23

Department of Information & Computing Sciences

University of Oklahoma

ALGORITHMS ON SETS AND RELATED PROBLEMS

Teofilo Gonzalez

Technical Report No. 75-15

December 1975

"Algorithms on Sets and Related Problems"

Teofilo Gonzalez
Information and Computing Sciences
University of Oklahoma

Abstract

A new structure for representing sets is presented. It is shown that algorithms which make use of this structure will solve in linear time the following problems: set equality, dictionaries, disjoint sets, element uniqueness, mode, IRS, min gap, etc. For all of these problems a lower bound on the time complexity of $O_c(n\log n)$ has been obtained. The storage requirements for these algorithms, even though bounded, might be extremely large compared to the number of inputs. We present an algorithm for the max-d gap problem which only requires linear time and space, however a similar problem, the min-d gap can only be shown to be as hard as some of these set problems.

Key Words: computational complexity, linear time algorithms, searching, dictionaries, element uniqueness, min gap, max gap, sets, set operations, mode, IRS.

I. Introduction

The problems we study in this paper are: mode, dictionaries, IRS, element uniqueness, time-space, max gap, min gap, set equality, union, intersection, symmetric difference, etc. For most of these problems a lower bound on the time complexity of $O_c(n\log n)$ has been obtained (see Reingold [6], Dobkin and Lipton [3] and Shamos and Hoey [7]).

For the subset problem ($A \subseteq B$), Knuth [4, p. 391] suggests three approaches which yield algorithms of time complexity $O(nm)$, $O((n+m)\log n)$ and $O(n+m)$, where $n=|A|$, $m=|B|$ and $n \leq m$. The linear time solution requires a "suitable" hashing function and the time required to find this function is not included in the analysis. For any given hash function the time complexity of this algorithm would in the worst case be of $O(nm)$, see Aho, Hopcroft and Ullman [1, p. 163].

We present a linear time algorithm for the subset problem. The space requirements, even though bounded, might be extremely large compared to the length of the sets. Let us now define memory and space complexity. The specific memory locations which the algorithm modifies during its execution, expressed as a function of the size of the problem is called the memory complexity of the algorithm. The total memory required (highest location used - lowest location used) by the algorithm, expressed as a function of the size of the problem is the space complexity

of the algorithm. Similarly as in [1] we can define asymptotic memory and space complexity. The algorithm we present for this problem has asymptotic time complexity of $O(n+m)$, memory complexity is $O(n+m)$ and the space complexity is bounded by the difference between the largest and smallest integer used to represent the elements of the set.

Any algorithm for a machine with sequential access memory device (e.g., Turing machines) has the property that the time complexity $\geq$ space complexity = memory complexity. For random access machines, it is not known whether this relation between time and space is always true. In other words, take any algorithm with $T_1(n)$, $M_1(n)$ and $S_1(n)$, where $T_1(n)$, $M_1(n)$ and $S_1(n)$ are functions representing the asymptotic time, memory and space complexity, such that $O(M_1(n)) \leq O(T_1(n)) < O(S_1(n))$. Is there an algorithm (to solve the same problem) with $T_2(n) = O(T_1(n))$, $M_2(n) = O(M_1(n))$ and $S_2(n) = O(M_1(n))$? In section II we show that this problem (time - space), is equivalent to the on line dictionary.

Throughout this paper we make the following assumptions:

i) elements in sets are unordered,

ii) sets are stored without repetitions,

and iii) groups of numbers are unordered.

We now define formally the problems we are going to study.

1. element uniqueness

input: elements $x_1, x_2, \ldots, x_n$ (x_i integers)

property: all elements are different

2. set equality

input: sets A and B (elements, integers)

property: A=B or A+B = $\emptyset$

3. subset

input: sets A and B (elements, integers)

property: $A \subseteq B$ or $A - B = \emptyset$

4. disjoint sets

input: sets A and B (elements, integers)

property: $A \cap B = \emptyset$ or $|A \cup B| = |A| + |B|$

5. mode

input: integer ℓ and elements $x_1, x_2, \ldots, x_n$ (x_i's integers)

property: there is an element repeated ℓ times.

6. multiset representation

input: integer k and elements $x_1, x_2, \ldots, x_n$ (x_i's integers)

property: there is a representation $\langle y_1, i_1 \rangle, \langle y_2, i_2 \rangle, \ldots, \langle y_k, i_k \rangle$ such that all y_i's are different, $\sum_{j=1}^{k} i_j = n$ and $\langle y_j, i_j \rangle \Rightarrow$ (there are i_j elements in X equal to y_j) for $j = 1, 2, \ldots k$.

