

**SET INTERSECTION USING BAEZA YATES ALGORITHM**

Submitted by :

Student Number: Subject:

Submitted to: Submitted On:

Srinivas Kasturi

3078674

Web Docs & Databases

Dr. Yangjun Chen

04-14-2017

**(Web Docs & Databases GACS-7102-001)**

CONTENTS

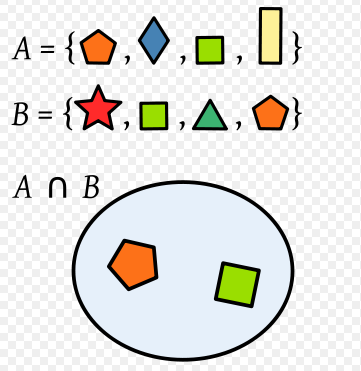
* Set Intersection
* Related Work
* Introduction
* Problem
* A New Algorithm and Its Analysis
* Implementation in c++
* Experimental Results
* Applications
* Conclusions

**Abstract:**

This paper presents and carefully studies a simple intersection set of computer instructions for sorted sequences that is fast on average. It is related to the multiple searching problem and to merging. We present the worst and average case analysis, showing that in the first thing just mentioned, the complex difficulty nicely changes to fit the smallest list size. In the last thing just mentioned case, it (does/completes) less comparison than the total number of elements on both inputs, n and m, when n = αm (α > 1), achieving O(m log(n/m)) complex difficulty. The set of computer instructions is (having a reason to do something) by its computer program to fast question processing in Web search engines, where large intersections, or differences, must be (sang, danced, acted, etc., in front of people)fast. In this case we experimentally show that the set of computer instructions is faster than previous solutions.

**Set Intersection:**

In mathematics, the **intersection** A ∩ B of two **sets** A and B is the **set** that contains all elements of A that also belong to B (or equivalently, all elements of B that also belong to A), but no other elements.



**Introduction**

**Multiple searching:**

Given an n-element data multi set, D, drawn from an ordered universe, search D for each element of an m-element query multiset, Q, drawn from the same universe. An algorithm solving the problem must report any elements in both multi sets.

The complexity metric is the number of three-way comparisons (<,=,>) between any pair of elements, worst case or average case (Rawlins, 1992; Baeza-Yates et al., 1993). This problem is motivated by Web search engines. Most search engines use inverted indexes, where for each different word, we have a list of positions or documents where it appears. In some settings those lists are ordered by position or by a global precomputed ranking, to facilitate set operations between lists (derived from Boolean query operations), which is equivalent to the ordered case.

Therefore, the complexity of this problem is interesting also for practical reasons, as in search engines, partial lists can have hundreds of millions elements for very frequent words.

**Our problem:**

Multiple search when and are sets already ordered, hence we obtain the intersection of both sets. We use n=|D|, m=|Q|

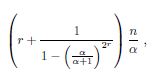
And n > m

**Related works:**

This is an application of what is called *doubling search* or *galloping search*, in which mimics binary search for unbounded sequences obtaining the same *O*(log *n*) complexity, a classical result of Bentley and Yao. If we find *x*, we add it to the result. Then, we remember the position where *x* was (or the position where it should have been) so we know that from that position backwards we already processed the set.

Lower bounds for the element uniqueness problem and merging (sorted case) do not apply to the multiple search problem (but the converse is true). For the ordered case, lower bounds on set intersection are also lower bounds for merging both sets. However, the converse is not true, as in set intersection we do not need to find the actual position of each element in the union of both sets, just if it is in D or not.

Fernandez de la Vega et al. (1998) analyzed the average case of a simplified version of Hwang-Lin’s binary merge (1972) finding that if α=n/m with α>1 and not a power of 2, we get



Where r= [lg2α]

Adaptive multiple set intersection algorithm (Demaine et al., 2000 & 2001). They also defined the difficulty of a problem instance, which was refined later by Barbay and Kenyon (2002).

