Chapter 5

Performance Enhancement of Library Sort Algorithm with Non-Uniform Gap Distribution (LNGD)

Library sort, or gapped insertion sort is a sorting algorithm that uses an insertion sort, but with gaps in the array to accelerate subsequent insertions.

5.1 Objective

Bender et al has suggested the library sort algorithm with uniform gap distribution. But what happens if we have many elements that belongs to the same place in the array and there is only one gap after that element. So to overcome this problem, we have proposed the library sort with non-uniform gap distribution (LNGD).

The proposed algorithm is considered the concept of mean and median. In the proposed technique, non-uniform gap is given based on the property of insertion
sort. This property tells that more updates should be done in the beginning of an array for generating more gaps. LNGD algorithm consists three steps, first two steps are same as LUGD but the third step is different. The LNGD algorithm consists of three steps. The first two steps will be the same as the LUGD algorithm \[77\], but the third step will be different.

**Step1. Binary Search with blanks:** Lets see the working of step 1 with the help of example.

**Example:**

```
1 -1 3 -1 5 -1 7 -1 9 -1
```

In the following array ‘-1’ shows the gaps in the array. The array position is start from 0 up to 9. Now let search an element say 5.

low = 0
high = 9
mid = \((0+9)/2 = 4 = S[4]\)
here \(S[4] = 5\) we got the element and terminate the search.

```
1 -1 3 -1 -1 -1 7 -1 9 -1
```

In this array, we do not have element 5 but we are going to search it.

Here also low = 0
High = 9
Mid == \((0+9)/2 = 4 = S[4]\)
\(S[4] = S[mid] = -1\)
In this case, we have to find \(m1\) and \(m2\) as a mid which are represented by \(S[m1]\) and \(S[m2]\) greater than ‘-1’ in both the direction limiting to low and high respectively. Here the value of \(m1 = S[2] = 3\) and the value of \(m2 = S[6] = 7\). According to \(m1\) and \(m2\) values, we update the low and high to perform binary search.

91
Step2. Insertion: Let’s see the working of step 2 with the help of example.

Example:
We insert the elements in the manner of \(2^i\) in the array. i.e in the power of 2. This is stored in \(S[i]\).

\[ S[i] = \text{pow}(2, i) \] where ‘\(i\)’ is the pass number i.e \(i = 0, 1, 2, 3...\) if \(i = 0\) then \(S1 = 2^0 = 1\). Now we search the position for the insertion element in the array and add the element at position returned by the search function. Next time \(i = 1\) then \(S1 = 2^1 = 2\), and \(S[i] = \text{pow}(2, i-1)\) to \(\text{pow}(2, i)\) i.e the value of \(S1\) is 1 to 2 and so on for all values of ‘\(i\)’.

Step3. Re-balancing: Re-balancing is done after inserting \(2^i\) elements where \(i = 1, 2, 3, 4...\) and the spaces are added when re-balancing is called. In the previous approach, the gaps were uniform in nature. In the proposed technique, non-uniform gap distribution is given based on the property of insertion sort. This property tells that more updates should be done in the beginning of an array for generating more gaps. Gaps are generated using the equation (5.1.2).

\[
Ratio = n * (((\mu/\sigma)/2)
\]

(5.1.1)

Here \(\mu\) is mean and \(\sigma\) is standard deviation.

\[
ee = 2 * (n/ratio)
\]

(5.1.2)

Initially we have \(e+ee\) gaps, but ‘\(ee\)’ is decreased when we have parsed number equal to the ratio.
Algorithm 15 LNGD Re-balancing

INPUT: List of elements n and re-balancing factor e.
OUTPUT: List with non-uniform gaps.

