Chapter 4

Performance Enhancement of Library Sort Algorithm with Uniform Gap Distribution

Many authors have invented many sorting algorithms [41][62], among them insertion sort is the one of the simplest algorithm used for sorting [42]. Insertion sort [43] [75] is less efficient on large number of items as it takes $O(n^2)$ time in worst case [44] [76], and the best case of insertion sorting occurs when data is in sorted manner and it is $O(n)$ in best case. Insertion sort is stable sorting algorithm [45]. The improvement to the insertion sort algorithm was invented by D.L Shell and the modified version is called shell sort [46]. Shell sort [47] is more efficient for large items. Library sort is an adaptive sorting [49] and also stable sorting algorithm [50]. If we leave more space, the fewer elements we move on insertion. The author achieves the $O(\log n)$ insertions with high probability using the evenly distributed gap, and the algorithm runs $O(n \log n)$ with high probability. $O(n \log n)$ is better than $O(n^2)$. The idea of leaving gaps for insertions in a data structure is used by Itai, Konheim, and Rodeh [52]. This idea has found recent application in external memory and cache-oblivious algorithms in the packed memory structure of Bender, Demaine and Farach-Colton and later
4.1 Objective

Library sort has better run time than insertion sort, but the library sort also has some issues.
The first issue is the value of gap which is denoted by ‘\( \varepsilon \)’, the range of gap is given, but it is yet to be implemented to check that given range is satisfying the concept of library sort algorithm.
The second issue is re-balancing which accounts the cost and time of library sort algorithm.
The third issue is that, only a theoretical concept of library sort is given, but the concept is not implemented. So, to overcome these issues of library sort, in this section, we have implemented the concept of library sort and done the detailed experimental analysis of library sort algorithm, and measure the performance of library sort algorithm on a dataset.

4.2 Library Sort Algorithm

Library sort [48] is the formulation of insertion sort algorithm. The author has given the theoretical concept about library sort. Author has given the gaps after each insertion in the array and gaps denoted by the epsilon but he has not given the value of epsilon. He used the re-balancing concept and he re-balanced the array after inserting the \( 2^i \) elements in the array, whether re-balancing is necessary in the array, but it is also amounts cost and time. So we have to decide that is re-balancing required after inserting the \( 2^i \) elements in the array. These are the some questions of library sort, So in this paper, we are going to overcome the questions of library sort algorithm. The algorithm of library sort is as follows:

Algorithm of Library Sort: There are three steps of the algorithm.
1. Binary Search with blanks

2. Insertion

3. Re-balancing

1. **Binary Search with blanks:** In library sort we have to search a number and the best search for an array is found by binary search. The array ‘\( S \)’ is sorted but has gap. As in computer, gaps of memory will hold some value and this value is fixed to sentential value that is ‘-1’. Due to this reason we cannot directly use the binary search for sorting. So we have modified the binary search. After finding the mid, if it comes out to be ‘-1’ then we move linearly left and right until we get a non zero value. These values are named as \( m_1 \) and \( m_2 \). Based on these values we define new low, high and mid for the working. Another difference of the binary search presented below is that it not only searches the element in the list but also reports the correct position where we have to insert the number.

2. **Insertion:** As we know, library sort is also known by the name ‘gapped insertion sort’. If the value to be inserted is in the gap, then it is ok but if there is an element in that particular position, we have to shift the elements till we find the next gap.

---

**Algorithm 13 Library Sort: Insertion**

**INPUT:** Data to be sorted \( n \) and pass number \( i \)

**OUTPUT:** Sorted list but without gaps

```python
if i == 1 then
    i1 = i - 1
    c1 = 0
end if
S1 = \text{pow}(2,i)
if S1 > size then
    S1 = size
end if
for j = (\text{pow}(2,i1-1) - c1) to S1 do
    k = \text{search}(\text{pow}(2,i)+\text{pow}(2,i+1),a[j])
    if S[k] != -1 then
        manage\text{till}(k)
    end if
    S[k] = a[j]
end for
```

