Extension of Merge Sort Algorithm for the Union, Intersection and Ringsum of Two Sorted Sets.

Himank Goel iit2016061@iiita.ac.in

Siddhant Gautam iit2016069@iiita.ac.in

Akshat iit2016094@iiita.ac.in

Ankit Shrey iit2016050@iiita.ac.in

***Abstract*—In this paper we focus on finding the union, intersection and ringsum of two sorted sets. To achieve our objective, we have extended the existing merge sort algorithm. Further we have analyzed the various time complexities for each of the aforementioned cases by plotting graphs of Time vs Input Size of Problem.**

I. Introduction and literature survey

This document is our final report for the Assignment 1 in the course of Design and Analysis of Algorithm. The report describes an algorithm to find the union, intersection and ringsum of two sorted sets.

We have implemented the said algorithm in C and have used GNU Plot to plot the time complexity analysis graphs and other relevant analysis related to our algorithm. For implementing the algorithm we have worked on the basic mathematical definitions of Union, Intersection and Ringsum of two sets.

II. Algorithm Design

A. Union of sets

The union of two sets A and B is defined as the set of elements which are in A, in B, or in both A and B.

In symbols, $A \cup B = \{x : x \in A \text{ or } x \in B\}$

Using the above stated definition we devised the following algorithm. The algorithm takes in two sorted arrays as input along with their sizes.

We initialize a third array to store the final output.

We take two pointers to iterate over the two arrays. The pointers are initialized to zero if the arrays are in ascending order else they are initialized to the end of the respective arrays. This is done to ensure the the traversal of the complete array in ascending order.

A third pointer is also initialized to mark the position of elements in the the output array.

Since the arrays are sorted we compare the corresponding elements of both the arrays. The smaller of the two elements is added to the output array and the pointer to the source and destination array is incremented. In case, the elements are equal, the element is added to the output array and all the three pointers are incremented.

If either of the arrays is exhausted, the loop exits and the remaining elements of the other array are copied onto the output array.

Algorithm 1 Union of two sorted sets

Input: n , m , arr1[] , arr2[]
Output: Union of arr1[] , arr2[] in arr3[].

Initialisation :
$k \leftarrow 0$
if $(n = 1)$ OR $(array1[0] < array1[1])$ **then**
  $i \leftarrow 0$
  $increment_1 \leftarrow 1$
else
  $i \leftarrow n - 1$
  $increment_1 \leftarrow -1$
end if
if $(m = 1)$ OR $(array2[0] < array2[1])$ **then**
  $j \leftarrow 0$
  $increment_2 \leftarrow 1$
else
  $j \leftarrow m - 1$
  $increment_2 \leftarrow -1$
end if
while $(i < n)$ AND $(j < m)$ AND $(i >= 0)$ AND $(j >= 0)$ **do**
  if $(arr1[i] < arr2[j])$ **then**
    $arr3[k] = arr1[i]$
    $i \leftarrow increment_1 + 1$
  else if $(arr1[i] > arr2[j])$ **then**
    $arr3[k] = arr2[j]$
    $j \leftarrow increment_2 + 1$
  else
    $arr3[k] = arr2[j]$
    $i \leftarrow increment_1 + 1$
    $j \leftarrow increment_2 + 1$
  end if
  $k \leftarrow k + 1$
end while
while $(i < n)$ AND $(i >= 0)$ **do**
  $arr3[k] = arr1[i]$
  $i \leftarrow increment_1 + 1$
  $k \leftarrow k + 1$
end while
while $(j < n)$ AND $(j >= 0)$ **do**
  $arr3[k] = arr2[j]$
  $j \leftarrow increment_2 + 1$
  $k \leftarrow k + 1$
end while

B. Intersection of sets

The intersection of two sets A and B is defined as the set of elements which are in both A and B.

In symbols, $A \cap B = \{x : x \in A \text{ and } x \in B\}$

Using the above stated definition we devised the following algorithm. The algorithm takes in two sorted arrays as input along with their sizes.

We initialize a third array to store the final output.

We take two pointers to iterate over the two arrays. The pointers are initialized to zero if the arrays are in ascending order else they are initialized to the end of the respective arrays. This is done to ensure the the traversal of the complete array in ascending order.

A third pointer is also initialized to mark the position of elements in the the output array.