Compute $\mu$ and $\sigma$
Ratio $\leftarrow n \times (\mu / \sigma) / 2$
$ee \leftarrow 2n / \text{Ratio}$
if $(j \% \text{Ratio} == 0 \text{ and } j > 0 \text{ and } e + ee > 0)$ then
  $ee \leftarrow$
end if
while $(l < n)$ do
  if $(S[j] != -1)$ then
    reba[i] = $S[j]$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
    $l \leftarrow l + 1$
    for $(k = 0 \text{ to } ee + e)$ do
      reba[i] = -1
      $i \leftarrow i + 1$
    end for
  else
    $j \leftarrow j + 1$
  end if
  for $(k = 0 \text{ to } i)$ do
    $S[k] = \text{reba}[k]$
  end for
end while

5.2 Performance Evaluation

Execution Time Testing and Comparison of LUGD and LNGD

We have tested the LUGD and LNGD algorithms on a dataset [T10I4D100K(.gz)] [31] by increasing the value of the gap ($\varepsilon$). The dataset contains 1010228 items. We have tested four cases of the data set.
1. Random with repeated data (Random data)
2. Reverse sorted with repeated data (Reverse sorted data)
3. Sorted with repeated data (Sorted data)
4. Nearly sorted with repeated data (Nearly sorted data)
Table 5.1, shows the execution time of LUGD and LNGD algorithms in microsec-
onds using the above mentioned cases.

Table 5.1: Execution Time of Library Sort Algorithm in Microseconds Based on Gap Values

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Random</th>
<th>Nearly Sorted</th>
<th>Reverse Sorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of $\varepsilon$</td>
<td>LUGD</td>
<td>LNGD</td>
<td>LUGD</td>
<td>LNGD</td>
</tr>
<tr>
<td>$\varepsilon = 1$</td>
<td>981267333</td>
<td>802909204</td>
<td>864558882</td>
<td>306063385</td>
</tr>
<tr>
<td>$\varepsilon = 2$</td>
<td>729981576</td>
<td>708580455</td>
<td>620115904</td>
<td>230939335</td>
</tr>
<tr>
<td>$\varepsilon = 3$</td>
<td>119727353</td>
<td>101921406</td>
<td>358670053</td>
<td>185759986</td>
</tr>
<tr>
<td>$\varepsilon = 4$</td>
<td>23003046</td>
<td>10557332</td>
<td>117188830</td>
<td>107729204</td>
</tr>
</tbody>
</table>

The performance of the LUGD and LNGD are compared with random data, nearly sorted data, reverse sorted and sorted data. The execution time in microseconds are presented in Table 5.1. The Results are presented for different value of $\varepsilon$. Epsilon ($\varepsilon$) is the minimum number of gaps between the two elements. The execution time comparison of LUGD and LNGD algorithms has also been shown in Figure 5.1 to 5.4. In all Figure 5.1 to 5.4, the $X$-axis represents the different value of gap and the $Y$-axis represents the execution time in microseconds.

![Figure 5.1: Execution time comparison between LUGD and LNGD using random data](image)

Figure 5.1 shows the comparison of LUGD and LNGD for different values of gap. It can be seen from the graph that the LNGD has outperformed LUGD.
The maximum improvement in execution time by LNGD is 36.7% for the value of $\varepsilon = 4$.

Figure 5.2: Execution time comparison between LUGD and LNGD using nearly sorted data

Figure 5.3: Execution time comparison between LUGD and LNGD using reverse sorted data

Figure 5.2 describes the execution time of the two algorithms LUGD and LNGD on the nearly sorted data. We found major improvement in the case of $\varepsilon = 1$. We also observed that the improvement in execution time by LNGD is 64.59% at $\varepsilon = 1$. With observations, we have found that execution time is
inversely proportional to the value of ‘ε’. The execution time is calculated as 8% in the case of ε = 4.

In the case of reverse sorted data the trend for execution time is reversed. It is nearly 8% for the ε = 1, and it further decreases for ε = 2, ε = 3 and ε = 4 up to 55%. The same has been shown in Figure 5.3.

Figure 5.4: Execution time comparison between LUGD and LNGD using sorted data

Figure 5.4 describes the execution time of both the algorithms on the sorted data, the improvement can be seen from the ε = 1 to ε = 4. It is maximum at ε = 1 that is 63.67% and minimum at ε = 4 that is 8%.

