} \quad 1 \leq \lfloor i/2 \rfloor < i \leq n
\]

Thus, \(a_1 \geq a_2, a_1 \geq a_3, a_2 \geq a_4, \text{ etc.}\) This ensures that the largest value appears at the top of the heap.

\[a_1 = \max(a_1, a_2, \ldots, a_n)\]

At the end of every iteration, the value \(a_1\) is removed and placed at its appropriate place in the sorted sequence. After the move, this value is replaced by \(-\infty\) and the process is repeated. Data structures and book keeping methods aside, this reduces to a straight insertion sort which cannot be used to sort a sequence optimally.
Chapter 4

Adaptive Sorting Algorithms: A Study

In the previous chapter, we reviewed conventional sorting methods from the viewpoint of the application on hand and came to a conclusion that none of those algorithms is optimal. All the algorithms discussed in chapter 3 do not take the presortedness of a sequence into account before sorting them. In this chapter, we shall study the algorithms which calculate some measure of disorder and use that information in sorting. These measures of disorder are used to quantify the randomness of a given sequence. Such algorithms are called adaptive sorting algorithms in the literature.

4.1 Sorting Presorted Sequences

In 1979, Mehlhorn [6] studied the problem of sorting presorted sequences whose complexity is adaptive with respect to a certain measure of disorder.

Mehlhorn chose the number of inversions as the measure of disorder or presortedness. His sorting algorithm is based on straight insertion sort in conjunction with a sophisticated data structure for efficient insertion. He proved that the algorithm runs in $O(n(1 + \log(F/n)))$, where $n$ is the length of the sequence and $F$ is the number of inversions.
in a given sequence. As shown in chapter 3, straight insertion sort is not the solution to the problem on hand. His main contributions include introduction of the concept of adaptive sorting and use of a novel data structure for searching and insertion.

### 4.2 Sorting Nearly Sorted Sequences

Cook and Kim in [7] reported on an empirical study to determine the best sorting algorithm for nearly sorted sequences.

The "best" sorting algorithm is defined as the one whose weighted sum of the number of comparisons, moves, and exchanges (where an exchange is equal to two comparisons or two moves) is minimal.

"Sortedness Ratio" is defined as

\[
\text{Sortedness Ratio} = \frac{n - k}{n}
\]

where \( n \) is the length of the sequence and \( k \) is the length of the longest ascending sequence.

**Example:** If 13758426 is the given permutation, then \( k = 4 \) and \( \text{Sortedness Ratio} = \frac{8-4}{8} = \frac{4}{8} = 0.5 \).

The authors create a random sequence with a given sortedness ratio as follows:

1. Choose \((n - k)\) elements from the identity permutation at random.

2. Insert these elements in an array of \( n \) elements at random.

3. Insert the remaining \( k \) elements in the vacant positions in order, so that the longest ascending sequence is of required length.
4. If one of the \((n-k)\) elements lies in between and has a value between two of the inserted elements of step 3, then the random element is flipped with one of its neighbors so that the sortedness ratio is preserved.

The following algorithms were used in the experiments.

- Straight insertion sort
- Quickersort
- Shellsort
- Straight merge sort
- Heap sort

Sortedness ratios varied from 0 to 0.2 in steps of 0.02. They used random sequences of lengths varying from 50 to 2000 in their simulation. Empirical results show that straight insertion sort performs well for small sequences with small values of sortedness ratio. Quickersort performs well on longer sequences and greater values of sortedness ratio. The authors have suggested a new algorithm which is a combination of straight insertion, quickersort and merging. This algorithm performs better than all the algorithms under review in almost all the cases. The algorithm is summarized as follows:

1. The original list is scanned and pairs of unordered elements are removed and placed in another array.

2. After a pair of unordered elements has been removed, the next pair compared are the elements immediately preceding and immediately following the pair just removed.
3. After the original list is exhausted, the array of unordered pairs of elements are sorted by straight insertion if there are no more than 30 elements and by quickersort otherwise.

4. The two arrays are merged to get final sorted sequence.

Example:

Given permutation: 13758426

Unordered pairs array: 758432

Straight insertion sort on this needs 4 moves. This is followed by merging to get the final result and this will require some moves in addition to 4, where as the optimal algorithm should take only 3 moves to sort the whole string.

4.3 melsort

Skiena [8] used encroaching lists as the measure of presortedness in his algorithm called melsort (Merge Encroaching Lists Sort).

An ordered list consists of ordered set of elements. The head of an ordered list \( l \), \( \text{head}(l) \), is the smallest element in \( l \), and tail(\( l \)) is the largest element in \( l \). An encroaching lists set is an ordered set of ordered lists \( l_1, l_2, \ldots, l_m \) such that \( \text{head}(l_i) \leq \text{head}(l_{i+1}) \) and \( \text{tail}(l_i) \geq \text{tail}(l_{i+1}) \) for \( 1 \leq i < m \). Thus the lists nest or encroach upon one another. Encroaching lists are a generalization of monotone sequences in permutations. Since ordered permutations contain fewer encroaching lists than random ones, the number of such lists \( m \) provides a measure of presortedness. In an encroaching lists set, \( l_1, l_2, \ldots, l_m \) are called sublists.

The construction of encroaching lists is explained below with the help of an example.
Let us consider the permutation 465291387. Initially, \( l_1 \) consists of 4 and the second element (6) fits at the end of \( l_1 \). The third element 5 is between 4 and 6, hence \( l_2 \) is created to hold it. The next three fit on the oldest sublist (which is \( l_1 \)) and are placed there. 3 and 8 fit at the ends of \( l_2 \), but 7 requires a new sublist. The final sublists are:

\[
\begin{align*}
  l_1 & : 12469 \\
  l_2 & : 358 \\
  l_3 & : 7
\end{align*}
\]

**Lemma 4.1** *In the worst case, a permutation of \( n \) elements will result in \( \lceil \frac{n}{2} \rceil \) sublists.*

The proof of this is rather straightforward and can be found in [8].

After constructing the encroaching lists set, the actual sorting algorithm proceeds as follows:

```
while (listcount > 1) {
    if (ODD(listcount)) then
        MERGE(listcount - 1, listcount);
    for (i = 1; i ≤ listcount/2; i + +)
        MERGE(i, (listcount/2) + i);
    listcount = listcount/2;
}
```

Effectively, the whole process performs a straight merge sort on encroaching lists. As we have seen in chapter 3, merge sort is not suitable for our application.
4.4 Properties of Disorder Measures

Mannila [9] has derived properties which should be obeyed by all the measures of disorder.

If $X$ is the permutation and $m$ is the measure of disorder, then

1. $m(X) = 0$, if $X$ is in ascending order.

2. if $X = (x_1, x_2, \ldots, x_n)$, $Y = (y_1, y_2, \ldots, y_n)$ and $x_i < x_j$ if and only if $y_i < y_j$ for all $i$ and $j$, then $m(X) = m(Y)$.

3. if $X$ is a subsequence of $Y$, then $m(X) \leq m(Y)$.

4. if $X < Y$, i.e. every element of $X$ is smaller than every element of $Y$, then $m(XY) \leq m(X) + m(Y)$.

5. for all $a$, we have $m((a)X) \leq |X| + m(X)$.

In his paper, he has defined four measures of disorder which obey these properties. They are as follows:

- $Inv(X) =$ total number of inversions in $X$.

- $Runs(X) =$ number of ascending runs in $X$.

- $Rem(X) =$ minimum number of elements which should be removed to get a sorted sequence.

- $Exc(X) =$ minimum number of exchanges needed to sort a sequence $= n - \text{number of cycles in the cyclic representation of } X$.

Another measure which does not satisfy the properties is

- $las(X) =$ length of the longest ascending sequence in $X = n - \text{rem}(X)$. 

Some of these measures will be discussed in detail with examples in the following section.

Mannila has studied the problem of finding optimal algorithm with respect to a certain measure of disorder. But, his definition of optimality is different from that of what we are looking for. Hence, none of his conclusions is useful in solving the problem on hand.

4.5 Strip Sort

In [10], Estivill-Castro and Wood have defined three measures of disorder and introduce a sorting algorithm which is adaptive to one of them.

Given a permutation $X$, $\text{Rem}(X)$ is the minimum number of elements that should be removed to get a sorted sequence.

**Example:** $X = 1365472$

By removing 5, 4, and 2, we have a sorted sequence. Therefore, $\text{Rem}(X) = 3$.

$\text{Rem}(X) = 0$ if and only if $X$ is sorted and for any $X$, we have $0 \leq \text{Rem}(X) < n$, where $n$ is the length of $X$. The closer, $\text{Rem}(X)$ is to 0, the closer $X$ is to sorted.

$\text{Pos}(X)$ is the number of elements in $X$ which are not in their correct positions.

**Example:** $X = 3142$

$\text{Pos}(X) = 4$ as all the elements are not in their correct positions.

$\text{Pos}(X) = 0$ only if the sequence is sorted and $\text{Pos}(X) = 1$ is not possible. For any $X$, $0 \leq \text{Pos}(X) \leq n$ and $\text{Pos}(X) \leq \text{Rem}(X)$.

$\text{Exc}(X)$ is the minimum number of exchanges needed to sort $X$. 
Example: \( X = 3142 \)

\[
3142 \xrightarrow{31} 1342 \xrightarrow{42} 1324 \xrightarrow{32} 1234
\]

Therefore, \( \text{Exc}(X) = 3 \).

