# A Fast Set Intersection Algorithm for Sorted Sequences

Ricardo Baeza-Yates

Center for Web Research
Departamento de Ciencias de la Computación,
Universidad de Chile, Casilla 2777, Santiago, Chile
rbaeza@dcc.uchile.cl

**Abstract.** This paper introduces a simple intersection algorithm for two 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 former, the complexity nicely adapts to the smallest list size. In the later case, it performs less comparisons than the total number of elements on both inputs when $n = \alpha m$ ($\alpha > 1$). Finally, we show its application to fast query processing in Web search engines, where large intersections, or differences, must be performed fast.

**Keywords:** Set operations, merging, multiple search, Web search engines, inverted indices.

## 1 Introduction

Our problem is a particular case of a generic problem called multiple searching [2] (see also [15], research problem 5, page 156). Given an $n$-element data multiset, $D$, 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 multisets. The metric is the number of three-way comparisons ($<, =, >$) between any pair of elements, worst case or average case. Throughout this paper $n \geq m$ and logarithms are base two unless explicitly stated otherwise.

Multiply search is directly related to computing the intersection of two sets. In fact, the elements found is the intersection of both sets. Although in the general case, $D$ and $Q$ are arbitrary, an important case is when $D$ and $Q$ are sets (and not multisets) already ordered. In this case, multiply search can be solved by merging both sets. However, this is not optimal for all possible cases. In fact, if $m$ is small (say if $m = o(n/\lg n)$), it is better to do $m$ binary searches obtaining an $O(m \lg n)$ algorithm. Can we have an adaptive algorithm that matches both complexities depending on the value of $m$? We present an algorithm which on average performs less than $m+n$ comparisons when both sets are ordered under some pessimistic assumptions. Fast average case algorithms are important for large $n$ and/or $m$.

S.C. Sahinalp et al. (Eds.): CPM 2004, LNCS 3109, pp. 400–408, 2004.

This problem is motivated by Web search engines. Most search engines use inverted indices, 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. In other settings, the lists of positions are sorted by frequency of occurrence in a document, to facilitate ranking based on the vectorial model [1, 3]. The same happens with word positions in each file (full inversion to allow sentence searching). 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.

In section 2 we present related work. Section 3 presents our intersection algorithm and its analysis. Section 4 presents the motivation for our problem and some practical issues. We end with some concluding remarks and on-going work.

## 2 Related Work

If an algorithm determines whether any elements of a set of $n+m$ elements are equal, then, by the element uniqueness lower bound in algebraic-decision trees (see [11]), the algorithm requires $\Omega((n+m)\lg(n+m))$ comparisons in the worst case. However, this lower bound does not apply to the search problem because a search algorithm need not need to determine the uniqueness of either $D$ or $Q$; it need only to determine whether $D \cap Q$ is empty. For example, an algorithm for $m = 1$ must find whether some element of $D$ equals the element in $Q$, not whether any two elements of $D$ are equal. Conversely, however, lower bounds on the search problem (or, equivalently, the set intersection problem) apply to the element uniqueness problem [10]. In fact, this idea was exploited by Demaine *et al.* to define an adaptive multiple set intersection algorithm [8, 9]. They also defined the difficulty of a problem instance, which was refined later by Barbay and Kenyon [6].

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. Although there has been a lot of work on minimum comparison merging in the worst case, almost no research has been done on the average case because it does not make much of a difference. However, this is not true for multiple search, and hence for set intersection [2].

In the case of merging, Fernandez de la Vega *et al.* [13] analyzed the average case of a simplified version of Hwang-Lin's binary merge [14] finding that if $\alpha = n/m$ with $\alpha > 1$ and not a power of 2, then the expected number of comparisons was

$$\left( r + \frac{1}{1 - \left(\frac{\alpha}{\alpha+1}\right)^{2^r}} \right) \frac{n}{\alpha}$$

with $r = \lfloor \lg_2 \alpha \rfloor$. When $\alpha$ is a power of 2, the result is more complicated, but similar. Fernandez de la Vega *et al.* [12] also designed a probabilistic algorithm that improved upon Hwang-Lin's algorithm on the worst case for $1.618m \leq n \leq 3m$.

In the case of upper bounds, good algorithms for multiple search can be used to compute the intersection of two sets, obtaining the same time complexity. They can be also used to compute the union of two sets, by subtracting the intersection of both sets to the set obtained by merging both sets. Similarly to compute the difference of two sets.