5.3 Re-balancing based Testing and Comparison of LUGD and LNGD

We use re-balancing after inserting $a^i$ element, which increases the size of the array. The size of the array depends on ‘ε’. In this process, we require an auxiliary array of the same size therefore an array having the same values with gaps. We have calculated re-balancing till $a^i$ where $a = 2, 3, 4$ and $i = 0, 1, 2, 3, 4$... with the value of gaps ε = 1, 2, 3, 4.
(A). Example of re-balancing using LUGD algorithm

(1). Example for $\epsilon = 1$ and $a = 2$.

$2^i = 2^0, 2^1, 2^2, 2^3, 2^4$

= 1, 2, 4, 8, 16

(1.1). Re-balance for $2^0 = 1$

\[ \begin{array}{c|c} 1 & -1 \end{array} \]

(1.2). Re-balance for $2^1 = 2$

\[ \begin{array}{c|c} 1 & 2 \end{array} \]

After re-balancing, this array is as follows:

\[ \begin{array}{c|c|c|c} 1 & -1 & 2 & -1 \end{array} \]

(1.3). Re-balance for $2^2 = 4$

\[ \begin{array}{c|c|c|c} 1 & 2 & 3 & 4 \end{array} \]

After re-balancing, the array is:

\[ \begin{array}{c|c|c|c|c|c} 1 & -1 & 2 & -1 & 3 & -1 \end{array} \]

(2). Example for $\epsilon = 1$ and $a = 3$.

$3^i = 3^0, 3^1, 3^2, 3^3, 3^4, \ldots$

= 1, 3, 9, 27, \ldots

(2.1). Re-balance for $3^0 = 1$

\[ \begin{array}{c|c} 1 & -1 \end{array} \]

97
(2.2). Re-balance for $3^1 = 3$
In the above array only one space is empty. This shows that only one element can be inserted. On the other hand, according to re-balancing factor $3^1 = 3$ we require two spaces in the array. In this situation we need to shift the data to make space for the new element. In this way performance of the algorithm degrades as we are having the larger number of swapping to generate the spaces which is same as that in the case of traditional insertion sort.

(B). Example of re-balancing using LNGD algorithm

(1). Example for $a=2$.

$2^i = 2^0, 2^1, 2^2, 2^3, 2^4, ...$

$= 1, 2, 4, 8, 16, ...$

In the proposed algorithm we have used two parameters ‘$ee$’ and ‘ratio’ which is defined prior in the algorithm along with the value of gaps. To understand this concept, we consider an example; say the list to be sorted is 1, 2, 3, and 4. The average and standard deviation are calculated first. The mean and standard deviation are calculated to 2.5 and 1.2 respectively. The ratio and ‘$ee$’ is calculated using equation (5.1.1) and (5.1.2). Ratio = 3. $ee = 8/3 = 2$ as integer The total gaps is 1+2 = 3

(1.1). Re-balance for $2^0 = 1$

After re-balancing, the array is:

```
  1  -1  -1  -1
```

(1.2). Re-balance for $2^1 = 2$

In this case initially $j = 1$ that means we have $e+ee$ gaps that is equal to 3. At second iteration ‘$j$’ is equal to 2, now we have the condition that is $j = ratio$ so we decrement the value of ‘$ee$’ by 1. Initially we have 3 gaps, then 2 gaps.

After re-balancing, the array is:
(1.3). Re-balance for $2^2 = 4$

\[
\begin{array}{cccc}
1 & -1 & -1 & 2 \\
\end{array}
\]

After re-balancing, this array is as follows:

\[
\begin{array}{ccccccc}
1 & -1 & -1 & 2 & -1 & -1 & 4 \\
\end{array}
\]

Initially have 3 gaps for $j=1$.
- For $j=2$, $j$% ratio is equal to zero, therefore ‘ee’ will be decremented by 1.
- For $j=3$, the value remains unchanged to 2 gaps.
- For $j=4$, again the value is decremented by 1 so there is only single gap.

