Seminar report

Set Intersection Problem

Marco Bellò

April 16, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	

Tekijä — Författare — Author

Marco Bellò

Työn nimi — Arbetets titel — Title

Set Intersection Problem

Ohjaajat — Handledare — Supervisors

Prof. J. Kärkkäinen, Prof. V. Mäkinen, Dr. D. Diaz

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminar report	April 16, 2025	18 pages, 1 appendix pages

Tiivistelmä — Referat — Abstract

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Software and its engineering → Software creation and management → Designing software
Applied computing → Document management and text processing → Document searching

Avainsanat — Nyckelord — Keywords

algorithms, data structures, set intersection, inverted index

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Seminar in Algorithm Engineering

Contents

1 Introduction

The act of *searching* has become so deeply ingrained in the modern society that we tend to take it for granted, not only assuming it normal to have immediate and easily accessible information on the tip of our thumbs, but expecting it: a study from 2004 showed that users were not willing to wait more than ten seconds for a page to load (Nah, 2004). Fast forward twenty years and nowadays even a couple seconds holdup would be unacceptable, thus query retrieval needs to be fast. Blazingly fast in fact, since we need to account for all the delays typical of a gargantuan structure as big as the modern web, and, as the reader probably knows, it is *not* a good idea to rely on memory's performances increasing over time: the smart way to tackle this problem is via research and development of efficient algorithms, and exactly which type should be self-evident from the title of this document. The problem of the set intersection constitutes the backbone of every query resolver in a (web) search engine, since every word in a query is interpreted as a collection of documents' IDs which contains it.
In this survey-style paper we will first explain what searching (i.e., querying) entails, show how a document (e.g., a web page) can be transformed into word tokens which are then further processed into inverted indexes, and, finally, we will see a collection of algorithms that concern themselves with intersecting sets, meaning finding common elements between two or more comparable collections.

1.1 How Do We Search?

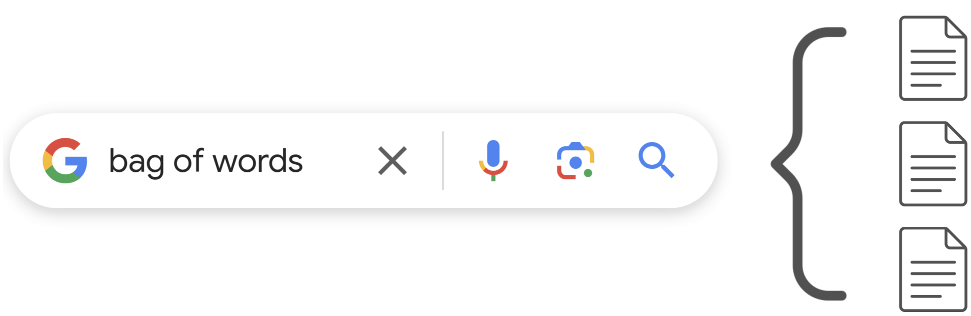

Figure 1.1: From a bag of words to a set of documents

Generally speaking, a query is called a *bag of words*, and finding its result means computing which documents contain all word tokens that are being searched for [1.1]. Let's make and example: word `abiura` is contained in documents number `[31, 42, 127]`, while word `bitonto` is contained in documents number `[20, 42, 72]`.
Thus `query = (abiura,bitonto)` will return the result `42`.

Dictionary	Posting List (ASC)	Relevance
abaco	1, 7, 136	0.6, 0.3, 0.8
abiura	31, 42, 127	0.12, 0.5, 0.77
bitonto	20, 42, 72	0.8, 0.1, 0.03

Figure 1.2: Table of word tokens

Both The example above [1.2] and all the algorithms we will see in this survey consider the problem of searching as the problem of complete intersection, but modern search engine (e.g., Google) leverage input relevance and filter unneeded outputs to obtain faster and better results. Unfortunately finding information about how they do it is near impossible, since everything is covered by trade secret.
Let's now see what inverted indexes are and how we can obtain them starting from a document corpus.

2 Inverted Indexes

Most of the information present in this chapter is thanks to Mahapatra and Biswas "Inverted indexes: Types and techniques" (Mahapatra and Biswas, 2011).

What we will need for the algorithms presented in the rest of this documents are inverted indexes (also called posting lists). To get them we first need to process documents into lists of words (called *word tokens*), then for each token compute a list of `IDs` that refer to the documents which contain that specific token. Let's see each step in order.

2.1 Document Pre-Processing

Documents go through a series of processing steps before being indexed: they get converted into token in the lexing phase, which are then possibly normalized, stemmed or even pruned (removed) entirely.

