Utilize UNION operation on given two sets of positive integers to output the sorted array

BISWAJEET DAS(IIT2019019), SHREYANSH PATIDAR(IIT2019018), HRITIK SHARMA(IIT2019020)
4th Semester , Department of Information Technology
Indian Institute of Information Technology , Allahabad , India

---

Abstract— This Paper contains the algorithm to Utilize UNION operation on given two sets of positive integers to output the sorted array. Three approaches have been taken and we will see the difference in complexity between both.

I. INTRODUCTION

Union: the union (denoted by ∪) of a collection of sets is the set of all elements in the collection.
Sorting is the process of arranging the elements of a set in a fashionable order i.e either in ascending or descending order of the elements of the set.
This report further contains -
II. Algorithm Design
III. Algorithm Analysis
IV. Result
V. Conclusion
VI. Result

II. ALGORITHM DESIGN

Steps for designing this algorithm are -

Approach 1:

1. Find the maximum number among both the sets (maxALL) by individually getting the max of each set
(maxA: maximum positive integer of set A and maxB: maximum positive integer of set B) and maxALL = max(maxA, maxB).
2. Iterate over a loop from 1 to maxALL (both inclusive).
3. We check whether i is present in both the sets(A and B) or not.
4. If i is present in both sets A,B then we print i.
5. Answer is printed according to the result of the algorithm.

Algorithm 1:

```
int maxa=0,maxb=0;
for(i: 1→sizeof(A) ) maxa=max(maxa,A[i]);
for(i: 1→sizeof(B) ) maxb=max(maxb,B[i]);
maxALL = max(maxA,maxB);

for(i: 1→maxALL)
{
        if(i in A and i in B)
              Insert i in ans_array;
}

print(ans_array);
```

Approach 2:

1. Make a new array ans_array.
2. Traverse over array A and check for each element x is present in ans_array or not. If its present then continue, else insert it in ans_array.
3. Insertion here will be based on binary search technique(divide and conquer). We take mid element of ans_array and check whether its greater or smaller than x. If greater,then lower r to be mid-1, else increase l to be mid+1. If we get a position mid such that there exist no element greater smaller than x then we insert x in ans_array at position mid+1.
4. We do the same thing for array B.
5. Finally we are left with the ans in ans_array

Algorithm 2:

```
for(i : 1→n)
{
        int l =1, r= n, pos=-1;
        while(l<=r)
        {
         int mid = (l+r)/2;
          if(A[mid]==x)break;
          else if(A[mid]<x)pos=mid,l=mid+1;
          else r = mid-1;
        }
        Insert x in pos if pos!=-1;
}
Same in B[].
```

Approach 3:

1. Sort both the arrays A and B individually.
2. Make a new array ans_array.
3. Take 2 pointers(pa,pb) one for each array, pointing at the first element of A and B respectively.
4. Check if A[pa]>B[pb] then check if B[pb] is present in ans_array or not. If not then insert B[pb] into ans_array. Increment pb to pb+1
5. If A[pa]<B[pb] then check if A[pa] is present in ans_array or not. If not then insert A[pa] into ans_array. Increment pa to pa+1
6. If A[pa]=B[pb] then check if A[pa] is present in ans_array or not. If not then insert A[pa] into ans_array. Increment pa to pa+1 and pb to pb+1
7. Finally print ans_array.

Algorithm 3:

```
int p1=0,p2=0;
while(p1<n  and   p2<m)
{
        if(A[p1]<B[p2])
        {
          if(Search in ans_arr(A[p1]))
            continue;
          else
            Insert A[p1] in ans_array
           p1++;
        }
if(A[p1]>B[p2])
        {
          if(Search in ans_arr(B[p2]))
            continue;
          else
            Insert B[p2] in ans_array
           p2++;
        }
        if(A[p1]==B[p2])
        {
          if(Search in ans_arr(A[p1]))
            continue;
          else
            Insert A[p1] in ans_array
           p1++; p2++;
        }
}
```

III. ALGORITHM ANALYSIS

Approach 1:

Here the maximum value of maxALL can go to maximum value of integer value which is 2^{32} - 1.

And each searching operation takes linear time i.e. ∝ n and m.

So the time complexity will be O((maxALL) * (n+m)).

$\mathbf{t_{best}}$ **:** when n=m=0, = O(0) = 0ms

$\mathbf{t_{worst}}$ **:** when n=m=1000(max limit given) and maxALL = INT_MAX. = O(10^{15})= 10^{10} ms.