7. IRS

input: employer forms $\langle x_i, y_i, z_i \rangle$ (employer x_i paid employee y_i, z_i dollars) and employee forms $\langle u_i, v_i, w_i \rangle$ (employee u_i, received from employer v_i, w_i dollars). (all integers)

property: there is a perfect match between employee reports and employer reports.

8. dictionaries

input: operations of the form insert x , search x and delete x . (x integer)

property: answer all search queries.

9. time - space

input: algorithm $(T_1(n), M_1(n), S_1(n))$

output: there is an algorithm (for the same problem) such that $T_2(n) = O(T_1(n))$, $M_2(n) = O(M_1(n))$ and $S_2(n) = O(M_1(n))$.

10. min gap

input: numbers $d, x_1, x_2, \ldots, x_n$ (x_i's and d integers)

property: there are two numbers x_i and x_j such that $|x_i - x_j| \leq d$ and either $x_k \leq \min(x_i, x_j)$ or $x_k \geq \max(x_i, x_j)$ for $k=1,2,\ldots,n$.

11. max gap

input: numbers $d, x_1, x_2, \ldots, x_n$ (d & x_i's reals)

property: there are two numbers x_i and x_j such that $|x_i - x_j| \geq d$ and either $x_k \leq \min(x_i, x_j)$ or $x_k \geq \max(x_i, x_j)$ for $k=1,2,\ldots,n$.

It can be easily shown that some of these problems are just instances of more general problems, for example: set equality of symmetric difference; disjoint set of set union of set intersection and subset of relative complement. Some of these problems are better known as optimization problems: max mode is just the maximum ℓ for which the mode has a solution and in min-d gap (max-d gap) we find the minimal (maximal) d such that min gap (max gap) has a solution.

In section II we present a new structure for representing sets. Then we will show several new reductions between these problems. Finally we present an algorithm which will execute any n instructions for dictionaries in O(n) time, and show how to solve problems 1 to 7 and 10 in linear time. In section III we present a linear time and space algorithm for the max-d gap problem.

II. Set Operations and Related Problems

Sets are usually represented by a list. This structure requires O(n) space and the set up time is also O(n), where n is the length of the set. Another structure is the bit or characteristic vector. The space requirement is O(m) and the set up time is O(n+m), where m is the length of the universe of discourse.

The structure we propose is a combination of the list and characteristic vector representations. The sequential list contains the elements in the set and the characteristic vector contains the indexes of the elements in the list. Note that the index vector need not be initialized and it may contain any information. In order to identify elements in the set from elements not in the set, we make use of a 1-1 correspondence between the list and the index vector. The space required is O(n+m). The initialization time is O(n). Figure 1 is an example of this representation for set A={5,3,9,2}