\( \text{Exc}(X) = 0 \) if and only if \( X \) is sorted. For any \( X \), \( 0 \leq \text{Exc}(X) \leq n - 1 \). Another interesting property is \( \text{Exc}(X) \leq \text{Pos}(X) \leq 2\text{Exc}(X) \), the proof of which can be found in [10].

The Algorithm:

This algorithm SPLITs the original sequence into \( X_S \), a sorted sequence and \( X_U \), an unsorted sequence. Then \( X_U \) is sorted using some well known optimal algorithm like heap sort. Then in the next step, JOIN binary merges \( X_S \) and the sorted \( X_U \) to get the final result.

\( X_U \) is constructed in such a way that

\[
\text{Rem}(X) \leq |X_U| \leq 2\text{Rem}(X)
\]

where \( |X_U| \) is the cardinality of the set \( X_U \). The process of constructing \( X_U \) is explained below by way of an example. Let

\[
X = 467215389
\]

Scan \( X \) from left to right placing elements one at a time into \( X_S \) as long as the current element does not destroy its sorted order. If the current element is less than the last element of \( X_S \), then the current element and the last element from \( X_S \) are placed at the end of \( X_U \).
In the above example, this condition arises when we encounter 2. The lists look like

\[ X_S = 46 \text{ and } X_U = 72 \]

at this stage. Continuing in this fashion, we get

\[ X_S = 489 \text{ and } X_U = 726153 \]

We can see that \(|X_U| = 6\) which is within the claimed bounds as \(\text{Rem}(X) = 4\).

In a nutshell, this algorithm is a combination of heap sort and merging, neither of which is suitable for use in automatic zoning evaluation. Similar algorithms can be found in [11].
Chapter 5

A Sub-Optimal Sorting Algorithm

In the previous two chapters, both conventional sorting methods and adaptive sorting methods were reviewed. None of the methods was found to be optimal. This was shown by way of counter examples. Latifi [3] has focussed on the definition of optimality as required by the application. He has developed a couple of ways in which a given sequence can be sorted. In this chapter, his algorithms will be reviewed in detail and examples will be given to prove their sub-optimal behavior. The shortcomings of these algorithms will be explained with the help of examples and reasons for their sub-optimal behavior will be discussed.

5.1 Preliminaries

The model presented in chapter 2 is used in the construction of this algorithm. An \(i\)-reduction is referred to any move that will reduce the length of the sequence by \(i\).

5.1.1 1-reduction

A move which reduces the length of the sequence by 1.

Example:

\[
31542 \xrightarrow{move 1 \ before \ 2} 35412
\]
5.1.2 2-reduction

A move which reduces the length of the sequence by 2.

Example:

\[13542 \xrightarrow{\text{move 2 before 3}} 12354\]

5.1.3 3-reduction

A move which reduces the length of the sequence by 3.

Example:

\[142635 \xrightarrow{\text{move 4 before 5}} 126345\]

Theorem 5.1 The maximum reduction in the length of a sequence after any move is 3.

The proof of this is very straightforward and can be found in [3]

5.2 Bounds on Sorting Complexity

In this section, absolute bounds on the sorting complexities will be derived for any value of \(n\) (length of a sequence).

5.2.1 Upper Bound

Any move should give rise to an \(i\)-reduction, \(1 \leq i \leq 3\). Assuming the worst case of 1-reduction after every move, the upper bound can be determined to be equal to

\[\text{Moves Upper Bound} = n - 1\] (5.1)
An example sequence which requires \((n - 1)\) moves is the flipped permutation of identity, i.e. \(X = (n(n - 1)\ldots321)\).

### 5.2.2 Lower Bound

According to Theorem 5.1, the maximum reduction possible in the length after any move is 3. By this, after every move the reduction in length is 3 at best and thus the following absolute lower bound can be established.

\[
\text{Moves Lower Bound} = \left\lfloor \frac{n - 1}{3} \right\rfloor
\]  

(5.2)

An example where the lower bound is achieved is 1324657. The lower bound for \(n = 7\) is 2. The above sequence can be corrected in 2 moves as follows:

\[
\begin{align*}
1324657 & \xrightarrow{\text{move 3 after 2}} 1234657 & \xrightarrow{\text{move 6 after 5}} 1234567
\end{align*}
\]

### 5.3 The Algorithms

Latifi [3] has developed a couple of approaches to sort a sequence of \(n\) integers. The algorithm proceeds by making a series of valid moves. There is another important theorem which states

**Theorem 5.2** *The choice of a candidate for a 2-reduction or a 3-reduction will not affect the performance of the algorithm.*

The proof of this is based on enumeration.
5.3.1 Algorithm Without Look Ahead

The algorithm is to sort a given permutation of $n$ integers to identity. The input to the algorithm is: $X = (a_1, a_2, \ldots, a_n)$, $1 \leq a_i \leq n$, $1 \leq i \leq n$ and the output is $I = (123\ldots n)$.

The length of the sequence (in terms of the number of elements it contains) is $\ell$ which is initialized to $n$ in the beginning. The flow chart for the algorithm is shown in Figure 5.1.

After any move, the value of $\ell$ reduces by at least one which guarantees termination of the process with complexity bounds derived in the previous section. The process terminates
when $\ell = 1$. Note that, if there is no 2 or 3-reduction available, the algorithm performs a 1-reduction by moving $n$ after $(n - 1)$. The sequence is scanned from left to right to detect the presence of a 2-reduction. A 2-reduction can be detected by looking for patterns of the form $x, y, (x + 1)$ or $x, (x + 2)$. After detecting any of these patterns, a 2-reduction can be achieved by moving $y$ before $(y + 1)$ or moving $(x + 1)$ before $(x + 2)$. To find whether the move could give a 3-reduction, we will have to check for patterns $(y - 1), (y + 1)$ or $y, x, (y + 1)$. These patterns can be searched for in linear time. Therefore, the algorithm will have a complexity of $O(Number \ of \ Moves \times n)$. Given the upper bound on the number of moves, the worst case complexity can be established as $O((n - 1) \times n) = O(n^2)$.

Why is this not optimal?

The non-optimality will be demonstrated with the help of an example. Let $X = 5436217$.

```
5436217 move 7 after 6 5436721 (543621) move 6 after 5 564321 (54321) move 5 after 4
45321 (4321) move 4 after 3 3421 (321) move 3 after 2 231 (21) move 2 after 1 12 (1)
```

The above algorithm used 6 moves to sort $X$, whereas it can be sorted optimally in 4 moves as follows:

```
5436217 move 2 after 1 5436127 (432516) move 1 before 2 4312 (3214) move 3 after 2
2314 (213) move 2 after 1 123 (1)
```

The shortcoming of this algorithm is the 1-reduction which is performed when there are no 2 or 3-reductions. The algorithm moves $n$ after $n - 1$ without checking their relative
positions. In the above example, 5 is followed by 6 and 6 is followed by 7. But the algorithm
ignored it and used two moves in bringing them together. On the other hand, the optimal
algorithm moved the other elements efficiently in a way which uses the inherent sortedness
of 5, 6, and 7. Under such circumstances, this algorithm fails to use the presortedness of the
input sequence and performs sub-optimally.

5.3.2 Algorithm with 1-step look ahead

This algorithm is essentially the same as the one without look ahead except when there are
no 2 or 3-reductions in the sequence. When there are no 2 or 3-reductions, this algorithm
does not necessarily move $n$ after $n - 1$. It looks ahead 1-step to find out if a 2 or 3-
reduction can be set up by performing a 1-reduction. Such a 1-reduction is called Favored
1-reduction. If a Favored 1-reduction exists, it is given priority over the movement of $n$ after
$n - 1$. The flow chart for this is shown in Figure 5.2. In searching for a Favored 1-reduction,
the algorithm has to look for one of the following patterns:

- $x(x + 3)$
- $xy(x + 2)$
- $xyz(x + 1)$

This can be searched for in linear time and the worst case complexity can be established as
$O(n^2)$. 
Figure 5.2: Flow chart of the algorithm with 1-step look ahead
Why is this not optimal?

Let $X = 5274361$. Using the flow chart of Figure 5.2, $X$ can be sorted as follows:

$$
5274361 \xrightarrow{\text{move 2 before 3}} 5742361(463251) \xrightarrow{\text{move 5 before 6}} 456321(4321) \xrightarrow{\text{move 4 after 3}} 3421(321) \xrightarrow{\text{move 3 after 2}} 231(21) \xrightarrow{\text{move 2 after 1}} 12(1)
$$

$X$ can be sorted in 4 moves as follows:

$$
5274361 \xrightarrow{\text{move 3 before 4}} 5273461(426351) \xrightarrow{\text{move 4 before 5}} 263451(2431) \xrightarrow{\text{move 3 before 4}} 2341(21) \xrightarrow{\text{move 2 after 1}} 12(1)
$$

In the above case, the algorithm failed to look ahead judiciously. It accepted the first Favored 1-reduction which involved moving 2 before 3. Again 2 is followed by 3 in the input sequence and this turned out to be a redundant move. Moreover, if there are no 2 or 3 or Favored 1-reductions, moving $n$ after $n - 1$ may not always be useful.

### 5.4 Observations

From the examples and analysis of the algorithms, it is obvious that

- In the case of without look ahead, when there are no 2 or 3-reductions, a 1-reduction should be chosen by taking into account the presortedness of the sequence.