**BASES for Set Intersection**

For the purpose of this set intersection we need to perform the following operations

1. Merging: *O(m+n)*

In this process of merging we need to gather all the elements of two sets which we need to intersect.

1. Binary search the smaller set in the larger: *O(m ln n)*

In this binary search process we need to search all the elements below the median and above the median ranges.

Doing this above operations faster & perhaps on average is the required function.

**BAEZA-YATES ALGORITHM**

* To overcome the above problems we use this this Algorithm.
* This is also called as double binary search algorithm.

This is a balanced version of “Hwang and Lin’s Algorithm” because roots were adopted from it.

Let us consider two sets of the data such as

Q={1,2,3………1000}

D={1,2,3,4……..100000}

If the set D is sorted and Q is smaller than D then we can search every element of Q in D by using the binary search.

If both the sets are sorted? Then in that case set intersection can be solved by merging both the sets. Straight merging requires {m+n-1} comparisons. We call it as a double binary search and can be called as balanced algorithm which is adopted from the “Hwang & Lin’s” algorithm.

Now let us go to the cases in deep. Let us consider set “Q” as a small one & now perform the binary search of median of Q in D. If we found that values which are lesser than median of ‘Q’ in the left position of median in ‘D’ & elements bigger than median in right of ‘D’.

If Q’s size is larger than ‘D’ then exchange the roles of ‘Q’ & ‘D’.

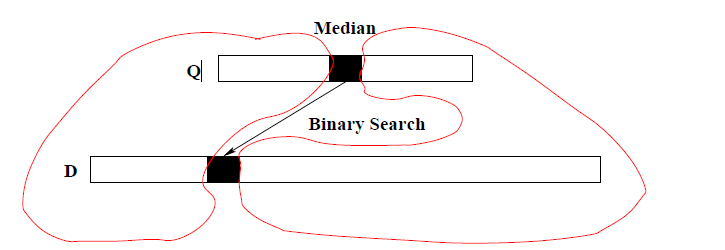
A simple way to improve this algorithm is to start comparing the smallest elements of both the sets with the largest elements in both. If both sets don’t overlap, we use O(1) time. Otherwise, we search the smallest & largest elements of D in Q, to find the overlap, just O(lg m) time. Then we apply the previous algorithm just to subset that actually overlap.

**TIME COMPLEXITY:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Cases** | **Search** | **insert** | **Merge** |
| **Best Case** | **O(1)** | **O(lg2n)** | **O{m+n}** |
| **Average Case** | **O(1)** | **O{n}** | **O{m+n}** |
| **Worst Case** | **O(**lg2α**)** | **O{n}** | **O{m+n}** |

**A Simple but Good Average Case Algorithm:**

* Double Binary Search: Can be seen as a balanced version of Hwang and Lin’s algorithm adapted to set intersection.



* Solve both sub-problems (merging & Bin Search) recursively.
* i.e by exchange Q and D if |Q|> |D|.

**Improvements Small improvements in theory:**

* We compare the smallest elements of both sets with the largest elements in both sets, is ***O (1)***.
* Otherwise, we search the smallest and largest element of D in Q, to find the real overlap, using just time ***O(lg m)***.
* Doing the opposite is not worth it for small.

**Analysis:**

In the case of analysis we are going to deal with both the Best case and Worst cases of this algorithm.

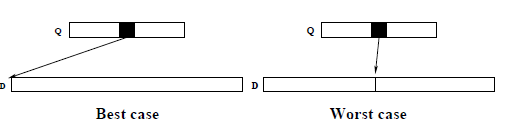
**Best Case:**

Best case of the algorithm is where we can find the median value at the starting position and we can reinitiate one sided search and merging. Due to this we can obtain the intersection results as early as possible.