We run a while loop until either of the pointers traverses the complete array. Since the arrays are sorted we compare the corresponding elements of both the arrays. In case, the elements are equal, the element is added to the output array and all the three pointers are incremented.

In all the other cases, the pointer corresponding to the array which has smaller of the two elements is incremented.

Algorithm 2 Intersection of two sorted sets

Input: n , m , array1[] , array2[]
Output: Intersection of of array1[] , array2[]

Initialisation :
$k \leftarrow 0$
if $(n = 1)$ OR $(array1[0] < array1[1])$ **then**
  $i \leftarrow 0$
  $increment_1 \leftarrow 1$
else
  $i \leftarrow n - 1$
  $increment_1 \leftarrow -1$
end if
if $(m = 1)$ OR $(array2[0] < array2[1])$ **then**
  $j \leftarrow 0$
  $increment_2 \leftarrow 1$
else
  $j \leftarrow m - 1$
  $increment_2 \leftarrow -1$
end if
while $(i < n)$ AND $(j < m)$ AND $(i >= 0)$ AND $(j >= 0)$ **do**
  if $(arr1[i] < arr2[j])$ **then**
    $i \leftarrow i + increment_1$
  else if $(arr1[i] > arr2[j])$ **then**
    $j \leftarrow j + increment_2$
  else
    $arr3[k] = arr1[i]$
    $i \leftarrow i + increment_1$
    $j \leftarrow j + increment_2$
    $k \leftarrow k + 1$
  end if
end while

C. Ringsum of sets

The ringsum or symmetric difference of two sets is equivalent to the union of both relative complements of the two sets.

In symbols, $A \triangle B = (A \setminus B) \cup (B \setminus A)$

Using the above stated definition we devised the following algorithm. The algorithm takes in two sorted arrays as input along with their sizes. We initialize a third array to store the final output.

We take two pointers to iterate over the two arrays. The pointers are initialized to zero if the arrays are in ascending order else they are initialized to the end of the respective arrays. This is done to ensure the the traversal of the complete array in ascending order. A third pointer is also initialized to mark the position of elements in the the output array.

We run a while loop until either of the pointers traverses the complete array. Since the arrays are sorted we compare the corresponding elements of both the arrays.If either of the element is smaller, the element is added to the output array and the pointer of the source and output array is incremented. If however, the elements are equal the element is skipped and pointer of the original arrays are incremented.

If either of the arrays is exhausted, the loop exits and the remaining elements of the other array are copied onto the output array.

Algorithm 3 Ringsum of two sorted sets

Input: n , m , arr1[] , arr2[]

Initialisation :
$k \leftarrow 0$
if $(n = 1)$ OR $(array1[0] < array1[1])$ **then**
  $i \leftarrow 0$
  $increment_1 \leftarrow 1$
else
  $i \leftarrow n - 1$
  $increment_1 \leftarrow -1$
end if
if $(m = 1)$ OR $(array2[0] < array2[1])$ **then**
  $j \leftarrow 0$
  $increment_2 \leftarrow 1$
else
  $j \leftarrow m - 1$
  $increment_2 \leftarrow -1$
end if
while $(i < n)$ AND $(j < m)$ AND $(i >= 0)$ AND $(j >= 0)$ **do**
  if $(arr1[i] < arr2[j])$ **then**
    $arr3[k] = arr1[i]$
    $i \leftarrow increment_1 + 1$
    $k \leftarrow k + 1$
  else if $(arr1[i] > arr2[j])$ **then**
    $arr3[k] = arr2[j]$
    $j \leftarrow increment_2 + 1$
    $k \leftarrow k + 1$
  else

```
        i ← increment_1 + 1
        j ← increment_2 + 1
    end if
end while
while (i < n) AND (i >= 0) do
    arr3[k] = arr1[i]
    i ← increment_1 + 1
    k ← k + 1
end while
while (j < n) AND (j >= 0) do
    arr3[k] = arr2[j]
    j ← increment_2 + 1
    k ← k + 1
end while
```

III. Analysis and Discussions

A. Union

For the union of two sets according to the algorithm presented above the the following analysis takes place.