---

78
Algorithm 12 Library Sort: Binary-Search with Blanks

**INPUT:** Data to be sorted $n$ and Number to be searched $k$

**OUTPUT:** Position to enter the element $d$

```plaintext
while (low < high) do
    mid = (low + high)/2
    if (S[mid] = = -1) then
        $m1 = m2 = mid$
        if ($m1 == 0 and m2 >= high+1) then
            if ($k < S[m1]$) then
                low = high = $m1$
            else
                low = high = $m1+1$
            end if
        end if
        if ($m1 > 0 and m2 < high+1$) then
            if ($k <= S[m1]$) then
                if ($k == S[m]$) then
                    low = high = $m1$
                else
                    high = $m1-1$
                end if
            end if
            if ($k > S[m1]$ and $k < S[m2]$) then
                low = $m1 + 1$
                high = $m2 - 1$
            end if
        end if
        if ($m2 < high$) then
            low = $m2 + 1$
        else
            low = $m2$
        end if
    end if
end while
```

if ($m1 == 0 and m2 <= high$) then
    if ($k >= S[m2]$) then
        if ($m2 <= high$) then
            low = $m2 + 1$
        else
            low = $m2$
        end if
    end if
end if
else
    if ($S[mid] < k$) then
        low = mid + 1
    end if
end if
3. **Re-balancing**: Re-balancing is done after inserting $2^i$ elements. This increases the size of the array. The increase in the size of array will depend on $\varepsilon$ (number of spaces to be inserted). To do this process we will require an auxiliary array of same size so as to make a duplicate copy with a gap.

**Algorithm 14 Library Sort: Re-balancing**

**INPUT:** Sorted data but not uniformly gapped and re-balancing factor $\varepsilon$.

**OUTPUT:** Sorted list of $n$ items

```plaintext
while $i < n$ do
    if $S[j] \neq -1$ then
        $\text{reba}[i] = S[j]$
        $i++$
        $j++$
        $l++$
        for $k = 0$ to $\varepsilon$ do
            $\text{reba}[i] = -1$
            $i++$
        end for
    else
        $j++$
    end if
    for $k = 0$ to $i$ do
        $S[k] = \text{reba}[k]$
    end for
end while
```

4.3 **Execution time based testing of library sort algorithm**

We have tested the library sort algorithm on a standard dataset [T10I4D100K (.gz)] [31] by increasing the value of gap ($\varepsilon$). The dataset contains the 1010228 items. We have tested on four cases. Table 4.1 shows the execution time in microseconds of library sort algorithm using the standard dataset. By analyzing the Table 4.1, we can see that when we increase the gap value between the elements the execution time will decrease. By analyzing the Table 4.1, we can see that when we increase the gap value between the elements the execution time will decrease. The following figures show this effect. In all the figures $X$-
axis represents the increasing value of the gap and the $Y$-axis shows the time in microseconds.

Table 4.1: Execution Time of Library Sort Algorithm in Microseconds Based on Gap Values

<table>
<thead>
<tr>
<th>Epsilon</th>
<th>Random</th>
<th>Nearly Sorted</th>
<th>Reverse Sorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon = 1$</td>
<td>981267433</td>
<td>864558882</td>
<td>1450636163</td>
<td>861929937</td>
</tr>
<tr>
<td>$\varepsilon = 2$</td>
<td>729981576</td>
<td>620115904</td>
<td>1065247938</td>
<td>609647355</td>
</tr>
<tr>
<td>$\varepsilon = 3$</td>
<td>119727535</td>
<td>358670053</td>
<td>278810310</td>
<td>356489846</td>
</tr>
<tr>
<td>$\varepsilon = 4$</td>
<td>23003046</td>
<td>117188830</td>
<td>263693774</td>
<td>116590140</td>
</tr>
</tbody>
</table>

Figure 4.1: Execution time of random data using value of gaps

Figure 4.2: Execution time of nearly sorted data using value of gaps
We have plotted Figure 4.1 to 4.4 by using Table 4.1. By examining these figures, we can see that how the execution time is decreasing when the gap value between items is increasing. In Figure 4.1 to 4.4, we are representing the execution time in microseconds in all the four cases of dataset.

**The value of epsilon:** when we increase the value of epsilon, the execution time will decrease, but at some point, value of epsilon gets saturated point because we are allocating more gaps, but these gaps are more than are actually required for the operation, so it will only be an extra memory overhead because we need more memory to store the elements. So in this way the space complexity of the
algorithm increases linearly, when we increase the value of epsilon. The concept of space complexity will be explained in the next section with the help of graph.

4.4 Memory based Testing of Library Sort

Auxiliary space complexity of library sort is $O(n)$, but the space complexity is not only limited to auxiliary space. It is the total space taken by the program which includes the following.