Operations that takes place in the best case are:

[lg(m+1)][lg(n+1)]

If m=O(n) is O(lg2n)

****

**Worst Case:**

Worst case of the algorithm is where we can find the median value at the exact middle of the other set due to this we should initiate the binary search on both the sides. Due to this it will take more time to initiate the intersection of both the sets and merging algorithm can’t be used effectively in this case.

Operations that takes place in the Worst case are:

*If m =2k-1 we have*

*W(m,n) = [lg (n+1)] + W ((m-1)/2,[n/2]) + W((m-1)/2,[n/2])*

*Then*

*W(m,n) = 2(m+1)lg((n+1)/(m+1))+2m+O(lg n)*

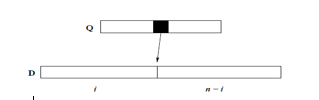
If n=αm it is O(n) and the ratio between this algorithm and merging is

2(1 + lg(α))/(1+α)

**Average Case Analysis**

We use two pessimistic assumptions here in this first average case analysis

1. We never find the median of Q in D
2. Then the median will divide D in the sets of Size i and n-i with the same probability for all i



After the division the binary search will be initiated for searching matches in both the left and right divisions of D for the Q’S matches.

**Query Processing in Inverted Indexes:**

Inverted indexes are a vocabulary and a list of references per word to its occurrences in documents and optionally to where they occur per document (full inversion)

Lists usually use linked blocks of references (variable size)

The references are sorted in:

* Boolean model: intersection is basic for set operations
* Vectorial model: by chunks of doc-ids with similar term-frequency
* Word positions per document: shifted intersection
* (full inversion, needed for phrase search)
* Proximity search: that is *i±k*

In this algorithm all the references were sorted in the above given sequences and the processing will be initiated sequentially for the intersection results.

**ALGORITHM**

Intersect(D, Q, minD,maxD, minQ,maxQ)

1. //if Q or D are empty, we finish the recursion

2. if minD > maxD bfor minQ > maxQ

3. return ∅

4. miqQ ← round((minQ + maxQ)/2)

5. midQval ← Q[midQ]

6. midD ← binsearch(midQval,D, minD,maxD)

7. if |D[minD..midD − 1]| > |Q[minQ..midQ − 1]| // subset(D) > subset(Q)

8. Result ← Result ∪ Intersect(D, Q,minD,midD − 1,minQ,midQ− 1)

9. else //we exchange the roles of D and Q

10. Result ← Result ∪ Intersect(Q,D, minQ,midQ − 1,minD,midD − 1)

11. if D[midD] == midQval

12. Result ← Result ∪ {midQval}

13. midD ← posModD − 1

14. if |D[midD + 1..maxD]| > |Q[midQ + 1..maxQ]|// subset(D) > subset(Q)

15. Result ← Result ∪ Iintersect(D, Q, midD,maxD, midQ + 1,maxQ)

16. else //we exchange the roles of D and Q

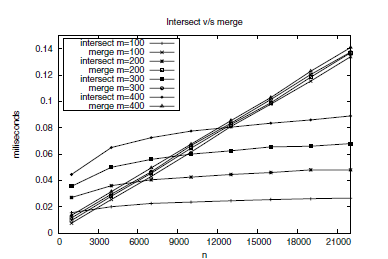
17. Result ← Result ∪ Intersect(Q,D, midQ + 1, maxQ, midD,maxD)

18. return Result

**Experimental Analysis**

We compare the efficiency of the algorithm, which we call Intersect in this section, with an intersection algorithm based on merging, and with an adaptation of the Adaptive algorithm for the intersection of two sequences. In addition, we show the results obtained with the optimizations of the algorithm.

We used sequences of integer random numbers, uniformly distributed in the range. We varied the length of one of the lists (n) from 1,000 to 22,000 with a step of 3,000. For each of these lengths we intersected those sequences with sequences of four different lengths (m), from 100 to 400. We use twenty random instances per case and ten thousand runs (to eliminate the variations due to the operating system given the small resulting times).