(2). Example for $a = 3$.
$3^i = 3^0, 3^1, 3^2, 3^3, 3^4$ ....
$= 1, 3, 9, 27$....

(2.1). Re-balance for $3^0 = 1$
After re-balancing, the array is described as:

\[
\begin{array}{cccc}
1 & -1 & -1 & -1 \\
\end{array}
\]

(2.2). Re-balance for $3^1 = 3$

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]
After re-balancing, the array is as follows:

\[
\begin{array}{cccccccc}
1 & -1 & -1 & -1 & 2 & -1 & -1 & 3 & -1 & -1
\end{array}
\]

The spaces are calculated using the equation (5.1.2). In the similar manner, re-balancing of the array for the remaining value of the ‘a’ is held till the re-balancing is not possible. The reason for re-balancing not being possible is that in the array according to requirement spaces is not possible. In this way performance of algorithm degrades as we have larger number of swaps to generate the spaces which is same as that in the case of traditional insertion sort.

Table 5.2 describes the execution time of the LUGD and LNGD algorithm using different type of data set that are random data, nearly sorted data, reverse sorted data and sorted data. Along with the different dataset value, the table also describes the value of ‘ε’ and re-balancing factor ‘a’. The re-balancing comparison of LUGD and LNGD algorithms is shown in Figure 5.5 to 5.8. From Figure 5.5 to 5.8, the X-axis represents the value of gap (ε) and re-balancing factor(a) and Y-axis represents the execution time in microseconds.

Table 5.2: Execution Time of Library Sort Algorithm in Microseconds Based on Gap Values

<table>
<thead>
<tr>
<th>Re-balancing</th>
<th>Value of ε</th>
<th>LUGD</th>
<th>LNGD</th>
<th>LUGD</th>
<th>LNGD</th>
<th>LUGD</th>
<th>LNGD</th>
<th>LUGD</th>
<th>LNGD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε = 1</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ε = 2</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ε = 3</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ε = 4</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

100
Figure 5.5: Re-balancing execution time comparison between LUGD and LNGD using random data

Figure 5.5 describes the plot at random data for the different values of ‘ε’ and re-balancing factor ‘a’. It is observed from Figure 5.14, as if we increase the re-balance factor in the case of LUGD the execution time also increases significantly, but in the case of LNGD the improvement of execution time achieved upto 94% in comparison to LUGD.

Figure 5.6: Re-balancing execution time comparison between LUGD and LNGD using nearly sorted data

Figure 5.6 shows the comparison of LUGD with LNGD at different gaps and
re-balancing factors. Again the improvement is upto 92% at \( a = 4 \) and \( \varepsilon = 4 \).

Figure 5.7: Re-balancing execution time comparison between LUGD and LNGD using reverse sorted data

Figure 5.7 shows the comparison of execution time of reverse sorted data at the different values of \( \varepsilon \) and re-balancing factor \( a \). Initially, in this case results are improved by 8% but maximum upto 57% at \( \varepsilon = 4 \) and \( a = 4 \).

Figure 5.8: Re-balancing execution time comparison between LUGD and LNGD using sorted data

Figure 5.8 represents the result on the sorted data with the different value of gaps and re-balancing factor. The result shows that maximum improvement
achieved is up to 91% in comparison to that of LUGD.

5.4 Conclusion

The final conclusion of this chapter is that, the proposed approach of LNGD proved to be a better algorithm in comparison to that of LUGD. We have achieved an improvement that ranges from 8% to 90%. The improvement of 90% has been found in the cases where the LUGD was performing poorer. We have also found that the performance of LNGD is better for different values of re-balancing factor which was not achieved in the case LUGD. The LNGD and LUGD both algorithms are implemented in C language.

In future, we will investigate the locality of data in more details. This will help not only in allocating the spaces accurately, but may also reduce the extra spaces which have been allocated and will act as an overhead both on the space and execution time of the program.