- In the case of 1-step look ahead, when there are no 2 or 3-reductions, a Favored 1-reduction or a 1-reduction should be chosen by taking into account the presortedness of the sequence.
Chapter 6

New Adaptive Approaches

In the previous chapter, sub-optimal sorting algorithms developed by Latifi [3] were reviewed and reasons for their sub-optimal behavior were described. In this chapter, an $O(n!)$ optimal algorithm will be presented. New adaptive approaches will be introduced and their performances will be compared with that of optimal and sub-optimal algorithms of the previous chapter.

6.1 The $O(n!)$ Optimal Algorithm

Given a sequence $X$, an optimal algorithm proceeds as follows:

- For any sequence of length $n$, all the possible $n$ moves (by moving $x$ in front of $(x+1)$ or after $(x-1)$, for $1 \leq x \leq n$ are considered. This will give rise to $n$ new sequences, each of length $(n-1)$ or less after the appropriate reduction.

- This procedure is repeated for all the new sequences. The termination condition occurs when we get a sequence of length 1.

- The number of steps along a particular path is the number of moves required to sort the sequence by performing that set of moves.
• The minimum of all the paths is the optimal number of moves. Note that the optimal solution is not unique. There may be many paths which give optimal solution.

This algorithm is implemented as a Depth First Search (DFS) tree. The pseudo code for the optimal algorithm is given below.

An example of the optimal algorithm is given in Figure 6.1. The sequences within the braces are the reduced ones. The children of a particular node are the result of moving and merging the digits, one at a time from left to right. Note that the optimal path which sorts this sequence is not unique. There are eight paths which will give the optimal solution.

![Diagram of the optimal sorting algorithm](image)

**Figure 6.1: The optimal sorting of X = 3142**

To analyze the worst case complexity of the optimal algorithm, we have to count the number of nodes in the tree. Assuming the worst case of getting only a 1-reduction after each and every move, we get $n$ sequences of length $(n - 1)$ each starting with the original sequence. Each of these $n$ sequences will give rise to $(n - 1)$ sequences of length $(n - 2)$
each. Continuing in this manner, the worst case will have

\[ \text{Number of Nodes in the Tree} = 1 + n + n(n - 1) + \cdots + n(n - 1) \ldots 1 \]

\[ = \frac{n!}{n!} + \frac{n!}{(n-1)!} + \cdots + \frac{n!}{1!} \]

\[ = \sum_{i=1}^{n} \frac{n!}{i!} \]

which gives rise to the worst case complexity of \( O(n!) \). The worst case performance can be observed in a sequence of the form \( n(n - 1) \ldots 21 \). This is true since moving any digit reduces the length of the sequence by 1 and gives rise to another sequence of the form \( x(x - 1)(x - 2) \ldots 1 \). This is valid till we get a sequence of length 1.

### 6.2 Basics of Sorting

In [3], it has been proved that the maximum possible reduction by moving a single digit is 3 and the choice of a 2 or 3-reduction cannot affect the performance of the sorting algorithm. These have been stated as Theorems 5.1 and 5.2 in the previous chapter. In this section, few more theorems will be stated and proved. These theorems will be used in the sorting algorithms discussed later.

Two digits \( x \) and \( (x + 1) \) are said to be **conversely sorted** if \( x \) immediately follows \( (x + 1) \) in a sequence.

**Theorem 6.1** Exchanging conversely sorted digits is a necessary move.

**Proof:** First we show, using an example, that if two digits are not conversely sorted (i.e., a substring of the form \( \ldots (x + 1) \ldots x \ldots \), a hasty move of \( x \) before \( (x + 1) \) may not be an optimal move. Then we prove that exchanging conversely sorted digits is always a necessary
move.

Case 1. A pattern of the form \((x + 1)\ldots xz\) exists in the permutation, where \(z > x\). Let us consider forming the substring \(x(x + 1)\ldots(z - 1)z\). There could be an instance where the substring \((x + 1)(x + 2)\ldots(z - 1)\) could be formed and moved as a whole between \(x\) and \(z\). Consider the example 37258164. 1 follows 2 and is separated by 2 digits. Sorting the substring 12...6 can be done in three moves as follows.

\[
\begin{align*}
37258164 & \rightarrow move \ 3 \ before \ 4 \ 72581634 & move \ 34 \ after \ 2 \rightarrow 72345816 \\
 & \rightarrow move \ 2345 \ after \ 1 \rightarrow 78 \ 123456
\end{align*}
\]

The whole string can be sorted in one more move.

On the other hand, a hasty move of 1 before 2 or 2 after 1 will result in sub-optimal solution.

\[
\begin{align*}
37258164 & \rightarrow move \ 1 \ before \ 2 \ 37125864 & move \ 3 \ before \ 4 \ 71258634 \\
 & \rightarrow move \ 34 \ before \ 5 \ 71234586 & move \ 6 \ before \ 8 \ 71234568
\end{align*}
\]

It took four moves to form the sequence 123456. We need one more move to sort the whole sequence. This is sub-optimal and is the result of a hasty move.

Case 2. A pattern of the form \((x + 1)xz\) exists in the permutation, where \(z > x\). Again, consider forming the substring \(x(x + 1)\ldots(z - 1)z\). In order to be able to form the substring \((x + 1)(x + 2)\ldots(z - 1)\) and move it as a whole between \(x\) and \(z\), the substring \((x + 2)\ldots(z - 1)\) has to be moved in between \((x + 1)\) and \(z\). The substring \((x + 1)(x + 2)\ldots(z - 1)\) can be moved as whole between \(x\) and \(z\) after this is done.
Alternatively, the substring \((x + 2)(x + 3) \ldots (z - 1)\) can be moved in between \(x\) and \(z\) and \((x + 1)\) can be moved after \(x\) separately. This will take exactly the same number of moves as forming \((x + 1)(x + 2) \ldots (z - 1)\) and moving it as a whole.

**Example:** Let \(X = 42153\).

\[
\begin{align*}
42153 & \xrightarrow{\text{move 3 before 4}} 34215 & \xrightarrow{\text{move 34 before 1}} 23415 & \xrightarrow{\text{move 234 before 5}} 12345
\end{align*}
\]

We could sort the sequence in three moves by forming 234 and moving it as a whole. In the above example, instead of moving 34 before 1 during the second move, we could have moved it before 5. In the next move, 2 will be moved after 1 to sort the sequence in three moves.

On the other hand, we might come across a pattern of the form \(y(x + 1)x\) in the permutation, where \(y < x\). Consider forming the substring \((y + 1)(y + 2) \ldots x\) and moving it as a whole in between \(y\) and \((x + 1)\). This case is similar to the one presented above. By similar reasoning, it can be proved that exchanging \(x\) and \((x + 1)\) cannot affect the overall performance of an algorithm.

This shows that exchanging conversely sorted digits is a necessary move and can be made as and when available □

A permutation is said to be \(K\)-**conversely sorted** if it contains the following subsequence

\[i(i - 1)(i - 2) \ldots (i - k + 1)\] for \(2 \leq i \leq n\) and \(2 \leq k \leq n\)
Lemma 6.1 The optimal number of moves to sort a K-conversely sorted permutation will be

\[ \text{Moves}_{\text{Optimal}} = \text{Moves}_{\text{Reduced}} + (K - 1) \]

where \( \text{Moves}_{\text{Optimal}} \) is the number of moves required to sort the original permutation optimally, \( \text{Moves}_{\text{Reduced}} \) is the optimal number of moves to sort the reduced sequence obtained by replacing the K-conversely sorted sub-sequence with an appropriate single element.

An adjacent inversion is defined as an inversion between any two successive values. For example, the sequence 4261537 has 3 adjacent inversions (12, 34, and 56).

Theorem 6.2 Moving a single digit can either reduce the number of adjacent inversions in the resulting sequence by at most 1 or leave it unaltered.

Proof: Cases of moving 1 and \( n \) have to be treated differently from that of moving other digits. If 1 is located to the right of 2, by moving 1 in front of 2, the number of adjacent inversions in the resulting sequence will reduce by 1. This move cannot eliminate any other adjacent inversion present in the original sequence. Similarly, if \( n \) is located to the left of \( (n - 1) \), by moving \( n \) after \( (n - 1) \), only the adjacent inversion corresponding to the digits \( (n - 1)n \) will be eliminated.

For any \( x, 2 \leq x \leq (n - 1) \), we need to consider the arrangement \( (x - 1) \) and \( (x + 1) \) to analyze the effect of moving \( x \).

Case 1: A pattern of the form \( (x + 1) \ldots x \ldots (x - 1) \) exists in the sequence. Moving \( x \) in front of \( (x + 1) \) or after \( (x - 1) \) can eliminate only the adjacent inversion corresponding to those digits.

Case 2: A pattern of the form \( x \ldots (x - 1) \ldots (x + 1) \) exists in the sequence. Moving \( x \)
after \((x - 1)\) will eliminate the adjacent inversion corresponding to \((x - 1)x\).

**Case 3:** A pattern of the form \(x \ldots (x + 1) \ldots (x - 1)\) exists in the sequence. Moving \(x\) after \((x - 1)\) eliminates the adjacent inversion corresponding to \((x - 1)x\), but this is neutralized by the creation of an adjacent inversion corresponding to \(x(x + 1)\).