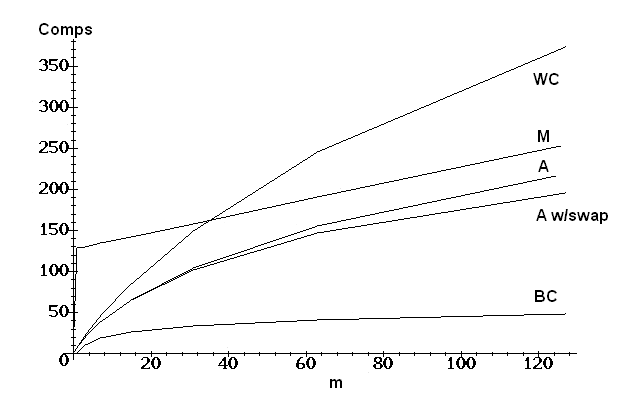


Experimental results for Intersect and Merge for different values of *n* and *m*

**Actual comparisons of cases**

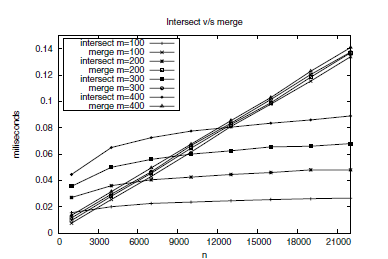
In the below graph we are comparing the BC (best case), WC (worst case) medium case and actual cases with the symmetric inputs as given below. This graphical readings show that this algorithm can even sync with the worst cases to provide best results.

* For n=128 and all powers of 2 for m≤n.
* Number of comparisons in the best, worst and average case (with and without swaps) for *n* = 128, as well as for merging (M)

****

**Experimental Results**

* We use uniformly random integer numbers in the range 1 to 109 and we implemented the algorithms using the Gcc 3.3.3 compiler in a Linux platform running on an Intel Xeon of 3GHz.

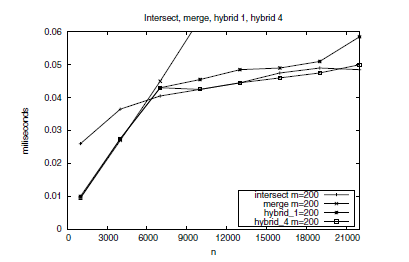


**Hybrid Algorithm**

When merge is better, stop the recursion. We found the empirical line when this happens in our implementation.

We can see from the experimental results obtained for the basic algorithm that there is a section of values of n where Merge is better than Intersect. Hence, a natural idea is to combine both algorithms in one hybrid algorithm that runs each of them when convenient. However this will depend on the data, the implementation, and the actual hardware and software platform used. So the following discussion is based on our context but can be replicated, possibly with different results, for other cases.

The hybrid algorithm works by running Merge whenever m > 0.033n+8.884, and running Intersect otherwise. The condition is evaluated on each step of the recursion.



Experimental results for Intersect, Merge and the hybrids 1 and 4 for different values of *n* and for *m* = 200

**Implementation in C++ :**

For implementing this algorithm in C++ we are using the pre-defined boost libraries in which this Baeza intersect algorithm is defined and integrated with the predefined libraries.

**First we are building 2 sets using random functions:**

void createSets( vector<int>& set1, vector<int>& set2 )

{

srand ( time(NULL) );

set1.reserve(100000);

set2.reserve(1000);

// Create 100,000 values for set1

for ( int i = 0; i < 100000; i++ )

{

int value = 1000000000 + i;

set1.push\_back(value);

}

// Try to get half of our values intersecting

float ratio = 200000.0f / RAND\_MAX;

// Create 1,000 values for set2

for ( int i = 0; i < 1000; i++ )

{

int random = rand() \* ratio + 1;

int value = 1000000000 + random;

set2.push\_back(value);