1) Best Case Analysis: The best case scenario for the algorithm would be to spend time in the minimum number of comparisons. For this, for any input size (m+n), where m is the size of first array and n the size of second. Given m be any number, the value of n must be one, in this scenario the least number of comparisons would take place and the algorithm would exit at the earliest.
Therefore the Best complexity comes out to be

$$f(n) = 18 + 11 + 4 + 5 + 7(m-1) + 4$$
$$m < 7m + 28 < 35m$$
$$g(m) < f(m) < h(m)$$

Here $g(m)$ is the tightest lower bound and $h(m)$ is the lowest upper bound. Therefore, best case complexity is $\Omega(m)$

2) Worst Case Analysis: The worst case scenario for the algorithm would be to spend time in the maximum number of comparisons. For this, for any input size (m+n), where m is the size of first array and n the size of second. The sizes, m and n should be either equal or at max differ by one. Also the elements should be alternate in accordance with their value.
For eg. Array1 = $\{1, 10, 15, 21\}$
Array2 = $\{7, 11, 18, 25, 35\}$
Therefore the Big-Oh complexity comes out to be

$$Case1 : m = n$$
$$f(n,m) = 18 + 7m + 10(n-1) + 5(n+m-1) + 4 + 2 + 9$$
$$f(n,m) = 12m + 15n + 18$$
$$f(n) = 27n + 18$$

$$Case2 : m = n + 1$$
$$f(n,m) = 18 + 7(m-1) + 10(n) + 5(n+m-1) + 4 + 2 + 9$$
$$f(n,m) = 12m + 15n + 21$$
$$f(n) = 27n + 33$$

For m = n + 1, the final output $f(n)$ differs from the $f(n)$ for m = n. This is due to the difference in flow of code. But ignoring the minor difference in constants

$$n < f(n) < 61n$$
$$g(n) < f(n) < h(n)$$

Here g(n) is the tightest lower bound and h(n) is the lowest upper bound. Therefore, worst case complexity is $O(n)$

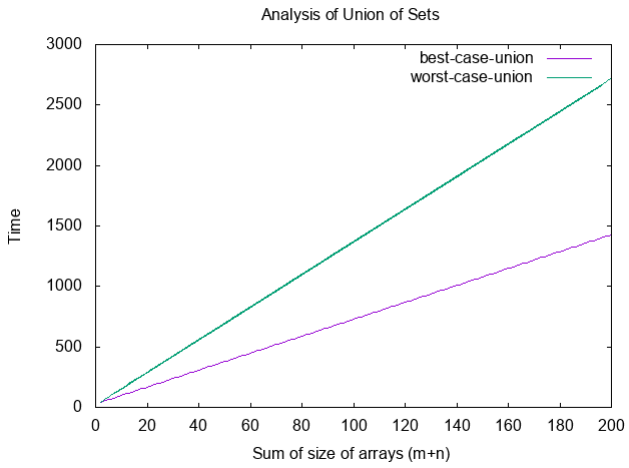

Fig. 1. Analysis for union for sets

B. Intersection

For the intersection of two sets according to the algorithm presented above the the following analysis takes place.

1) Best Case Analysis: The best case scenario for the algorithm would be to spend time in the minimum number of comparisons. For this, for any input size (m+n), where m is the size of first array and n the size of second. Given m be any number the value of n must be one, in this scenario the least number of comparisons would take place and the algorithm would exit at the earliest.
Therefore the Best complexity comes out to be

$$f(n) = 18 + 12 + 4 + 4 = 38$$

Thus , the best case complexity for intersection comes out to be constant i.e. $O(1)$.

2) Worst Case Analysis: The worst case scenario for the algorithm would be to spend time in the maximum number of comparisons. For this, for any input size (m+n), where m is the size of first array and n the size of second. The sizes, m and n should be either equal or at max differ by one. Also the elements should be alternate in accordance with their value.
For eg. Array1 = $\{2, 11, 16, 22\}$
Array2 = $\{8, 12, 19, 26, 36\}$

Therefore the Big-Oh complexity comes out to be

$$Case1 : m = n$$
$$f(n,m) = 18 + 4m + 7(n-1) + 4(m+n-1)$$
$$f(n,m) = 11n + 8m + 7$$
$$f(n) = 19n + 7$$

$$Case2 : m = n + 1$$
$$f(n,m) = 18 + 4m + 7(n-1) + 4(m+n-1)$$
$$f(n,m) = 18 + 4m + 7(n-1) + 4(m+n-1)$$
$$f(n) = 19n + 15$$

For m = n + 1, the final output $f(n)$ differs from the $f(n)$ for m = n. This is due to the difference in flow of code. But ignoring the minor difference in constants

$$n < f(n) < 35n$$
$$g(n) < f(n) < h(n)$$

Here g(n) is the tightest lower bound and h(n) is the lowest upper bound. Therefore, best case complexity is $O(n)$ Here g(n) is the tightest lower bound and h(n) is the lowest upper bound.