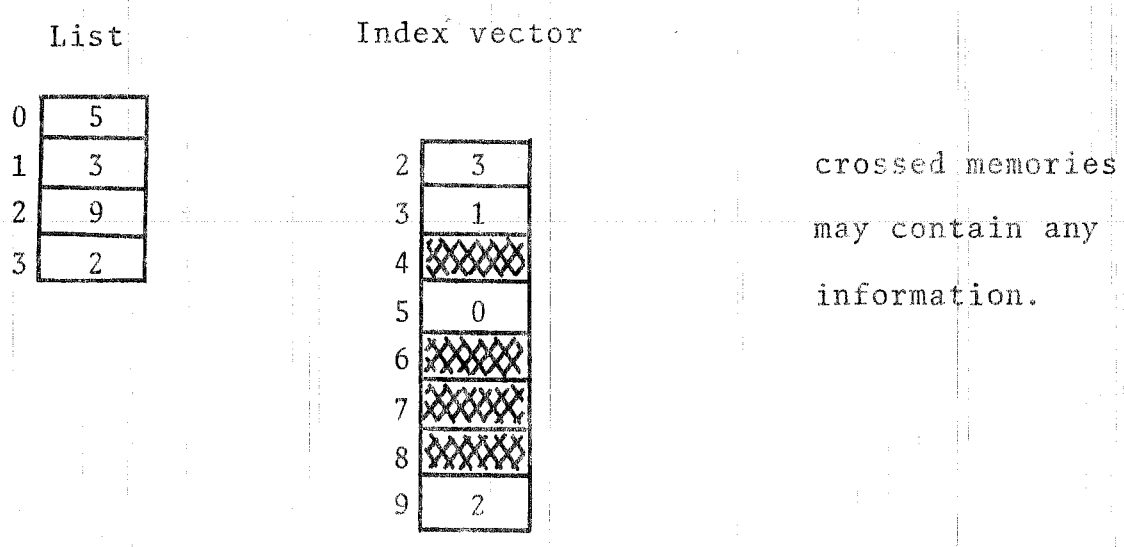

Fig. 1 Representation of set A={5,3,9,2}

We can easily see that this structure can be used to store sparse tables, matrices, trees, graphs, etc. With this comparison in mind we can improve hashing methods. All known hashing schemes (Knuth [4]) make use of a table of length m such that $n \leq m \leq kn$ for some constant k. By using this structure we may use larger tables e.g., $O(n^2)$. Since there is no need to initialize the table and the size of the table is larger[1], we obtain algorithms with better average and worst case time.

We now present several reductions between these problems. By $P_1 \alpha P_2$ we mean that problem P_1 can be reduced to an instance of problem P_2 after at most O(n) operations in O(n) space, where n is the size of problems P_1 and P_2. Thus a linear time and space algorithm for P_2 implies one for P_1.

The reduction set equality α subset is shown by Reingold [6]. Disjoint set α element uniqueness is trivial. Element uniqueness α min-d gap is shown by Shamos and Hoey [7] and similarly we can obtain it for mode and multiset representation. Set equality α IRS and set equality α multiset representation can be easily obtained. The lower bound of $O_c(n\log n)$ on the time complexity for these problems follows from these reductions and the proof of Reingold [6] for the set equality problem and the proof of Dobkin and Lipton [3] for the element uniqueness problem.

---

[1] We assume n is sufficiently large so that $k_1 n^2 > kn$

We now show the equivalence[1] between dictionaries and the time - space problem.

Lemma 1 The dictionary and time - space problems are equivalent.

Proof: The proof is presented in two parts.

a) time - space α dictionary

Given any algorithm with $T_1(n)$, $M_1(n)$ and $S_1(n)$ such that $O(M_1(n)) \leq O(T_1(n)) < O(S_1(n))$, where $T_1(n)$, $M_1(n)$ and $S_1(n)$ are functions representing the asymptotic time, memory and space complexity of the algorithm. We are going to simulate this algorithm by the use of a dictionary. The keys in the dictionary are the specific memory locations used and every time a memory is inserted in the dictionary a new consecutive address is assigned to it. The simulation will permit the algorithm to execute any operation until there is a request to use a particular memory m, in this case we search for m in the dictionary. If the search is successful (we have already assigned an address to m) then its address is returned to the original algorithm and let it continue. Otherwise we assign the next consecutive address to m, insert it into the dictionary and return the address to the original algorithm.

Thus if we can execute N operations on a dictionary in O(N) time and space, then we have an algorithm for the original

---

[1] Two problems P_1 and P_2 are equivalent iff $P_1 \alpha P_2$ and $P_2 \alpha P_1$.

problem of time $T_2(n) = O(T_1(n))$, space $S_2(n) = O(M_1(n))$ and memory $M_2(n) = O(M_1(n))$.

b) dictionary α time space

This part of the proof follows directly from the algorithm which we present for the dictionary problem. This algorithm is of the form required by the time - space problem.

We should note that if we have an algorithm of time $T_1(n)$, space $S_1(n)$ and memory $M_1(n)$, by using lemma 1 and if we implement the dictionary by balanced search trees, we automatically obtain an algorithm of time $T_2(n) = O(T_1(n) \log(T_1(n))$, space $S_2(n) = O(M_1(n))$ and memory $M_2(n) = O(M_1(n))$.

Now we present an algorithm that executes n instructions in a dictionary in O(n) time.

Line Algorithm DICT

//Global variables used:

table: Vector of size M-L (where M is the largest and L the smallest key) used to store indexes. Initially it may contain any information.

elem: Vector used to store the elements previously inserted.