}

**The process of left intersection and right intersection is implemented as below:**

probe1 = begin1 + ( ( end1 - begin1 ) >> 1 );

probe2 = lower\_bound< Probe >( begin2, end2, \*probe1 );

baeza\_intersect< Probe >(begin1, probe1, begin2, probe2, out); // intersect left

if (! (probe2 == end2 || \*probe1 < \*probe2 ))

\*out++ = \*probe2++;

baeza\_intersect< Probe >(++probe1, end1, probe2, end2, out); // intersect right

}

else

{

if ( begin2 == end2 )

return;

probe2 = begin2 + ( ( end2 - begin2 ) >> 1 );

probe1 = lower\_bound< Probe >( begin1, end1, \*probe2 );

baeza\_intersect< Probe >(begin1, probe1, begin2, probe2, out); // intersect left

if (! (probe1 == end1 || \*probe2 < \*probe1 ))

\*out++ = \*probe1++;

baeza\_intersect< Probe >(probe1, end1, ++probe2, end2, out); // intersect right

}

}

After running the set intersection for both these sets the ‘timer.h’ function finally calculates the time takes to set intersection in the form of milliseconds.

We have another developed same algorithm with using the BOOST libraries and just by using **stl\_intersection** for both binary sets that were divided.

**OUTPUT of C++ Code:**

* For a large set of q=100000
* And small set of d=1000



**Output of Java Code**

When this same algorithm is implemented in java

* For a large set of q=100000
* And small set of d=1000



When compared to C++ java can produce more results in less amount of time this is because of the heap memory used by the java where as it is not possible in C++.

**Hardware Specifications:**

OS: Windows 10

Processor: Intel core i5

Ram : Min 2gb

**Software Required:**

IDE: Microsoft Visual Studio

Language: C++

**Advantages:**

* This Set intersection algorithm is a fundamental operation in information retrieval and database systems.
* This algorithm shows the same times than Intersect in the section where the latter is better than Merge, combining the advantages of both algorithms in the best way.
* It can used for both traditional AND-mode querying, and also in ranked querying environments when dynamic pruning techniques are used, or when pre-computed static scores such as PageRank contribute to answer ordering.

**Disadvantages while applying in web**

* Global ranking schemes (e.g. PageRank): document ids are defined by the ranking which are predefined bot the crawlers.
* The distribution of word occurrences is a power law, and the same is true with query frequencies.
* Correlation is almost null in the usage of this algorithm when compared to the other intersecting algorithms.
* That means that the average lengths of the lists, m and n, when sampled, will satisfy n=Ө(m)(uniform), rather than n= m+O(1)(power law).

**Future Work**

A simple way to improve this algorithm is to start to comparison of both the sets at the beginning to decide the smallest and by this we can save more time and execution will be very fast.

**Conclusion**

In practice, Web queries are short: 1 to 3 words. Hence, there is almost no need to do multiset intersection and if so, they can be easily handled by pairing the smaller sets firsts. We need partial evaluation, as most people only look at less than two result pages. Adaptive algorithm that finds first the first answers and in the right order? That implies first to do the recursion on the left side.

**References:**

1. Ricardo Baeza-Yates1 and Alejandro Salinger2 Fast Intersection Algorithms for Sorted Sequences.
2. Baeza-Yates, R.A.: Efficient Text Searching. PhD thesis, Dept. of Computer Science.

University of Waterloo (May 1989); Also as Research Report CS-89-17

1. Baeza-Yates, R.A., Bradford, P.G., Culberson, J.C., Rawlins, G.J.E.: The Complexity

of Multiple Searching (1993) (unpublished manuscript)

1. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval, 513 pages.

ACM Press/Addison-Wesley, England (1999)

1. Baeza-Yates, R.A., Saint-Jean, F.: A three level search engine index based in query

log distribution. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE

2003. LNCS, vol. 2857, pp. 56–65. Springer, Heidelberg (2003)

1. Baeza-Yates, R.A.: Query usage mining in search engines. In: Scime, A. (ed.) Web

Mining: Applications and Techniques. Idea Group, USA (2004)