Finally if all the digits are in order, moving any one of them cannot reduce the number of adjacent inversions in the resulting sequence \(\Box\)

**Theorem 6.3** Moving a single digit can either reduce the number of ascending runs in the resulting sequence by at most 1 or leave it unaltered.

**Proof:** An important observation before the proof of this is given as a lemma below.

**Lemma 6.2** Inserting any digit in its appropriate position cannot create a run or reduce the number of runs.

Any digit can be inserted in one of three places with respect to a run. It can be inserted at the beginning, somewhere in the middle, or at the end of a run. It is straightforward to see that none of these can create a run or reduce the number of runs. With this in mind, let us proceed with the proof of Theorem 6.3. We need to consider 4 cases. Let \(X = x_1x_2 \ldots x_n\) be the given permutation.

**Case 1:** Moving \(x_i\), which forms a run by itself. This implies that \(x_{i+1} < x_i < x_{i-1}\).

Moving \(x_i\) cannot link the runs ending with \(x_{i-1}\) and starting with \(x_{i+1}\) as \(x_{i-1} > x_{i+1}\).

Moving \(x_i\) makes it part of already existing run, thus reducing the number of runs by 1.

**Case 2:** Moving \(x_i\), which is at the start of a run. The number of runs will reduce by 1 if \(x_{i-1} < x_{i+1}\), as the two runs get linked to form a single longer run. The number of runs will remain the same if \(x_{i-1} > x_{i+1}\).
Case 3: Moving \( x_i \), which is at the end of a run. The number of runs may reduce by 1 or remain the same. The explanation is similar to that of Case 2.

Case 4: Move \( x_i \), which lies in the middle of a run. Moving \( x_i \) cannot reduce the number of runs as \( x_{i-1} < x_{i+1} \) before and after the move □

Theorems 6.2 and 6.3 will be used to determine the adaptive lower bound on the number of moves to sort any given sequence.

### 6.3 Properties of Inversion Table

If \((b_1, b_2, \ldots, b_n)\) is the inversion table of the given permutation, the following properties can be used in sorting the permutation.

1. \( b_1 = b_2 = \ldots = b_n = 0 \) only if the sequence is sorted.

2. If \( b_{i+1} > b_i \), this will be called a "jump". Each jump corresponds to an inversion in the original permutation. In other words, if \( b_{i+1} > b_i \), then \( i \) follows \( (i + 1) \) in the permutation.

3. By moving any digit, the number of jumps in the resultant inversion table will either reduce by 1 or remain the same. This has been proved in Theorem 6.2.

4. Propagating jump: If a certain move does not eliminate a jump, it is called a propagating jump. This occurs when patterns of the form \( xyz \) occur in the inversion table, where \( x < y, x < z, \) but \( y > z \). An example of propagating jump is given below:

Let \( X = 13725846 \) for which the inversion table is \( 00101042 \). Consider moving 6. If 6 is moved in front of 7, we are propagating a jump as 6 moves in front of 5 which was not the case to start with. Moving 7 also propagates a jump.
5. Non-Propagating jump: A jump which is eliminated by making a certain move is known as non-propagating jump. An example is given below:

Let \( X = 13258746 \) for which the inversion table is \( 00101023 \). By moving 6 before 7 or after 5, a jump is eliminated.

6. The following patterns in the inversion table will give rise to 2 or 3-reductions:

- \( b_{n-1} = b_n + 1 \). This implies that \( n \) and \( (n - 1) \) are separated by a single digit. If \( x \) is the digit between \( n \) and \( (n - 1) \), it can be moved in front of \( (x + 1) \) or after \( (x - 1) \) to get a minimum of 2-reduction.

- \( b_{n-2} = b_n \) and \( b_{n-1} > b_n \). This implies that \( (n - 2) \) is immediately followed by \( n \) and \( (n - 1) \) is located to the left of \( (n - 2) \). Here, \( (n - 1) \) can be moved in between \( (n - 2) \) and \( n \) to get a minimum of 2-reduction.

- \( b_{n-2} = b_{n-1} \) and \( b_n = b_{n-1} + 1 \). This implies that the pattern \( (n - 2)n(n - 1) \) exists in the permutation. In this case, \( (n - 1) \) can be moved in between \( (n - 2) \) and \( n \) to get a 2-reduction.

- \( b_n = b_{n-2} + 1 \) and \( b_{n-1} < b_{n-2} \). This implies that the pattern \( (n - 2)n \) occurs in the permutation and \( (n - 1) \) is located to the right of \( n \). Here again \( (n - 1) \) can be moved in between \( (n - 2) \) and \( n \) to get a minimum of 2-reduction.

Note that these patterns in the inversion table will give rise to 2-reduction or 3-reduction only if the inversion table values correspond to the digits \( n, (n - 1), \) and \( (n - 2) \). These patterns in other digits may or may not give a 2 or 3-reduction. For instance, let \( 4261375 \) be the given permutation for which the inversion table is \( 0103031 \). We can observe one of the above mentioned patterns corresponding to the digits 3, 4, and 5. By moving 4, we cannot
get a 2-reduction in this case. The inversion table of any permutation can be constructed in \( O(n^2) \) time. The implementation is very simple and is shown below.

INVERSIONTABLE(sequence)
{
    for (i = 1; i \leq n; i++) {
        for (j = (i + 1); j \leq n; j++)
            if (a_i > a_j) b_{a_i}++;
    }
}

6.4 Properties of Difference Vector

If \((d_1, d_2, \ldots, d_{n-1})\) is the difference vector of the given permutation, the following properties can be used in sorting:

1. \(d_1 = d_2 = \ldots = d_{n-1} = 0\) only if the sequence is sorted.

2. A negative or a zero entity in the difference vector is called a "negative". A negative corresponds to an inversion in the original permutation, i.e. if \(d_i \leq 0\), then \(i\) follows \((i + 1)\) in the permutation.

3. By moving any digit, the number of negatives in the resultant difference vector will either reduce by 1 or remain the same. This has been proved in Theorem 6.2.

4. Propagating negative: If a certain move does not eliminate a negative, it is called a propagating negative. This occurs when patterns of the form \((x)(-y)(z)\) occur in the difference vector, where \(y > x\) and \(y > z\). Using the same example from the previous
section, \(X = 13725846\) for which the difference vector is \((2)(-1)(4)(-1)(2)(-4)(2)\). The difference value corresponding to the digits 6 and 7 is surrounded by smaller positive values. Hence, moving either 6 or 7 will not reduce the number of negatives by 1.

5. Non-Propagating negative: A negative which is eliminated as a result of a certain move is called a non-propagating negative. For example, \(X = 13258476\) for which the difference vector is \((1)(0)(3)(-1)(3)(0)(-1)\). Moving 6 before 7 or after 5 will reduce the number of negatives by 1.

6. Possible 2-reduction or 3-reduction can be identified by looking for the following patterns in the difference vector:

   - \(d_i = 1, 1 \leq i \leq (n - 1)\). This implies that \(i\) and \((i + 1)\) are separated by a single digit. The digit in between \(i\) and \((i + 1)\) can be moved to its appropriate place to get a minimum of 2-reduction.

   - \(d_{i+1} = -(d_i - 1)\). If \(d_i \leq 0\), \(i\) is immediately followed by \((i + 2)\) and \((i + 1)\) is located to the left of \(i\). If \(d_i > 0, 1 \leq i \leq n, i\) is immediately followed by \((i + 2)\) and \((i + 1)\) is located to the right of \((i + 2)\). In this case \((i + 1)\) can be moved in between \(i\) and \((i + 2)\) to get a minimum of 2-reduction.

**Theorem 6.4** The occurrence of patterns of the form \(d_i = 1, 1 \leq i \leq (n - 1)\) and \(d_{i+1} = -(d_i - 1)\) anywhere in the difference vector indicate the presence of a 2 or 3-reduction.

**Proof:** The formation of the digits in the event of such patterns occurring in the difference vector is given above and the necessary move to get a 2 or 3-reduction is also
given above. This is irrespective of the position of occurrence. This is true because the difference vector is a measure of the number of entries between two digits and hence is a clear indicator of the formation of digits □

Note that this is unlike the case of inversion table. Using the same example, we have the difference vector for 4261375 as (-1)(2)(-3)(5)(-3)(2). The values $d_3$ and $d_4$ which correspond to the formation of 3,4, and 5 do not obey any of the properties mentioned above.

The difference vector can be constructed in linear time. The difference vector is constructed in two steps. In the first step, a position vector is determined as follows.

\[ \text{position}[a_i] = i, \quad 1 \leq i \leq n \]

In the second step, the difference vector is constructed as follows.

\[ d_i = \text{position}[i + 1] - \text{position}[i] + 1 \text{ if position}[i] > \text{position}[i + 1] \]

\[ d_i = \text{position}[i + 1] - \text{position}[i] - 1 \text{ if position}[i] < \text{position}[i + 1] \]

Both these steps take linear time, and hence the whole process takes linear time.

### 6.5 Properties of Runs

The following properties associated with the number of runs (ascending sub-sequences) in a permutation will be used in some of the sorting algorithms discussed later in the chapter.

1. The number of runs in a sorted sequence is 1.
2. By moving any digit, the number of runs in the resulting sequence will either reduce by 1 or remain the same.