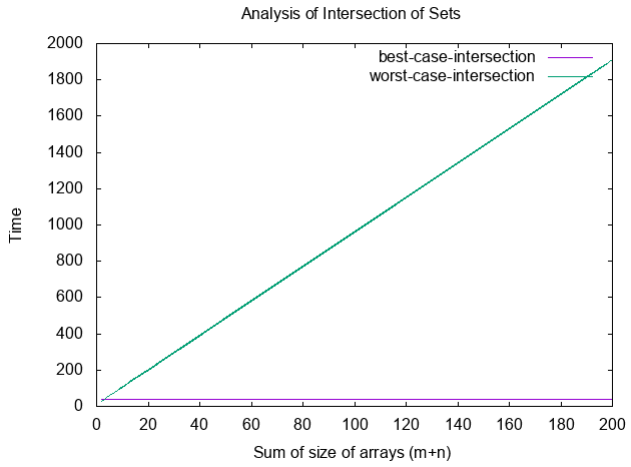

Fig. 2. Analysis for intersection of sets

C. Ringsum

For the ringsum of two sets according to the algorithm presented above the the following analysis takes place.

1) Best Case Analysis: The best case scenario for the algorithm would be to spend time in the minimum number of comparisons. For this, for any input size (m+n), where m is the size of first array and n the size of second. Given m be any number the value of n must be one, in this scenario the least number of comparisons would take place and the algorithm would exit at the earliest.

Therefore the Best complexity comes out to be

$$f(n) = 18 + 8 + 5 + 4 + 7(m-1) + 4$$
$$m < 7m + 32 < 39m$$
$$g(m) < f(m) < h(m)$$

Here $g(m)$ is the tightest lower bound and $h(m)$ is the lowest upper bound. Therefore, best case complexity is $\Omega(m)$

2) Worst Case Analysis: The worst case scenario for the algorithm would be to spend time in the maximum number of comparisons. For this, for any input size (m+n), where m is the size of first array and n the size of second. The sizes, m and n should be either equal or at max differ by one. Also the elements should be alternate in accordance with their value.
For eg. Array1 = $\{72, 35, 25, 11, 5\}$
Array2 = $\{70, 31, 18, 10, 2\}$
Therefore the Big-Oh complexity comes out to be

$$Case1 : m = n$$
$$f(n,m) = 18 + 8m + 11(n-1) + 5(n+m-1) + 4 + 2 + 9$$
$$f(n,m) = 16n + 13m + 17$$
$$f(n) = 29n + 17$$

$$Case2 : m = n + 1$$
$$f(n,m) = 18 + 8(m-1) + 11(n) + 5(n+m-1) + 4 + 2 + 9$$
$$f(n,m) = 16n + 13m + 20$$
$$f(n) = 29n + 33$$

For m = n + 1, the final output $f(n)$ differs from the $f(n)$ for m = n. This is due to the difference in flow of code. But ignoring the minor difference in constants

$$n < f(n) < 63n$$
$$g(n) < f(n) < h(n)$$

Here g(n) is the tightest lower bound and h(n) is the lowest upper bound. Therefore, worst case complexity is $O(n)$