1. Primary memory required to store input data ($M_{ip}$)
2. Secondary memory required to store input data ($M_{is}$)
3. Primary memory required to store output data ($M_{op}$)
4. Secondary memory required to store output data ($M_{os}$)
5. Memory required to hold the code ($M_c$)
6. Memory required to working space (temporary memory) variables + stack ($M_w$)

1) $M_{ip}$: For $M_{ip}$, we have to allocate memory of four bytes for each variable (element). As we are having total of 1010228 elements, so it will consume $1010228 \times 4 = 4040912$ bytes. Again, to input these items in an array we will have an index variable ‘a’ will of four bytes and 4 bytes for file pointer so it will be total of 4040912 bytes + 4 bytes of file pointer = 4040916 bytes, and 16 bytes are used for variable declared in the program so total space complexity taken by the $M_{ip}$ = 4040932 bytes.

2) $M_{is}$: We will get this input as storage file in secondary storage, but in file we store this data in a stream of bytes in character. For this, it will have slightly larger memory in comparison to primary memory.

3) $M_{op}$: As we get the result either in input variable or in temporary variable, it will not have requirement for storage on primary memory, but as we have to write this data on to secondary storage it will require file pointer of 4 bytes.
4) $M_{os}$: As we get the result in a temporary variable i.e. in Borland C++, the output stored in str file and the size of $M_{os}$ will be the size of str file and it will be same for all four cases of dataset, because we are using the 1010228 elements for all the cases of dataset.

5) $M_{c}$: To calculate this space, we have to find the size of .exe files created in windows for the discussed library sort program, as this program will be stored in main memory for their execution. The size of the .exe file depends on the sorting algorithms.

6) $M_{w}$: The space complexity of $M_{w}$ of an algorithm depends on the variable declared for the allocation. In our case we divided the memory in various parts each having its own variables for specific functions.

Library Module: It is having its own three variable consuming up $4 \times 3 = 12$ Bytes of memory. In this function we have two functions called insertion and re-balancing. The insertion module requires the maximum space $(1 + \varepsilon) \times n \times 4$ bytes of memory to store the sorted data and temporary data during the processing and 20 bytes for temporary variables. This module itself has two prime modules: search and managetill, this modules consume the 20 bytes and 12 bytes of memory. The re-balancing is also require extra space for adding the space in the array which will again equal to $(1 + \varepsilon) \times n \times 4$ bytes of memory and 24 bytes of memory required for temporary variables. So the total memory will be equal to $2 \times ((1 + \varepsilon) \times n \times 4 \times 4 + 12 + 4 + 20 + 20 + 4 + 12 + 4 + 24 + 4)$. The details of these values have been described in the Table 4.2

In Table 4.2, we have seen the total space complexity taken by the library sort using the dataset. From Table 4.2, we can see that there is no effect of re-balancing factor, but there is an effect of epsilon values. When we increase the gap value, the space taken by the program will also increase. We can see this effect with the help of graph shown in Figure 4.5. In Figure 4.5, the X-axis represents the value of epsilon and the Y-axis represents the memory occupied by the library sort algorithm in bytes. We can see that space complexity of the library sort algorithm increases linearly, when we increase the value of epsilon.
or gaps between the elements. It increases because we require more memory to store the elements and it is directly proportional to the value of epsilon. Due to this fact, the memory required is directly proportional to the value of epsilon, where epsilon is \((1 + \varepsilon)n\).

Table 4.2: Total Memory in Bytes of Library Sort with Increasing Value of Gaps and Re-balancing Factor

<table>
<thead>
<tr>
<th>Re-balancing</th>
<th>Value of (\varepsilon)</th>
<th>(M_{ip})</th>
<th>(M_{oa})</th>
<th>(M_{op})</th>
<th>(M_{oa})</th>
<th>(M_e)</th>
<th>(M_w)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>16163752</td>
<td>30151174</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>24245576</td>
<td>38232998</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>32327400</td>
<td>46314822</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>40409224</td>
<td>54396646</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>16163752</td>
<td>30151174</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>24245576</td>
<td>38232998</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>32327400</td>
<td>46314822</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>40409224</td>
<td>54396646</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>16163752</td>
<td>30151174</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>24245576</td>
<td>38232998</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>32327400</td>
<td>46314822</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4040932</td>
<td>4932283</td>
<td>4</td>
<td>4932283</td>
<td>81920</td>
<td>40409224</td>
<td>54396646</td>
</tr>
</tbody>
</table>

Figure 4.5: Memory occupied by library sort
4.5 Re-balancing based Testing of Library Sort

As the re-balancing is done after inserting $a^i$ elements, this increases the size of array. The size of array will depend on $\varepsilon$ (number of spaces to be inserted). To do this process we will require an auxiliary array of the same size so as to make a duplicate copy with gap. Whether re-balancing is necessary after $a^i$ elements, but it also amounts the cost and time of library sort algorithm and what will be the suitable value for ‘$a$’ is the question. We have calculated re-balancing till $a^4$ where $a = 2, 3, 4$ values with the value of gaps $\varepsilon = 1, 2, 3, 4$. We have found that when we increase the re-balancing factor ‘$a$’ from 2 to 4 then the execution time of library sort algorithm will also increase. We can see this effect with the help of Table 4.3 and graphs described in Figure 4.6 to Figure 4.9.