3. Moving a digit, which by itself forms a run, will reduce the number of runs by 1.

4. Moving a digit which is not located at the beginning or end of a run cannot reduce the number of runs.

The proof of these properties can be found in the proof of Theorem 6.3. The number of runs in a given permutation can be calculated in linear time as shown below.

```c
NUMBEROFRUNS(sequence)
{
    for (i = 1; i <= (n - 1); i++)
        if (a_i > a_{i+1}) number_of_runs +=;
        number_of_runs +=;
}
```

6.6 New Approaches

In this section, various new approaches will be described and examples will be given to analyze their sub-optimal behavior. At the end of this section, the performance of these algorithms will be compared with that of the optimal algorithm.

6.6.1 Inversion Vector Approach (inv)

This method tries to eliminate a jump during every pass. The patterns in the inversion table are examined and a corresponding move is made. This algorithm tries to sort the
sequence starting from \( n \) because of the reasons mentioned in section 6.3.

---

**Code 1. Inversion Vector Approach**

**Step 1.** Input the given sequence.

**Step 2.** Initialize \( \text{length} = \text{length of sequence} \) and \( \text{number of moves} = 0 \).

**Step 3.** Construct the inversion table.

**Step 4.** The possible patterns with the last three digits of the inversion table and the recommended moves are shown in Figures 6.2 and 6.3. The patterns which can never occur with the last three digits are shown in Figure 6.4.

**Step 5.** Update \( \text{length} \).

\[
\text{length} = \text{length} - i \text{ where } i \text{ is the number of reductions}
\]

**Step 6.** \( \text{number of moves}++ \).

**Step 7.** Determine the reduced sequence.

**Step 8.** Repeat Steps 3 through 7 until \( \text{length} = 1 \).

The inversion table can be constructed in \( O(n^2) \) time. After that, a move can be decided in linear time (linear time is due to the fact that when a certain pattern occurs, the next three digits are examined, otherwise a move can be decided in constant time). The total time required in deciding a move is \( O(n^2) \) which leads to the worst case complexity of \( O(n^3) \).

Let us consider an example to understand the shortcomings of this approach. Let \( X = 13628475 \) be the input sequence. The moves are shown in Table 6.1.
POSSIBLE CASES AND THE CORRESPONDING MOVES WITH THE LAST THREE DIGITS

<table>
<thead>
<tr>
<th>(n-2)</th>
<th>(n-1)</th>
<th>n</th>
<th>Recommended Move</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>This is a straight 2-reduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move (n-1) between n and (n-2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This is a straight 2-reduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move (n-1) between n and (n-2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This is a straight 2-reduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move (n-1) between n and (n-2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This is a straight 2-reduction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If x is the digit between (n-1) and n, move x before (x+1).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move n after (n-1).</td>
</tr>
</tbody>
</table>

Figure 6.2: Patterns to be examined for in Step 4 of Code 6.1

<table>
<thead>
<tr>
<th>Sequence after a move</th>
<th>Reduced sequence</th>
<th>Inversion table</th>
<th>Digit to be moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>13628475</td>
<td>00100313</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>13624785</td>
<td>1362475</td>
<td>0010031</td>
<td>6</td>
</tr>
<tr>
<td>1324756</td>
<td>132465</td>
<td>001001</td>
<td>6</td>
</tr>
<tr>
<td>132456</td>
<td>1324</td>
<td>0010</td>
<td>3</td>
</tr>
<tr>
<td>1234</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Inversion table approach applied to 13628475
POSSIBLE CASES (Continued...)

<table>
<thead>
<tr>
<th>(n-2)</th>
<th>(n-1)</th>
<th>n</th>
<th>Recommended Move</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move n after (n-1) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move (n-1) before n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move (n-1) before n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move (n-1) after (n-2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Proceed with next set of three digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>i.e. (n-3)(n-2)(n-1) and so on.</td>
</tr>
</tbody>
</table>

Figure 6.3: More patterns to be examined for in Step 4 of Code 6.1

IMPOSSIBLE CASES WITH THE LAST THREE DIGITS

* denotes any value

Figure 6.4: Patterns which cannot occur in the inversion table
This algorithm requires 4 moves to sort the sequence, which is one more than the optimal. The shortcomings of this approach are rather obvious. Although, 2-reductions are available initially, we are not able to take advantage of it. As much time is spent in constructing the inversion table, it is not advisable to scan the whole sequence again to find the presence of 2 or 3-reductions. The algorithms discussed in the following sections overcome these deficiencies.

6.6.2 Difference Vector Approach 1 (diff1)

The algorithms based on analyzing difference vector to decide a move try to reduce the number of negatives during every pass.

This is a variation of the Without Look Ahead (wola) algorithm proposed in [3] The algorithm proceeds as follows:

\begin{center}
\textbf{Code 6.2. Difference Vector Approach 1}
\end{center}

\begin{itemize}
\item \textbf{Step 1.} Input the given sequence.
\item \textbf{Step 2.} Initialize \texttt{length} = \texttt{length\_of\_sequence} and \texttt{number\_of\_moves} = 0.
\item \textbf{Step 3.} Construct the difference vector.
\item \textbf{Step 4.} Check for the presence of 2, 3-reduction or any 0 values in the difference vector. If any, make the appropriate move (A value of 0 in the difference vector corresponds to conversely sorted digits and it has been proved in Theorem 6.1 that exchanging of conversely sorted digits is a necessary move).
\item \textbf{Step 5.} If there are none, choose the first available negative (whether it is a propagating negative or not) and make the appropriate move.
\end{itemize}
Step 6. Update length.

\[ length = length - i \text{ where } i \text{ is the number of reductions} \]

Step 7. \textit{number of moves} ++.

Step 8. Determine the reduced sequence.

Step 9. Repeat Steps 3 through 8 until \( length = 1 \).

A move can be decided in linear time. Hence the worst case complexity of this algorithm is \( O(n \times (n - 1)) = O(n^2) \).

6.6.3 Difference Vector Approach 1 - Non-Propagating (\textit{diff1np})

This is a variation of \textit{diff1}. The algorithm is essentially the same except for Step 5 wherein a non-propagating negative is given priority over a propagating negative. This algorithm also has a worst case complexity of \( O(n^2) \), but the constant is higher than that of \textit{diff1}.

Although, the \textit{diff1np} algorithm performs optimally in more cases than the \textit{diff1} algorithm, there are instances in which \textit{diff1} performs better than \textit{diff1np}. Let us consider a couple of examples to understand the behavior of these two algorithms.

Let \( X = 16482753 \). The sequence of moves using \textit{diff1} and \textit{diff1np} are illustrated in Tables 6.2 and 6.3 respectively. In this case, \textit{diff1np} performs better than \textit{diff1}. There are 3 negatives in the difference vector initially. \textit{diff1} does not reduce the number of negatives after the first move and ends up using 5 moves to sort the sequence. On the other hand, \textit{diff1np} identifies a non-propagating negative and makes the corresponding move. This reduces the number of negatives by 1 after the first move and the sequence is sorted in 4
Let us consider an example where \textit{diff1} outperforms \textit{diff1np}. The input sequence is 37158426. The sequence of moves using \textit{diff1} and \textit{diff1np} are illustrated in Tables 6.4 and 6.5 respectively. In this example, \textit{diff1} performs optimally. Although, it does not reduce the number of negatives after the first move, it eliminates one negative in each of the following 3 moves. On the other hand, \textit{diff1np} could reduce the number of negatives
after the first move, but had to propagate twice later ending up sorting the sequence in 5 moves. This suggests that we might be able to perform better with look ahead. Table 6.6 gives the number of cases in which one algorithm performs better than the other for different sequence lengths.

<table>
<thead>
<tr>
<th>Length</th>
<th>diff1 better than diff1np</th>
<th>diff1np better than diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>145</td>
<td>668</td>
</tr>
</tbody>
</table>

Table 6.6: Performance comparison of diff1 and diff1np

There are instances where none of the algorithms is optimal. An example of this is 87426153. This rules out a combination of these two algorithms to achieve optimality.

6.6.4 Difference Vector Approach 2 (diff2)

This is a variation of the With Look Ahead (wla) algorithm of [3]. The algorithm is shown in Code 6.3.
Code 6.3. Difference Vector Approach 2

Step 1. Input the given sequence.

Step 2. Initialize $length = length\_of\_sequence$ and $number\_of\_moves = 0$.

Step 3. Construct the difference vector.

Step 4. Check for the presence of 2, 3-reduction or any 0 values in the difference vector. If any, make the appropriate move.

Step 5. If none of above patterns exist, look for Favored 1-reduction (a 1-reduction which will set up a 2 or 3-reduction in the next move) and make the move.

Step 6. If there are none, choose the first available negative (whether it is a propagating negative or not) and make the appropriate move.

Step 7. Update $length$.

$$length = length - i \text{ where } i \text{ is the number of reductions}$$

Step 8. $number\_of\_moves + +$.

Step 9. Determine the reduced sequence.

Step 10. Repeat Steps 3 through 9 until $length = 1$.

A move can be decided in linear time. Hence the worst case complexity of this algorithm is $O(n \times (n - 1)) = O(n^2)$. 
6.6.5 Difference Vector Approach 2 - Non-Propagating (*diff2np*)

This is a variation of *diff2*. The algorithm is essentially the same except for Steps 5 and 6 wherein a non-propagating negative is given priority over a propagating negative. This algorithm also has a worst case complexity of $O(n^2)$, but the constant is higher than that of *diff2*.