## 3 A Simple but Good Average Case Algorithm

Suppose that $D$ is sorted. In this case, obviously, if $Q$ is small, will be faster to search every element of $Q$ in $D$ by using binary search. Can we do better if both sets are sorted? In this case set intersection can be solved by merging. In the worst or average case, straight merging requires $m+n-1$ comparisons. Can we do better for set intersection? The following simple algorithm improves on average under some pessimistic assumptions. We call it double binary search and can be seen as a balanced version of Hwang and Lin's [14] algorithm adapted to our problem.

We first binary search the median (middle element) of $Q$ in $D$. If found, we add that element to the result. Found or not, we have divided the problem in searching the elements smaller than the median of $Q$ to the left of the position found on $D$, and the elements bigger than the median to the right of that position. We then solve recursively both parts using the same algorithm. If in any case, the size of the subset of $Q$ to be considered is larger than the subset of $D$, we exchange the roles of $Q$ and $D$. Note that set intersection is symmetric in this sense. If any of the subsets is empty, we do nothing.

In the best case, the median element in each iteration always falls outside $D$ (that is, all the elements in $Q$ are smaller or larger than all the elements in $D$). Hence, the total number of comparisons is $\lceil \lg(m+1) \rceil \lceil \lg(n+1) \rceil$, which for $m = O(n)$ is $O(\lg^2 n)$. This shows that there is room for doing less work. The worst case happens when the median is not found and divides $D$ into two sets of the same size (intuitively seems that the best and worst case are reversed). Hence, if $W(m,n)$ is the cost of the set intersection on the worst case, for $m$ of the form $2^k - 1$, we have

$$W(m,n) = \lceil \lg(n+1) \rceil + W((m-1)/2, \lceil n/2 \rceil) + W((m-1)/2, \lfloor n/2 \rfloor)$$

It is not difficult to show that

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

That is, for small $m$ the algorithm has $O(m \lg n)$ worst case, while for $n = \alpha m$ it is $O(n)$. In this case, the ratio between this algorithm and merging is $2(1+\lg(\alpha))/(1+\alpha)$ asymptotically, being 1 when $\alpha = 1$. The worst case is worse

than merging for $1 < \alpha < 6.3197$ having its maximum at $\alpha = 2.1596$ where it is 1.336 times slower than merging (this is shown in figure 1). Hence the worst case of the algorithm matches the complexity of both, the merging and the multiple binary search, approaches, adapting nicely to the size of $m$.

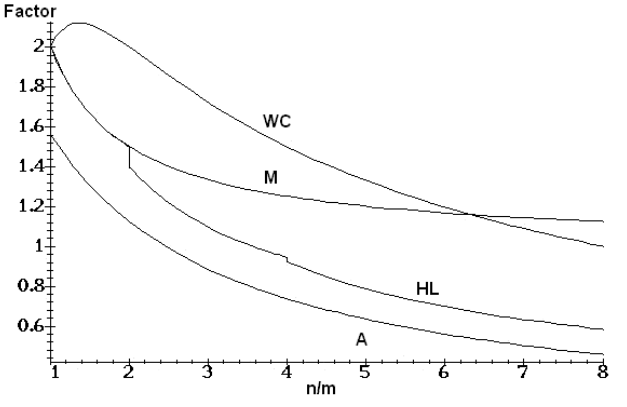


**Fig. 1.** Constant factor on $n$ depending on the ratio $\alpha = n/m$.

Let us consider now the average case. We use two assumptions: first, that we never find the median of $Q$ and hence we assume that some elements never appear in $D$; and second, that the median will divide $D$ in sets of size $i$ and $n-i$ with the same probability for all $i$ (this is equivalent to consider every element on $D$ as random, like in the average case analysis of Quicksort). The first assumption is pessimistic, while the second considers that overlaps are uniformly distributed, which is also pessimistic regarding our practical motivation as we do not take in account that word occurrences may and will have locality of reference. Figure 2 shows the actual number of comparisons for $n = 128$ and all powers of 2 for $m \leq n$, for all the cases already mentioned.