Table 4.3: Time taken by Library Sort Algorithm in Microseconds during Re-balancing

<table>
<thead>
<tr>
<th>Re-balancing</th>
<th>Value of $\varepsilon$</th>
<th>Random</th>
<th>NearlySorted</th>
<th>ReverseSorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>981267433</td>
<td>864558882</td>
<td>1450636163</td>
<td>861929937</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>729981576</td>
<td>620115904</td>
<td>1065247938</td>
<td>609647355</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>119727535</td>
<td>358670053</td>
<td>278810310</td>
<td>356489846</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>23003046</td>
<td>117188830</td>
<td>263693774</td>
<td>116590140</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2622591059</td>
<td>2214715182</td>
<td>2832112301</td>
<td>3011802732</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2103580421</td>
<td>1964645906</td>
<td>2585747568</td>
<td>2651992181</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2043974421</td>
<td>1728175857</td>
<td>2195021514</td>
<td>1962122927</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1620914312</td>
<td>1600879365</td>
<td>2130261056</td>
<td>1620374625</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2942693856</td>
<td>2467933298</td>
<td>3239333534</td>
<td>3281368964</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2705332601</td>
<td>2510103530</td>
<td>3154811065</td>
<td>2923182920</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2676681610</td>
<td>2613423098</td>
<td>3013676930</td>
<td>2378347887</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2611656774</td>
<td>2157740458</td>
<td>2993363707</td>
<td>2222906193</td>
</tr>
</tbody>
</table>

From Table 4.3, we can see that execution time of library sort is increasing when the re-balancing factor will increase in all the cases of dataset. The following graph is showing this effect.
Figure 4.6: Re-balancing of library sort using random dataset

From Figure 4.6, we can see that execution time of library sort is increasing when the re-balancing factor is increasing using the random dataset.

Figure 4.7: Re-balancing of library sort using reverse sorted dataset

From Figure 4.7, we can see that execution time of library sort is increasing when the re-balancing factor is increasing using the nearly sorted dataset.
From Figure 4.8, we can see that execution time of library sort is increasing, when the re-balancing factor is increasing using the reverse sorted dataset.

From Figure 4.9, we can see that execution time of library sort is increasing when the re-balancing factor is increasing using the sorted dataset.

From Figure 4.6 to Figure 4.9, X-axis represents the value of epsilon and Y-
axis represents the execution time in microseconds when the re-balancing factor value is $2^i, 3^i, 4^i$. By analyzing the figures, we can see that the nature of data has marginally effected on the re-balancing factor. If the re-balancing factor is $2^i$ i.e we have to re-balanced the elements in the following manner $2^0, 2^1, ...2^n$. Then, the performance of algorithm is good because in the array, proper space is there to insert the new elements. But the performance of algorithm is degraded if the re-balancing factor increases from $2^i$ to $4^i$ because if we use the re-balancing factor $3^i$ i.e we have to re-balance the elements in the following manner $3^0, 3^1, ...3^n$. Then, in the array there is no proper space to insert the new elements in the manner of $3^i$ and $4^i$. So shifting of data is required to insert the new elements and the spaces between many elements have been already consumed so in this way performance degrades have a larger number of swapping to generate the spaces which is same as that in the case of traditional insertion sort.

4.6 Conclusion

By execution time analysis, we have found that as we increase the value of epsilon then the execution time will decrease but at some point the value of epsilon get to a saturated point because we will have the extra spaces for the data to be inserted in between.

By space complexity analysis, we have found that space complexity of the library sort algorithm increases linearly. That is, when we increase the value of epsilon the memory consumption is also increases in the same proportion.

By execution time analysis of re-balancing, we have found that when we increase the re-balancing factor ‘$a$’ from 2 to 4 then the execution time of library sort algorithm will also increase as it moves towards traditional insertion sort. So, to find out the better result of library sort algorithm, the value of epsilon should be optimal and re-balancing factor should be minimum or ideally equal to 2.