Again, with *diff2* and *diff2np*, *diff2np* performs optimally in more cases than *diff2*. As with the previous case, there are instances where one outperforms the other. Tables 6.7 and 6.8 illustrate one of the cases in which *diff2np* is optimal. The input sequence is $X = 15372648$.

### Table 6.7: *diff2* applied to 15372648

<table>
<thead>
<tr>
<th>Move #</th>
<th>Sequence after a move</th>
<th>Reduced sequence</th>
<th>Difference vector</th>
<th>Digit to be moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15372648</td>
<td>1362547</td>
<td>(2)(-1)(3)(-4)(3)(-1)(3)</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>14235</td>
<td>1324</td>
<td>(1)(0)(1)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1234</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6.8: *diff2np* applied to 15372648

<table>
<thead>
<tr>
<th>Move #</th>
<th>Sequence after a move</th>
<th>Reduced sequence</th>
<th>Difference vector</th>
<th>Digit to be moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15372648</td>
<td>1462537</td>
<td>(2)(1)(-3)(2)(-1)(3)</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>14325</td>
<td>1324</td>
<td>(1)(0)(1)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1234</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tables 6.9 and 6.10 illustrate one of the cases in which *diff2* performs better than *diff2np*. The input sequence is $X = 486193725$.

The behavior of these algorithms can be analyzed on the same lines of *diff1* and
Table 6.9: \textit{diff2} applied to 486193725

<table>
<thead>
<tr>
<th>Move #</th>
<th>Sequence after a move</th>
<th>Reduced sequence</th>
<th>Difference vector</th>
<th>Digit to be moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>461893725</td>
<td>46183725</td>
<td>(3)(-1)(-3)(6)(-5)(3)(-1)</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>456183725</td>
<td>416352</td>
<td>(3)(-1)(-2)(3)(-1)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>163452</td>
<td>1432</td>
<td>(2)(0)(0)</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1423</td>
<td>132</td>
<td>(1)(0)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>123</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10: \textit{diff2np} applied to 486193725

<table>
<thead>
<tr>
<th>Move #</th>
<th>Sequence after a move</th>
<th>Reduced sequence</th>
<th>Difference vector</th>
<th>Digit to be moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>348619725</td>
<td>37518624</td>
<td>(2)(-5)(6)(-4)(2)(-3)(2)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>75186234</td>
<td>531642</td>
<td>(2)(-3)(2)(-3)(2)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>536412</td>
<td>42531</td>
<td>(-2)(1)(-2)(1)</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>45231</td>
<td>321</td>
<td>(0)(0)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>312</td>
<td>21</td>
<td>(0)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textit{diff2np}. Performance comparison of the \textit{diff2} and \textit{diff2np} is presented in Table 6.11.

<table>
<thead>
<tr>
<th>Length</th>
<th>\textit{diff2} better than \textit{diff2np}</th>
<th>\textit{diff2np} better than \textit{diff2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>551</td>
</tr>
</tbody>
</table>

Table 6.11: Performance comparison of \textit{diff2} and \textit{diff2np}

A combination of these two algorithms also cannot be used to achieve optimality as there are cases in which none of these algorithms perform optimally. An example is 74163825.
6.6.6 Runs Based Approach (runs)

Runs based approach checks for one more parameter before making any move in the absence of 2 or 3-reductions. It tries to move a digit which reduces both the number of negatives and the number of runs in the resulting sequence. The algorithm is described in Code 6.4.

---

**Code 6.4. Runs Based Approach**

**Step 1.** Input the given sequence.

**Step 2.** Initialize \( \text{length} = \text{length.of.sequence} \) and \( \text{number.of.moves} = 0 \)

**Step 3.** Construct the difference vector.

**Step 4.** Check for the presence of 2, 3-reduction or any 0 values in the difference vector. If any, make the appropriate move.

**Step 5.** If none available, move all the digits to their appropriate positions and construct the \( \text{RUNS} \) array and \( \text{NEGS} \) array. \( \text{RUNS}[i] \) will contain the number of runs in the sequence obtained by moving digit \( i \). Similarly, \( \text{NEGS}[i] \) will have the number of negatives obtained by moving digit \( i \).

**Step 6.** The \( \text{RUNS} \) and \( \text{NEGS} \) array will be examined and any digit which reduces both the number of runs and negatives by 1 will be chosen. **Step 7.** If none, any digit which reduces the number of runs but not the number of negatives will be chosen.

**Step 8.** If none, any digit which reduces the number of negatives but not the number of runs will be chosen.

**Step 9.** If none, a non-propagating negative will be chosen.

**Step 10.** If none, the first available negative will be chosen.
Step 11. Update length.

\[
length = length - i \text{ where } i \text{ is the number of reductions}
\]

Step 12. number_of_moves ++.

Step 13. Determine the reduced sequence.

Step 14. Repeat Steps 3 through 13 until length = 1.

To analyze the complexity of this algorithm, it is necessary to determine the time taken to decide a move. In the worst case, i.e., there are no 2 or 3-reductions and no 0 values in the difference vector, the algorithm constructs the \textit{RUNS} and \textit{NEGS} arrays. The number of runs and the number of negatives can be computed in linear time for any sequence. Therefore, to construct the arrays, we need time of the order of \( n^2 \). After constructing the arrays, a move can be decided by a single pass through the arrays. Hence, the total time for a move is \( O(n^2 + n) = O(n^2) \). This gives a worst case complexity of \( O(n^3) \).

6.6.7 Runs Based Approach - Look Ahead (\textit{runs}la)

This is essentially the same as \textit{runs} except for Steps 6, 7, and 8. In these steps moves which sets up a 2 or 3-reduction is given priority over a move which does not. This also has a worst case complexity of \( O(n^3) \), but has a higher constant than that of \textit{runs}.

Let us look at a couple of examples to explain the behavior of these two algorithms. Tables 6.12 and 6.13 give the sequence of moves for \( X = 36148275 \) in the case of \textit{runs} and \textit{runs}la respectively.
For the case of 36148275, runs sorts the sequence in 4 moves whereas runs takes 5 moves to sort it. Let us take a look at an example where runs performs better than runsla.

Tables 6.14 and 6.15 give the performance for $X = 26483175$.

Table 6.14: runs applied to 26483175

Table 6.16 gives the performance comparison between runs and runsla.

Instances exist where none of these performs optimally. An example of this is 72584163.
<table>
<thead>
<tr>
<th>Move #</th>
<th>Sequence after a move</th>
<th>Reduced sequence</th>
<th>Number of runs</th>
<th>Number of negative</th>
<th>Digit to be moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26481375</td>
<td>26348175</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2371564</td>
<td>25143</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>25134</td>
<td>2413</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4123</td>
<td>21</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.15: runsla applied to 26483175

<table>
<thead>
<tr>
<th>Length</th>
<th>runs better than runsla</th>
<th>runsla better than runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>43</td>
<td>380</td>
</tr>
</tbody>
</table>

Table 6.16: Performance comparison of runs and runsla

6.6.8 Negatives Based Approach (negs)

This is similar to runs except that Step 7 and Step 8 are exchanged. This has the same complexity as runs.

6.6.9 Negatives Based Approach - Look Ahead (negsla)

This is similar to runsla except that Step 7 and Step 8 are exchanged. This has the same complexity as runsla.

Here also, there are instances where one algorithm outperforms the other and where none of them performs optimally. Examples are not presented because of the similarity of this approach to the previous one.
6.7 Performance Comparison With Optimal Algorithm

In this section each of the algorithm will be compared against the optimal algorithm. For a particular algorithm, three performance parameters were measured:

- Number of permutation for which the algorithm under consideration does not perform optimally (non\_opt).

- The above mentioned quantity as a percentage of the total number of irreducible permutations for a given length (% non\_opt).

- Maximum deviation in terms of the number of moves from the optimal answer (max\_dev).

These three quantities will be placed in the same box in all the Tables shown below.

6.7.1 wola and wla

Table 6.17 shows the performance of the algorithms developed in [3]. The results indicate that the best available algorithm could achieve optimality in 92.03% of the cases for a sequence of length 9.

<table>
<thead>
<tr>
<th>Length</th>
<th>wola</th>
<th></th>
<th></th>
<th>wla</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>non_opt</td>
<td>% non_opt</td>
<td>max_dev</td>
<td>non_opt</td>
<td>% non_opt</td>
<td>max_dev</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>9.09</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>11.32</td>
<td>1</td>
<td>1</td>
<td>1.89</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>66</td>
<td>20.71</td>
<td>2</td>
<td>7</td>
<td>2.29</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>454</td>
<td>25.58</td>
<td>3</td>
<td>88</td>
<td>4.15</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>5713</td>
<td>34.24</td>
<td>3</td>
<td>990</td>
<td>5.93</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>59385</td>
<td>40.34</td>
<td>4</td>
<td>11815</td>
<td>7.97</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.17: Performance comparison of wola and wla with optimal
6.7.2 *inv*

Table 6.18 shows the performance of *inv* compared to optimal.

<table>
<thead>
<tr>
<th>Length</th>
<th>non_opt</th>
<th>% non_opt</th>
<th>max_dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1.62</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>91</td>
<td>4.29</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1360</td>
<td>8.15</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>19248</td>
<td>12.98</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.18: Performance comparison of *inv* with optimal

Although for lengths of 5 and 6, *inv* performed better in more cases than *wla*, for higher values of length *wla* is better. On the other hand, the maximum deviation from optimal is lesser in the case of *inv*.