Entries for the algorithm

INITIALIZE, DELETE, INSERT and SEARCH.//

```
     //entry for initialization//
 1   entry  INITIALIZE(i,z)
     //i is the number of insertions made and z is the smallest
       key to be used.//
 2   i ← 0; L ← z
 3   return
     //entry for insertions//
 4   entry  INSERT(i,x)
     //x:  key to be inserted (assume to be between L and M)
       i:  number of successful insertions in dictionary//
 5   if  SEARCH(i,x) = false
 6             then [table (x-L) ← i; elem (i) ← x; i ← i+1]
 7   return
     //entry for deletions//
 8   entry  DELETE(i,x)
     //x and i as in insert//
 9   if  SEARCH(i,x) = true
10             then [elem (table (x-L)) ← L-1]
11   return
     //entry for searching.  SEARCH is a logical function//
12   entry  SEARCH(i,x)
     //x and i as in insert//
13   z ← x-L
     //next instruction is executed from left to right//
14   if  0≤table (z)≤i-1 and elem (table (z)) = x
15             then [SEARCH ← true; return]
16             else [SEARCH ← false; return]
17   end of algorithm DICT.
```

The following two lemmas will show that algorithm DICT correctly implements a dictionary in linear time.

Lemma 2 Algorithm DICT executes correctly any insert, delete and search operation.

Proof: The proof is by induction on the number of successful insertions.

basis: We prove that any instructions until the first insert is executed correctly.

From line 14, clearly any search operation will be false. Then any delete operation should do nothing. The first key we want to insert will be inserted as the search in line 5 will be false.

induction hypothesis: Assume every instruction until the i^{th} successful insert have been carried out correctly.

induction step: We now prove that any instruction after the i^{th} successful insert until the $i^{th}+1$ successful insert are executed correctly.

case i: key x has never been inserted nor deleted.

Then x is not in the element list and from line 14 we know that any search for x would be false. Delete x would do nothing and if insert x then in line 6 we would establish a 1-1 correspondence between the element list and the index table for key x.

case ii: The last insert or delete on key x was a delete before the i^{th} insert.

From the induction hypothesis we know that delete x was done properly, so x is not in the element list, therefore the same arguments used in case i also apply.

case iii: The last insert or delete on key x was an insert x.

Then there is a 1-1 correspondence for x between the element list and index table, so any search x would be true. An insert x would do nothing and a delete x would eliminate x from the element list so that further search x would be false.

case iv: The last insert or delete on key x was a delete after the i^{th} insert.

Then from case i, ii, or iii we know that the first delete x after the i^{th} insert was done correctly, so x is not in the element list, therefore the same proof as in case i applies. ■

Lemma 3 n delete, insert or search operations in algorithm DICT can be carried out in O(n) time on a random access machine.

Proof: As there are no loops, any call to this algorithm would take a constant amount of time, therefore n entries must take O(n) time. ■

We now show how to solve several problems by the use of dictionaries.

i) element uniqueness: We execute the following instructions: (insert x_i, search x_{i+1}) for i=1,2,...,n-1. The output is true if all search operations are false.

ii) symmetric difference: From set theory we know that A+B = (A-B) $\cup$ (B-A), so we just call the relative complement and set union algorithms. Note that to solve set equality we just perform the symmetric difference between the sets and check to see if the result is empty.

iii) relative complement: To solve this problem we make use of the following operations: insert b_i for i=1,2,...,m (b_i's are elements of set B), search a_i for i=1,2,...,n (a_i's are elements of set A), if for any a_i the search is successful, then a_i is not part of A-B, otherwise it is part of A-B.

iv) subset: we just call the relative complement and check if answer is empty.

v) set intersection: The following operations are performed: insert a_i for i=1,2,...,n and search b_i for i=1,2,...,m. Whenever a search is successful, element b_i is part of the solution set.

vi) set union: The operations we should execute are insert a_i for i=1,2,...,n and insert b_i for i=1,2,...,m, the resulting element list is the solution.

vii) mode: We write an algorithm for this problem.

```
Line  Algorithm  MODE  (X,n)
      //We are going to find the largest ℓ for which there is
        a mode.
        L is assumed to be the min {x_1,x_2,...,x_n}
        count:  a vector that has as values the number of times
                an element is repeated//
 1    max ← 0
 2    call INITIALIZE(i,L)
 3    for j ← 1 to n do
 4         if SEARCH(i,x_j) = true
 5              then [count(table(x_j-L)) ← count(table(x_j-L))+1
 6                   if count(max) < count(table(x_j-L))
 7                        then [max ← table(x_j-L)]]
 8              else [count(i) ← 1
 9                   call INSERT(i,x_j)]
10    end
11    return (count(max),elem(max))
12    end of MODE
```