If $A(m, n)$ denotes the average case number of comparisons, for $m$ of the form $2^k - 1$ we have

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{1}{n+1} \sum_{i=0}^{n} (A((m-1)/2, i) + A((m-1)/2, n-i))$$

We now show that

$$A(m,n) = (m+1)(\ln((n+1)/(m+1)) + 3 - 1/\ln(2)) + O(\lg n)$$

The recurrence equation can be simplified to

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{n} A((m-1)/2, i)$$

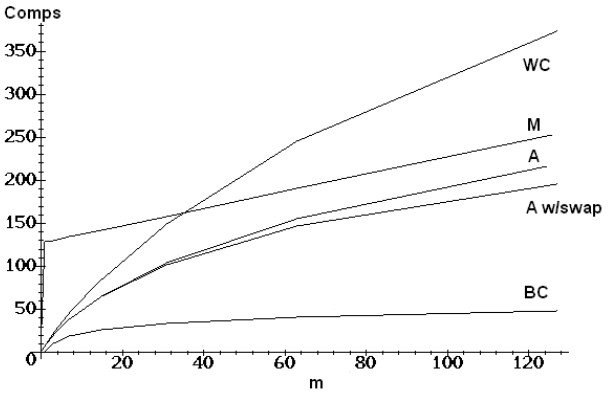


**Fig. 2.** Number of comparisons in the best, worst and average case (with and without swaps) for $n = 128$, as well as for merging (M).

As the algorithm is adaptive on the size of the lists, we have

$$A(m,n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{(m-1)/2} A(i,(m-1)/2) + \frac{2}{n+1} \sum_{(m+1)/2}^{n} A((m-1)/2,i)$$

by noticing that we switch the sets when $m > n$. However, solving this version of the recurrence is too hard, so we do not use this improvement. Nevertheless, this does not affect the main order term. Notice that our analysis allows any value for $n$.

Making the change of variable $m = 2^k - 1$ and using $k$ as subindex we get

$$A_k(n) = \lceil \lg(n+1) \rceil + \frac{2}{n+1} \sum_{i=0}^{n} A_{k-1}(i)$$

Eliminating the sum, we obtain

$$(n+1)A_k(n) = nA_k(n-1) + 2A_{k-1}(n) + \lceil \lg(n+1) \rceil + n\delta(n = 2^j)$$

where $\delta(n = 2^j)$ is 1 if $n$ is a power of 2, or 0 otherwise. Let $T_n(z) = \sum_k A_k(n) z^k$ be the generating function of $A$ in the variable $k$. Hence

$$T_n(z) = \frac{n}{n+1-2z} T_{n-1}(z) + \frac{\lceil \lg(n+1) \rceil + n\delta(n = 2^j)}{(n+1-2z)(1-z)}$$

Unwinding the recurrence in the subindex of the generating function, as $T_0(z) = 0$, we get

$$T_n(z) = \frac{n!}{(1-z)\Gamma(n+2-2z)} \sum_{i=1}^{n} \frac{\Gamma(i+1-2z)}{i!} (\lceil \lg(i+1) \rceil + i\delta(i = 2^j))$$

where $\Gamma(x)$ is the Gamma function (if $x$ is a positive integer, then $\Gamma(x) = (x-1)!$). Then, $A(2^k - 1, n) = [z^k]T_n(z)$ where $[z^k]$ is the coefficient of $z^k$. Expanding the terms in $z$ using partial fractions and extracting the coefficient, with the help of the Maple symbolic algebra system, we obtain the main order terms sought.

For $n = \alpha m$, the ratio between this algorithm and merging is $(\ln(\alpha) + 3 - 1/\ln(2))/(1+\alpha)$ which is at most 0.7913 when $\alpha = 1.2637$ and 0.7787 when $\alpha = 1$. This is also shown in figure 1, where we also include the average case analysis of Hwang and Lin's algorithm [13]. Recall that this analysis uses different assumptions, however shows the same behavior, improving over merging when $\alpha \geq 2$.