6.7.3 *diff1* and *diff1np*

The results with *diff1* and *diff1np* are shown in Table 6.19. The third column corresponds to the case where the optimal algorithm performs better than a the combination of both.

<table>
<thead>
<tr>
<th>Length</th>
<th>non_opt</th>
<th>% non_opt</th>
<th>max_dev</th>
<th>non_opt</th>
<th>% non_opt</th>
<th>max_dev</th>
<th>% non_opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0.32</td>
<td>1</td>
<td>1</td>
<td>0.32</td>
<td>1</td>
<td>0.32</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>0.61</td>
<td>1</td>
<td>12</td>
<td>0.57</td>
<td>1</td>
<td>0.57</td>
</tr>
<tr>
<td>8</td>
<td>220</td>
<td>1.32</td>
<td>1</td>
<td>196</td>
<td>1.17</td>
<td>1</td>
<td>1.14</td>
</tr>
<tr>
<td>9</td>
<td>3419</td>
<td>2.31</td>
<td>2</td>
<td>2895</td>
<td>1.95</td>
<td>2</td>
<td>1.86</td>
</tr>
</tbody>
</table>

Table 6.19: Performance comparison of *diff1*, *diff1np*, and both with optimal

From the table, it is obvious that *diff1* and *diff1np* perform better than *wola* (*diff1* and *diff1np* are a variation of *wola*) for all lengths. For a length of 9, *wola* works optimally in
59.66% of the cases, whereas diff1 and diff1np give 97.69% and 98.05% accuracy, respectively. It also performs better in terms of maximum deviation from the optimal. An interesting observation is that, these two algorithms perform better than wla.

### 6.7.4 diff2 and diff2np

These are variations of wla. Table 6.20 shows the performance of diff2, diff2np, and a combination of both.

<table>
<thead>
<tr>
<th>Length</th>
<th>diff2</th>
<th></th>
<th></th>
<th>diff2np</th>
<th></th>
<th></th>
<th>Both</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>non_opt</td>
<td>% non_opt</td>
<td>max_dev</td>
<td>non_opt</td>
<td>% non_opt</td>
<td>max_dev</td>
<td>% non_opt</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0.05</td>
<td>1</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>61</td>
<td>0.37</td>
<td>1</td>
<td>31</td>
<td>0.19</td>
<td>1</td>
<td>0.19</td>
</tr>
<tr>
<td>9</td>
<td>1158</td>
<td>0.78</td>
<td>1</td>
<td>639</td>
<td>0.43</td>
<td>1</td>
<td>0.41</td>
</tr>
</tbody>
</table>

Table 6.20: Performance comparison of diff2, diff2np, and both with optimal

There is an increase in the number of cases where the two algorithms discussed above perform optimally than wla. The accuracy is 99.22% in the case of diff2 and 99.57% in the case of diff2np for sequences of length 9, which are higher compared to the 92.03% achieved by wla. These algorithms perform better in terms of the maximum deviation also.

### 6.7.5 runs and runsla

The performances of runs, runsla, and a combination of both are shown in Table 6.21.

There is an improvement in performance compared to diff2 and diff2np. For sequences of length 9, the accuracies have improved to 99.65% and 99.87% in the case of runs and runsla, respectively. But the maximum deviation in the case of runs is 2 for the
Table 6.21: Performance comparison of runs, runsla, and both with optimal sequences of length 9, whereas it is 1 with $\text{diff}_2$ and $\text{diff}_2\text{np}$.

### 6.7.6 negs and negsla

This is similar to the previous algorithm. The results are shown in Table 6.22.

Table 6.22: Performance comparison of negs, negsla, and both with optimal

negs and negsla give identical results as runs and runsla up to sequences of length 8. There is a slight improvement in the performance for sequences of length 9. negsla could reach an accuracy of 99.88% which is the highest of all the approaches for length = 9. A combination of both could achieve an accuracy of 99.91%.
6.8 Adaptive Bounds

In [3], lower and upper bounds were derived for a given length. These bounds are absolute in the sense that they remain the same for any given sequence of a particular length. In this section, adaptive bounds to sort a particular sequence will be derived.

6.8.1 Lower Bound

From the properties of runs and negatives, it is known that by moving any digit, the number of runs and negatives in the resulting sequence can be reduced by at most 1. The values for the number of runs and the number of negatives in the case of a sorted sequence is 1 and 0, respectively. To determine the lower bound on the number of moves to sort a given sequence, we define three quantities as follows,

- $a_1 = \text{Number of runs in the given sequence} - 1$
- $a_2 = \text{Number of negatives (including zeros) in the difference vector of the given sequence.}$
- $a_3 = \text{Absolute lower bound} = \left\lceil \frac{n-1}{3} \right\rceil$, where $n$ is the length of the given sequence.

Now, the adaptive lower bound is given by

$$Ad\_Lower\_Bound_{Number\_Of\_Moves} = \max(a_1, a_2, a_3) \quad (6.1)$$

Examples:

1. $X = 1324657$
   
   $n = 7$
   
   $a_1 = 2, \quad a_2 = 2, \quad a_3 = 2$
Optimal number of moves = 2.

2. \( X = 3142 \)

\( n = 4 \ a_1 = 2, \ a_2 = 1 \ a_3 = 1 \)

Optimal number of moves = 2.

6.8.2 Upper Bound

To determine the upper bound on the number of moves, the concept of longest ascending sequence is used. Longest ascending sequence is the sorted sequence which is obtained by removing minimum number of elements from the original sequence. Given the length of the longest ascending sequence \( \text{las} \), the upper bound on the number of moves is given by

\[
\text{Ad.Lower.Bound.Number.Of.Moves} = n - \text{length}_{\text{las}}
\]  

(6.2)

This upper bound is valid because of the fact that any sequence can be sorted by moving the digits which are not part of a longest ascending sequence.

The string edit distance algorithm developed in [12] will be used to determine the length of the longest ascending sequence. Given a source string \( X = x_1 x_2 \ldots x_n \) and a destination string \( Y = y_1 y_2 \ldots y_m \), the algorithm calculates the cost of transforming \( X \) into \( Y \) in terms of elementary edit operations (insertions, deletions, and substitutions). The algorithm constructs a matrix of size \( n \times m \). An entry in the matrix \( \text{Cost}_{ij} \) gives the cost of transforming \( x_1 x_2 \ldots x_i \) into \( y_1 y_2 \ldots y_j \). Therefore, the value of the element \( \text{Cost}_{nm} \) gives
the cost of transforming $X$ into $Y$ using the least expensive path. The pseudo-code to construct the $Cost$ matrix is given below.

---

**Algorithm. Edit Distance**

**Step 1.** $Cost(0,0) = 0$

**Step 2.** for $i= 1$ to $n$

\[ \text{do } Cost(i,0) = Cost(i-1,0) + DEL(x_i) \]

**Step 3.** for $j = 1$ to $m$

\[ \text{do } Cost(0,j) = Cost(0,j-1) + INS(y_i) \]

**Step 4.** for $i = 1$ to $n$ and $j = 1$ to $m$

\[ \text{do } m_1 = Cost(i-1,j-1) + SUB(x_i,y_j) \]

\[ m_2 = Cost(i-1,j) + DEL(x_i) \]

\[ m_3 = Cost(i,j-1) + INS(y_j) \]

\[ Cost(i,j) = \min(m_1,m_2,m_3) \]

**Step 4.** Edit distance $= Cost(n,m)$

---

for the application, $X$ is the given sequence and $Y$ is the identity permutation. The cost model is given by

- Cost of deletion ($DEL(x)$) = 1

- Cost of insertion ($INS(y)$) = 1

- Cost of substitution ($SUB(x,y)$) = 2 if $x \neq y$, 0 otherwise.

The length of the longest ascending sequence is given by $n - \frac{Cost(n,m)}{2}$.

**Example:** $X = 3142$
The edit distance matrix is Length of \( las = 4 - \frac{4}{2} = 4 - 2 = 2 \).

\[
\begin{array}{cccc}
\phi & 1 & 2 & 3 & 4 \\
\phi & 0 & 1 & 2 & 3 & 4 \\
3 & 1 & 2 & 3 & 2 & 3 \\
1 & 2 & 1 & 2 & 3 & 4 \\
4 & 3 & 2 & 3 & 4 & 3 \\
2 & 4 & 3 & 2 & 3 & 4 \\
\end{array}
\]

\( Ad_{Upper\_Bound} = 4 - 2 = 2 \).

Optimal number of moves = 2.

The time required to determine the upper bound is \( O(n^2) \) and that for the lower bound is \( O(n) \). The whole process takes a time of \( O(n^2) \).

Ukkonen in [13] presents two algorithms to calculate the edit distance between two strings. The first algorithm works for an arbitrary cost and takes \( O(nd) \) time and space, where \( n \) is the length of each string, and \( d \) is the edit distance between the strings. The second algorithm works for cost function \((1,1,1)\) - i.e., cost of each insertion, deletion, and substitution is one, and takes \( O(nd) \) time but requires only \( O(d^2) \) space.

### 6.9 Other Approaches

#### 6.9.1 Recursive Approach

Given a sequence of length \( (n - 1) \), a new element \( n \) can be inserted in \( (n - 1) \) different places to form \( (n - 1) \) new sequences of length \( n \) each. \( n \) cannot be inserted immediately after \( (n - 1) \) as it will give rise to a reducible sequence.