viii) Multiset: This algorithm is very similar to the mode, just eliminate line 1, 6, 7, 11 and after line 10 do the following.

```
      for j ← to i-1 do
                Output (count(j),elem(j))
      end
```

ix) min gap

```
Line  Algorithm  MINGAP(X,n,L,M,d)
      //Assume L is smallest element//
      //In this algorithm we are going to place together a
        group of observations.  We are going to have several
        bins, where bin_0 contains element L, bin_1 contains
        elements (L; L+d]..... and the last, bin_{k+1} contains
        elements (L+kd; M] where M is the largest x_i and
        (M-(L+kd)) ≤ d.
        Vector B is used to store the value of the elements in
        the bins//
  1   call INITIALIZE(i,0)
  2   for j ← 1 to n do
  3       k ←⌈(x_i-L)/d⌉
  4       if SEARCH(i,k) = true
  5           then [OUTPUT(B(table(k)),x_j)
  6                return]
  7           else [if (M-x_j) ≥ d
  8                then [if SEARCH(i,k+1) = true
  9                     then [if B(table(k+1)) - x_j ≤ d
 10                          then [Output (B(table(k+1)),x_j)
 11                               return]]]
 12                if x_j≠L
 13                     then [if SEARCH(i,k-1)=true
 14                          then [if x_j-B(table(k-1)) ≤ d
 15                               then [Output (B(table(k-1),x_j))
```

```
16                                  return]]]
17                  B_i ← X_j, call INSERT(i,k)]
18  end
19  output (no result)
20  end of MINGAP.
```

x) IRS: this problem is included for its practical application. This algorithm is feasible provided sufficient auxiliary memory on disks or drums is available.

```
Line  Algorithm IRS(n,m)
      //input the employer forms, assume there are m of such
        forms.  <x,y,z>:  employer x paid employee y, z dollars.
        L:  smallest employee social security numbers//
 1    call INITIALIZE(i,L)
 2    for j ← 1 to m do
 3         input <x,y,z>
 4         if SEARCH(i,y)=true
 5              then [//sets S_i are stored sequentially as all
                       employees are assumed to have at most a
                       constant number of jobs//
 6                    S_table(y-L) ← S_table(y-L) ∪ {<x,z>}]
 7              else [S_i ← {<x,z>}
 8                    call INSERT(i,y)]
 9   end
```

```
     //read employee forms, assume n of such forms//
10   for j ← 1 to m do
11        input <u,v,w> //employee u, received from employer
                        v, w dollars//
12        if SEARCH(i,u) = true
13             then [if <v,w> ∈ S_table(u-L)
14                  then [S_table(u-L) ← S_table(u-L) - {<v,w>}]
15                  else [//Send message to employer stating that
                          employee reported this salary//
16                        output <u,v,w>]]
17             else [//send message to employer stating that
                     employee reported this earning//
18                   output <u,v,w>]
19   end
20   for j ← 0 to i-1 do
21        while S_j≠0 do
               //Send message to employee stating that employer
                 reported this earning//
22             <x,z> ← S_j          //copy first pair from set S_j//
23             S_j ← S_j - <x,z>
24             output <x,elem(j),z>
25        end
26   end
27   end of IRS
```

Theorem 1: Algorithms to solve problems i to x will take linear time on a random access machine.

Proof: Let us assume that n is the length of the input for these problems. From lemma 3 and the fact that all of these problems make O(n) calls to algorithm DICT, it follows that the time complexity is O(n). ■

III. Gap Problem

In this section we are going to look at the max-d gap problem. Clearly max gap α max-d gap. Shamos and Hoey [7] claim the reduction element uniqueness α max-d gap, however, it is not correct. We now present an O(n) time and space algorithm for the max-d problem. It is surprising that such algorithm exists for this problem, as it is very similar to the min-d gap for which there is no known linear time solution, and the min gap even though it has a linear time solution, its space might be very large. Sahni [8] also reported two similar problems, one of which, the Kolmogorov - Smirnov with a continuous distribution function can be solved in O(n) time and space (see [5]), however the set equality problem can be reduced to the K-S test with discrete functions. Again we have two very similar problems, one with a linear time and space solution and the other that requires O(nlogn) time and is as hard as the set problems.