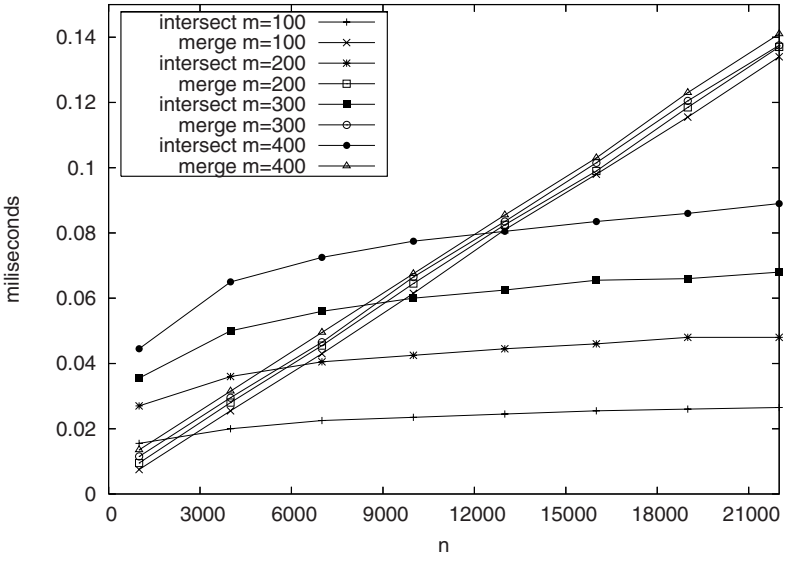


**Fig. 3.** Experimental results for the new algorithm and merging for several values of $m$ and $n$.

Figure 3 shows experimental results for four values of $m$ and varying $n$, for the new algorithm (intersect) compared to merge. We use twenty random instances per case and ten thousand runs to be able to measure the running times. We use uniformly random integer numbers in the range 1 to $10^9$ and we implemented the programs using the Gcc 3.3.3 compiler in a Linux platform running an Intel Xeon of 3GHz. We can see that the new algorithm becomes better than merging for larger $n$.

A simple way to improve this algorithm is to start comparing the smallest elements of both sets with the largest elements in both sets. If both sets do not overlap, we use just $O(1)$ time. Otherwise, we search the smallest and largest element of $D$ in $Q$, to find the overlap, using just $O(\lg m)$ time. Then we apply the previous algorithm just to the subsets that actually overlaps. This improves

both, the worst and the average case. The dual case is also valid, but then finding the overlap is $O(\lg n)$, which is not good for small $m$.

## 4 Application to Query Processing in Inverted Indices

Inverted indices are used in most text retrieval systems [3]. Logically, they are a vocabulary (set of unique words found in the text) and a list of references per word to its occurrences (typically a document identifier and a list of word positions in each document). In simple systems (Boolean model), the lists are sorted by document identifier, and there is no ranking (that is, there is no notion of relevance of a document). In that setting, our basic algorithm applies directly to compute Boolean operations on document identifiers: union is equivalent to merging, intersection is the complement operation (we only keep the repeated elements), and subtraction implies deleting the repeated elements. In practice, long lists are not stored sequentially, but in blocks. Nevertheless, these blocks are large, and the set operations can be performed in a block-by-block basis.

In complex systems ranking is used. Ranking is typically based in word statistics (number of word occurrences per document and the inverse of the number of documents having it). Both values can be precomputed and the reference lists are then stored by decreasing intra-document word frequency order to have first the most relevant documents. Lists are then processed by decreasing inverse extra-document word frequency order (that is, we process the shorter lists first), to obtain first the most relevant documents. However, in this case we cannot always have a document identifier mapping such that lists are sorted by that order.

The previous scheme was used initially on the Web, but as the Web grew, the ranking deteriorated because word statistics do not always represent the content and quality of a Web page and also can be "spammed" by repeating and adding (almost) invisible words. In 1998, Page and Brin [7] described a search engine (which was the starting point of Google) that used links to rate the quality of a page. This is called a global ranking based in popularity, and is independent of the query posed. It is out of the scope of this paper to explain Pagerank, but it models a random Web surfer and the ranking of a page is the probability of the Web surfer visiting it. This probability induces a total order that can be used as document identifier. Hence, in a pure link based search engine we can use our intersection algorithm as before. However, nowadays hybrid ranking schemes that combine link and word evidence are used. In spite of this, a link based mapping still gives good results as approximates well the true ranking (which can be corrected while is computed).

Another important type of query is sentence search. In this case we use the word position to know if a word follows or precedes a word. Hence, as usually sentences are small, after we find the Web pages that have all of them, we can process the first two words to find adjacent pairs and then those with the third word and so on. This is like to compute a particular intersection where instead of finding repeated elements we try to find correlative elements ($i$ and $i + 1$),

and therefore we can use again our algorithm as word positions are sorted. The same is true for proximity search. In this case, we can have a range $k$ of possible valid positions (that is $i \pm k$) or to use a different ranking weight depending on the proximity.