**Example:**

\( X = 3142 \) We can form 4 new sequences of length 5 each as follows:

53142, 35142, 31542, 31425.
Inserting \( n \) in a sequence of length \((n-1)\) can affect the number of moves to sort the sequence in two ways. The number of moves to sort the sequence of length \( n \) can increase by 1 compared to that of sorting the sequence of length \((n-1)\) or the number of moves remains the same.

Is it possible in any way to judge the effect of inserting \( n \) in a sequence of length \((n-1)\)? If it is possible, the problem can be solved recursively in optimal number of moves. If the effect of introducing \( n \) is to increase a move, then \( n \) can be moved after \((n-1)\) before proceeding with the sequence of length \((n-1)\). If the insertion does not have any effect, we can sort the sequence without moving \( n \).

One possible way to determine the effect of inserting \( n \) is as follows. Given all the optimal paths to sort a particular sequence of length \((n-1)\), is it possible to sort the sequence of length \( n \) by following any one of the optimal paths? If the answer is yes, then the introduction of \( n \) cannot increase the number of moves. Otherwise, the number of moves increases by one, in which case \( n \) can be moved after \((n-1)\) as the first move. But the problem is the number of optimal paths. As shown in section 6.1, the number of optimal paths for the case of 3142 is 8. For sequences of higher lengths, the number of optimal may be numerous and it is not a good proposition to follow all the possible optimal paths. But this allows one to analyze the effect of inserting \( n \).

Let us look at the possibilities and how each insertion affects the number of moves, number of runs, number of negatives in the difference vector, and length of the longest ascending sequence. The effect of this on the bounds will be discussed. \( \text{Lower Bound}(i) \) will be used to denote the lower bound on the number of moves of a length \( i \) sequence. Similarly, the upper bound will be denoted by \( \text{Upper Bound}(i) \).
1. Inserting \( n \) at the beginning of a sequence.

Increase in number of moves = 1. Following any optimal path will lead to 21 after \( \text{Lower Bound}(i - 1) \) moves. The sequence can be completely sorted in one more move.

Increase in number of runs = 1

Increase in number of negatives = 1

Length of the longest ascending sequence remains the same.

\[
\text{Lower Bound}(n) = \text{Lower Bound}(n - 1) + 1.
\]

\[
\text{Upper Bound}(n) = \text{Upper Bound}(n - 1) + 1.
\]

2. Inserting \( n \) at the end of a sequence. This is possible provided that \( (n - 1) \) is not located at the end of the original sequence.

Increase in number of moves = 0. Following any optimal path will lead to 12 after \( \text{Lower Bound}(i - 1) \) moves. The sequence is already sorted by exactly the same sequence of moves which sorted the sequence of length \( (n - 1) \).

Increase in number of runs = 0.

Increase in number of negatives = 0.

Length of the longest ascending sequence increases by 1.

\[
\text{Lower Bound}(n) = \text{Lower Bound}(n - 1).
\]

\[
\text{Upper Bound}(n) = \text{Upper Bound}(n - 1). \text{ Although, there is an increase in the length of the las, it is offset by the increase in length of the sequence.}
\]

3. Inserting \( n \) immediately before \( (n - 1) \).

Increase in number of moves = 1. If moving \( (n - 1) \) is a necessary move, then moving
$(n - 1)$ in front of $n$ or after $(n - 2)$ gives exactly the same reduced sequence as the original sequence of length $(n - 1)$. If $(n - 1)$ need not be moved, then we will have either 21 or 132 after Lower_Bound$(i - 1)$ moves. We need one more move to completely sort the sequence.

Increase in number of runs = 1.

Increase in number of negatives = 1.

Length of the longest ascending sequence remains the same.

Lower_Bound$(n)$ = Lower_Bound$(n - 1)$ + 1.

Upper_Bound$(n)$ = Upper_Bound$(n - 1)$ + 1.

4. Inserting $n$ in a reducible sequence of length $(n - 1)$. We can form an irreducible permutation of length $n$ if the reducible permutation of length $(n - 1)$ has only one pair of elements of the form $a_i a_{i+1}$, where $a_{i+1} = a_i + 1$. For example 15243 can be formed from 1243.

Increase in number of moves = 1. By moving $n$ after $(n - 1)$, we get a 2-reduction.

After this move, the resulting sequence will be of length $(n - 2)$ and exactly equivalent to the original sequence. The first move is the extra move.

Increase in number of runs = 1.

Increase in number of negatives = 0 or 1.

Length of longest ascending sequence may or may not increase.

5. Inserting $n$ before $(n - 1)$.

Increase in number of moves = 0 or 1.
Increase in number of runs = 0 or 1.
Increase in number of negatives = 1.
Length of longest ascending sequence remains the same.

6. Inserting $n$ after $(n - 1)$.
Increase in number of moves = 0 or 1.
Increase in number of runs = 0 or 1.
Increase in number of negatives = 0.
Length of longest ascending sequence may or may not increase.

Change in the number of runs and number of negatives cannot be used to determine the effect of inserting $n$. The following examples present the ambiguity with such an approach.

**Increase in both the number of runs and negatives but not in number of moves:**
$X = 2641375$ requires four moves. Inserting 8 in between 1 and 3 increases the number of runs and negatives but not the number of moves. This is also an example where the insertion of $n$ disrupts a 2-reduction but does not increase the number of moves.

**Increase in the both the number of runs and negatives and also in the number of moves:** Inserting $n$ at the beginning of the sequence.

**Increase in the number of runs, but not in the number of negatives and moves:**
$X = 13524$. Inserting 6 between 2 and 4.

**Increase in the number of negatives, but not in the number of runs and moves:**
$X = 2641375$. Inserting 8 between 4 and 1.
Increase in the number of runs, but not in the number of negatives which increases the number of moves: \( X = 153624 \). Inserting 7 between 2 and 4.

Increase in the number of negatives, but not in the number of runs which increases the number of moves: \( X = 24135 \). Inserting 6 between 4 and 1.

The number of runs, negatives and runs do not increase: Inserting \( n \) at the end of a sequence.

The number of runs and negatives remain the same, but not the moves: \( X = 51324 \). Inserting 6 between 3 and 2.

### 6.9.2 Longest Ascending Sequence Approach

This approach is based on identifying a longest ascending sequence and moving the digits which do not belong to this sequence in an intelligent manner to get the optimal solution.

For example, let \( X = 26418375 \). One of the longest ascending sequences is 268. By moving the other digits, the problem can be solved optimally as follows. The first move is to move 7 in front of 8. This is followed by moving 4 between 3 and 5. Then 345 is moved in between 2 and 6. The last move is to move 1 in front of 2. This sequence is sorted in 4 moves which is optimal.

The core of the algorithm is to identify a longest ascending sequence and the arrangement of the other digits. We need to identify other ascending sequences which is not nested with the \( las \) and can be moved into the \( las \) as a whole. In the above example, although 3 and 5 were not together initially, they were not nested with the \( las \) and could be moved in between 2 and 6.

The problem associated with this sequence is that longest ascending sequence is not unique. There may be many longest ascending sequences co-existing in the given permu-
All the sequences need to be analyzed and the sequence which sorts the sequence in minimum number of moves is optimal. For instance, with 248 as the longest ascending sequence in the above example, this approach will require 5 moves to sort the sequence. In order to analyze all the longest ascending sequences, it is necessary to determine all the longest ascending sequences in a given sequence. This can be done by back tracking the shortest cost path in the edit distance matrix of [12]. The bottleneck is the elimination of redundant paths from the back tracking algorithm. For the example given above, there are 7 distinct longest ascending sequences (248, 247, 245, 268, 267, 137, 135), but there are 67 paths which give the minimum cost.

Another problem associated with this approach is the analysis of worst case time complexity of the algorithm. How many longest ascending sequences of length $k$ can co-exist in a sequence of length $n$?

Although this approach seems to work intuitively, there is a need for a clear set of rules on which the moves will be based.
Chapter 7

Conclusions

The contributions of this thesis towards sorting any arbitrary permutation to be used in the automatic zoning evaluation are:

• Introduced the concept of adaptive sorting which involves pre-processing of the input permutation and making moves based on a clear set of rules.

• Developed and implemented various adaptive approaches towards sorting a given sequence. There is a marked improvement in the performance of these algorithms compared to the existing approaches.

• The time complexity of these algorithms is a maximum of $O(n^3)$, where $n$ is the length of the sequence.

• Developed a $O(n!)$ exhaustive search optimal algorithm which can be used in the performance evaluation of any sorting algorithm.

• Presented ways of determining the bounds on the number of moves adaptively. These bounds depend on the characteristics of the permutation to be sorted as opposed to the absolute bounds presented in [3]. These bounds have been made tighter than the
absolute bounds of [3] by utilizing the properties inherent to specific permutations.

- Presented new ways of analyzing the problem of sorting (Recursive Approach and Longest Ascending Sequence Approach); but the results presented are preliminary and far from conclusive.

Although an optimal solution has not been developed, many avenues have been explored. In the future, Recursive approach and Longest Ascending Sequence approach can be studied in depth to determine if any of those can sort a sequence in polynomial time. Another interesting question is the need for look ahead and how far should one look ahead? Of course, the proof of the problem being NP-complete or not will be the breakthrough in this research.
Bibliography