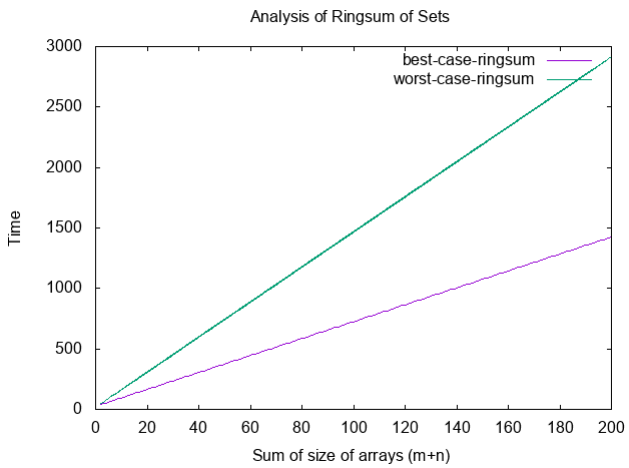

Fig. 3. Analysis for Ringsum of sets

IV. Experimental Study

For testing the above algorithms we plotted the time vs input size for 10,000 cases ranging from (m+n) = 2 to 200. For each value of (m+n) we chose all possible integral combinations for the value of m and n.
The numbers in the arrays were randomly generated using the $rand()$ function in C. Also we randomly arrange the generated arrays in ascending or descending order.
After this, we divided the input size into groups of 12 to analyze the T_{min}, T_{max}, and T_{avg} for the algorithm within the defined set space.

The tables and corresponding graphs generated are shown in below.

Union of Sets

$Size$	T_{max}	T_{min}	T_{avg}
6	157	45	105.30
30	459	207	318.17
54	776	375	550.50
78	1084	543	780.63
102	1382	700	1018.49
126	1672	1088	1371.50
150	1956	1491	1735.54
174	2311	1894	2102.29
198	2493	2324	2418.44

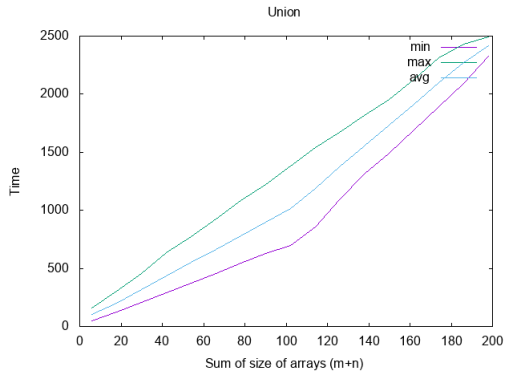

Fig. 4. Union of Sets Analysis

Intersection of Sets

$Size$	T_{max}	T_{min}	T_{avg}
6	109	33	62.70
30	334	33	161.80
54	558	33	272.82
78	778	33	383.06
102	994	33	508.97
126	1225	396	827.06
150	1443	838	1156.94
174	1659	1276	1490.48
198	1863	1712	1783.48

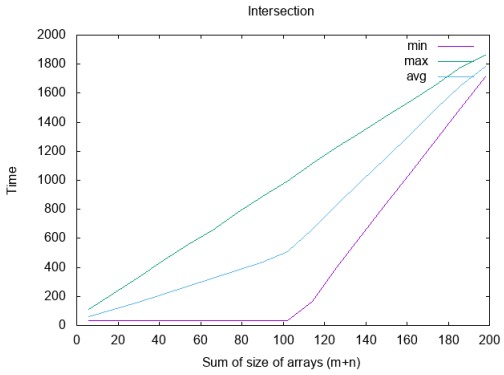

Fig. 5. Intersection of Sets Analysis

Ringsum of Sets

$Size$	T_{max}	T_{min}	T_{avg}
6	157	46	108.63
30	464	209	324.45
54	827	368	565.14
78	1106	536	808.28
102	1446	705	1045.83
126	1775	1069	1410.80
150	2032	1495	1790.80
174	2336	1993	2165.40
198	2632	2423	2513.20

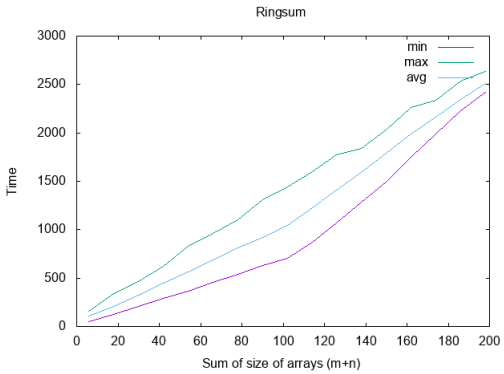

Fig. 6. Ringsum of Sets Analysis

V. Conclusions

In this paper, we proposed an algorithm to find the union, intersection and ringsum of two sorted sets in linear time and space complexities. We used two pointer (merge algorithm) technique to perform these operations. Our experiment indicates that the average time required to perform these operations is sum of the sizes of the two sets. Also, the time complexity of our algorithm is $\theta(n)$.