Finally, in the context of the Web, our algorithm is in practice much faster because the uniform distribution assumption is pessimistic. In the Web, the distribution of word occurrences is quite biased. The same is true with query frequencies. Both distributions follow a power law (a generalized Zipf distribution) [3, 5]. However, the correlation of both distributions is very small [4]. That implies that the average length of the lists involved in the query are not that biased. That means that the average lengths of the lists, $n$ and $m$, when sampled, will satisfy $n = \Theta(m)$ (uniform), rather than $n = m + O(1)$ (power law). Nevertheless, in both cases our algorithm makes an improvement.

## 5 Concluding Remarks

We have presented a simple set intersection algorithm that performs quite well in average and does not inspect all the elements involved. It can be seen as a natural hybrid of binary search and merging. We are currently studying how this algorithm behaves with other word occurrence distributions, for example, a Zipf distribution.

In practice, queries are short (on average 2 to 3 words [5]) so there is almost no need to do multiset intersection and if so, they can be easily handled by pairing the smaller sets firsts, which seems to be the most used algorithm [9]. In addition, we do not need to compute the complete result, as most people only look at less than two result pages [5]. Moreover, computing the complete result is too costly if one or more words occur several millions of times as happens in the Web and that is why most search engines use an intersection query as default. Hence, lazy evaluation strategies are used. If we use the straight classical merging algorithm, this naturally obtains first the most relevant Web pages. For our algorithm, it is not so simple, because although we have to process first the left side of the recursive problem, the Web pages obtained do not necessarily appear in the correct order. A simple solution is to process the smaller set from left to right doing binary search in the larger set. However this variant is efficient only for small $m$, achieving a complexity of $O(m \lg n)$ comparisons. An optimistic variant can use a prediction on the number of pages in the result and use an intermediate adaptive scheme that divides the smaller sets in non-symmetric parts with a bias to the left side. We are currently working on this problem for multiple sets.

## Acknowledgements

We thank Phil Bradford, Joe Culberson, and Greg Rawlins for many useful discussions on this and similar problems a long time ago. We also thank Alejandro Salinger for his help in the experimental results.

## References


1. R.A. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.
2. Ricardo Baeza-Yates, Phillip G. Bradford, Joseph C. Culberson, and Gregory J. E. Rawlins. The Complexity of Multiple Searching, unpublished manuscript, 1993.
3. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, ACM Press/Addison-Wesley, England, 513 pages, 1999.
4. R. Baeza-Yates, and Felipe Saint-Jean. A Three Level Search Engine Index based in Query Log Distribution. SPIRE 2003, Springer LNCS, Manaus, Brazil, October 2003.
5. Ricardo Baeza-Yates. Query Usage Mining in Search Engines. In Web Mining: Applications and Techniques, Anthony Scime, editor. Idea Group, 2004.
6. Jérémy Barbay and Claire Kenyon. Adaptive Intersection and $t$-Threshold Problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 390–399, San Francisco, CA, January 2002.
7. S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *7th WWW Conference*, Brisbane, Australia, April 1998.
8. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro, Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 743–752, San Francisco, CA, January 2000.
9. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro, Experiments on Adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments*, LNCS, Springer, Washington, DC, January 2001.
10. Dietz, Paul, Mehlhorn, Kurt, Raman, Rajeev, and Uhrig, Christian; "Lower Bounds for Set Intersection Queries," *Proceedings of the* $4^{th}$ *Annual Symposium on Discrete Algorithms,* 194–201, 1993.
11. Dobkin, David and Lipton, Richard; "On the Complexity of Computations Under Varying Sets of Primitives," *Journal of Computer and Systems Sciences,* **18**, 86–91, 1979.
12. W. Fernandez de la Vega, S. Kannan, and M. Santha. Two probabilistic results on merging, *SIAM J. on Computing* 22(2), pp. 261–271, 1993.
13. W. Fernandez de la Vega, A.M. Frieze, and M. Santha. Average case analysis of the merging algorithm of Hwang and Lin. *Algorithmica* 22 (4), pp. 483–489, 1998.
14. F.K. Hwang and S. Lin. A Simple algorithm for merging two disjoint linearly ordered lists, *SIAM J. on Computing* 1, pp. 31–39, 1972.
15. Rawlins, Gregory J. E.; *Compared to What?: An Introduction the the Analysis of Algorithms,* Computer Science Press/W. H. Freeman, 1992.