The technique used to solve this problem is similar to the one used by Gonzalez, Sahni and Franta [5] for the Kolmogorov - Smirnov and Lilliefors tests.

Line Algorithm MAXdGAP(n,X,d)

//This algorithm finds the maximum gap (d) between n ordered points. The input are n unordered points initially stored in vector X. Output is returned in d.

```
20          until XMAX_j ≠ small
21          if XMIN_j - XMAX_i > d then [d ← XMIN_j - XMAX_i]
22          i ← j
23      end
24      end of algorithm MAXdGAP.
```

Lemma 4: Algorithm MAXdGAP computes the correct value for the maximum gap d.

Proof: Since there are n+1 bins, it must be that at least one of them is empty. Furthermore we know it cannot be the first nor the last bin. Then the maximum gap must be $\geq \frac{1}{n} * diff$. But the maximum gap between points in the same bin is $< \frac{1}{n} * diff$, so we need only consider distances between points that belong to different bins (lines 17-24). Therefore d is computed correctly. ■

Theorem 2: Algorithm MAXdGAP has O(n) time and space complexity.

Proof: Obvious. ■

We should note that these gap problems are similar to the problems studied in [2]. A related problem is to find the k max-d gaps. This problem takes O(kn) time and O(n) space. The solution is by finding the largest gap, then substracting the gap to numbers above the gap. This process is repeated k times.

2 Vectors of size n+1 each are made use of:

$XMAX_i$...Maximum point in bin i
$XMIN_i$...Minimal point in bin i
$0 \leq i \leq n$ //

```
    //Find the smallest and largest elements//
 1  big ← small ← x_1
 2  for i ← 2 to n do
 3       if x_i ≤ big then [if x_i < small then [small ← x_i]]
 4                         else [big ← x_i]
 5  end
    //Initialize bins//
 6  for i ← 0 to n do
 7       XMIN_i ← big
 8       XMAX_i ← small
 9  end
    //Place x_i   into bins//
10  diff ← big - small
11  for i ← 1 to n do
12       j ← ⌈((x_i - small)/diff) *n⌉  //find bin//
13       if XMAX_j < x_i then [XMAX_j ← x_i]
14       if XMIN_j > x_i then [XMIN_j ← x_i]
15  end
    //Find interbin distances//
16  d ← 0; i ← 0; j ← 0
17  while i < n do
18       repeat
19            j ← j+1
```

IV. Conclusions

Even though the space required for the dictionary problem might be very large, we could have several search procedures over the same space as long as we take care of collisions, so the unused storage can be decreased. The space used for the table need not be scratch, for example it might be programs, files, etc., as long as they remain inactive. Whenever we use one of those stores, the information would be saved and at the end restored.

There are still several open problems. Is there a linear time solution for the min-d gap? Find algorithms which require less space for these set problems. Can we solve some of these problems in linear space but say nloglogn time? Note that any key comparison algorithm takes $O_c(n\log n)$.

For the reduced problem (e.g., keys in the range $[0; n^k)$) a O(kn) algorithm is known for some of these problems by the use of radix sort. Our method is still an improvement. The difference in time might not be very impressive if all our data can fit in the internal memory of the computer. However, if it is not the case, the difference on time is more impressive when using external storage devices.

References

1. Aho, A. V., Hopcroft, J. E. and Ullman, J. D., "The Design and Analysis of Computer Algorithms," Addison-Wesley, 1974.

2. Bodin, L. D., "A Graph Theoretic Approach to the Grouping of Ordering Data," Networks, 2, 1972, pp. 307-310.

3. Dobkin, D. and Lipton, R., "On the Complexity of Computations under Varying Sets of Primitives," Yale Univ., Dept. of Comp. Sci., Technical Report #42, 1975.

4. Knuth, D. E., "The Art of Computer Programming. Vol. III, Sorting and Searching," Addison-Wesley, 1973.

5. Gonzalez, T., Sahni, S. and Franta, W. R., "Efficient Algorithms for the Kolmogorov - Smirnov and Lilliefors Tests," to appear ACM TOMS.

6. Reingold, E. M., "On the Optimality of Some Set Algorithms," JACM, Vol. 19, No. 4, Oct. 1972, pp. 649-659.

7. Shamos, M. I. and Hoey, D., "Closest-Point Problems," Proceedings of the 16th Annual Symposium of Foundations of Computer Science, Oct. 1975, pp. 151-162.

8. Sahni, S., private communication, Jan. 1975.