Approach 2:

Here, traversing over each array with size n,m will take time ∝ n and m respectively. Finding mid pos for each element will take log(n+m) time. And insertion operation will take time ∝ n + m.

So, the time complexity will be

O((n+m)*log(n+m)*(n+m)).

$\mathbf{t_{best}}$ **:** when n=m=0, = O(0) = 0ms

t_{worst} : when n=m=1000(max limit given) = $O(4*10^6 * \log(2000))$ = 1.2 secs.

Approach 3:

Here, traversing over each array with size n,m will take time ∝ n and m. And searching the ans_array with binary search will take log(n+m) time. Insertion takes constant time.

So, the time complexity will be

$O((n+m)* \log(n+m))$

t_{best} : when n=m=0, = O(0) = 0ms

t_{worst} : when n=m=1000(max limit given) = $O(2000 * \log(2000))$ = 0.7ms.

The table below shows about how the value changes for a particular value of (N+M) for all the 3 algorithms.

SNo.	N+M	maxALL	Algo1	Algo 2	Algo 3
1	0	0	0	0	0
2	10	1000	10000	230.2585093	23.02585093
3	20	1000	20000	1198.292909	59.91464547
4	30	1000	30000	3061.077643	102.0359214
5	40	1000	40000	5902.207127	147.5551782
6	50	1000	50000	9780.057514	195.6011503

Table 1: values of O(n) for all algorithms

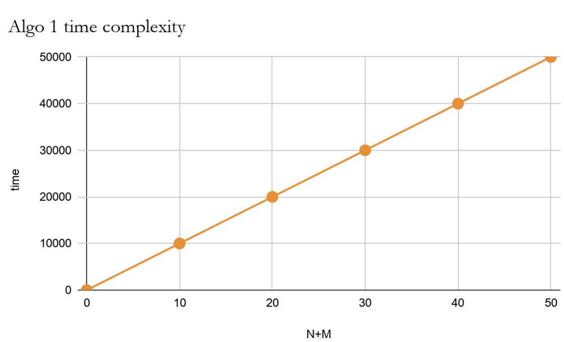

Fig 1: time complexity of algorithm 1 illustrated

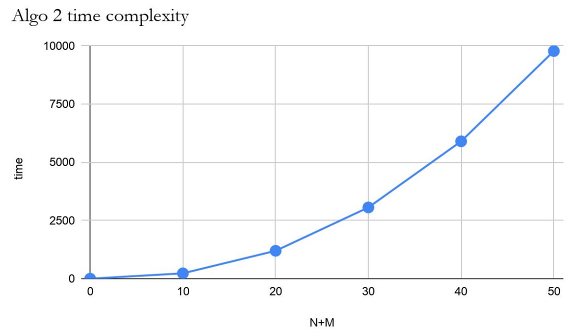

Fig 2: time complexity of algorithm 2 illustrated

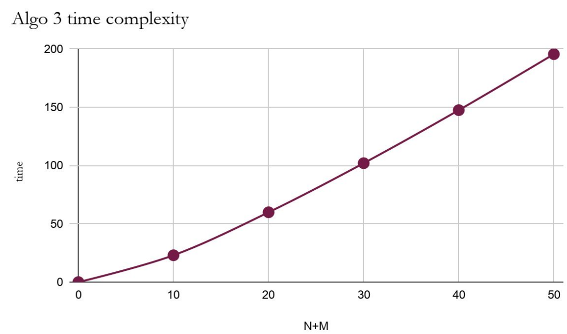

Fig 3: time complexity of algorithm 3 illustrated

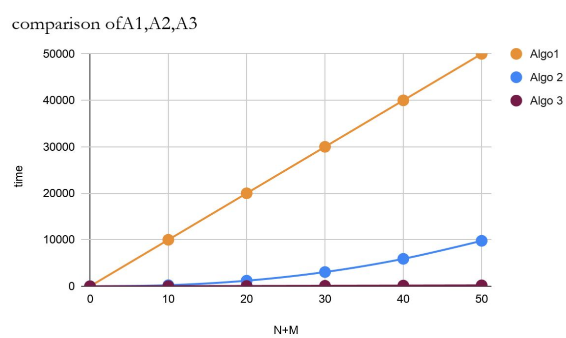

Fig 4: comparison of all algorithms, illustrated

IV. CONCLUSION

All the three methods have different time complexities and meet to fulfill the problem statement. The order in which they are good can be listed as:

I. Approach 3
II. Approach 2
III. Approach 1

Based on the time complexities.

V. REFERENCES

https://en.wikipedia.org/wiki/Union_(set_theory)