2.1.1 Lexing

The process of transforming a document into a list of tokens, each of which is a single word, si called *lexing* [2.1]. There often is a maximum length for a single token, as to prevent unbounded index growth in edge cases, and all input is generally first converted into lower-case to normalize it. All non-punctuation characters are added to the list of tokens one by one, and those that exceed a certain size are often pruned (removed from the corpus). It is not entirely clear how Google and other big companies do this step, and it certainly feels strange to think they employ a simple *brute force*, single scan approach, but as mentioned be-

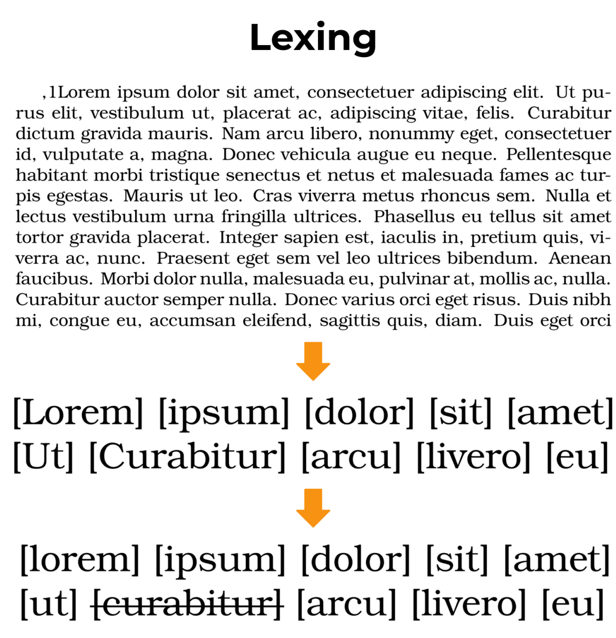

Figure 2.1: Lexing: from text to word tokens

fore it is not easy to find information about
it.
All of the above works only with alphabetic languages, ideographic ones (e.g., Chinese) need specialized search techniques.

2.1.2 Stemming

We can consider this step deprecated, since nowadays memory, especially for things like text and arrays (which inverted index basically are), is cheap and bountiful.
The idea is to find a sort of *root* (stem) of the words, and indexing that instead. To make an example: fishnet, fishery, fishing, fishy, fishmonger, can all be boiled down to their stem *fish* [2.2].

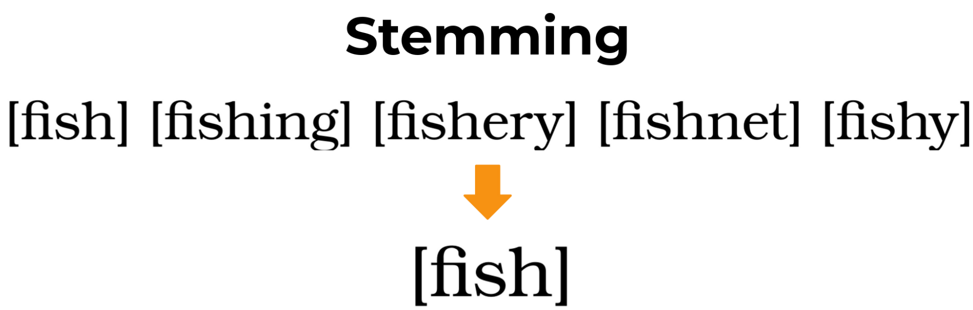

Figure 2.2: Stemming to stem "fish"

In the example above should be clear already that stemming carry some problems: a user searching for "fishnet" is likely not shopping for fishing equipment, thus most modern search engine skip this normalizing step, and most stemming algorithms (most famous of which is Porter's) are complex, full of exceptions and exceptions to the exceptions, while still failing to unite together the correct words. This step basically reduces query precision while providing very little in return.

2.1.3 Stop Words

Stop words are words that work as connectives of sorts, like *and*, *the*, *is*, *of*, *to*, etc.
Their quantity is language dependent (e.g., in English they could be around 500 words) and they are often removed from the corpus which, for normal queries, does not worsen the results while saving space in the index. However in some cases like searching for *to be or not to be* stop words are actually essential, and removing them would make the search

fail.

Thankfully they are so common that if saved as differences between consecutive different values, both their document number and word position lists can be compressed to save space. Because of this, the overhead is not as big as one might think, thus modern search engines (like Google) do not seem to remove them from the index, since doing so put them at a competitive advantage at the expense of a slightly bigger index.

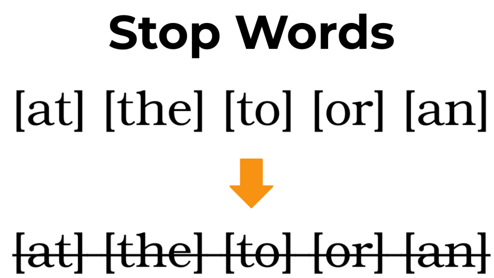

Figure 2.3: Stop words pruning

2.2 Inverted Indexes

Now that we have a set of word tokens, we can start building our inverted indexes (or posting lists): documents are often stored as lists of words, but we invert (hence the name) this concept by storing for each word the list of documents that contain it. There are several variants of this data structure, but at minimum you need to store for each word the list of documents that contain that specific word.

We can change the granularity by adding the frequency of the word in the document, which can be useful for query optimization, or by adding the word position in the document, allowing for in-document queries.

Space used by inverted indexes varies wildly in the range of five to one hundred percent (5-100%) of the total size of the document indexed, and this is because implementations come in many different variations: some store word positions and some don't, some aggressively pre-process documents and some don't, some dynamically update themselves and some don't, some use complex and powerful compression methods and some don't, and so on.

Table [2.1] show some sample documents, while table [2.2] shows some examples of inverted indexes, with different levels of granularity.

ID	Contents
1	The only way not to think about money is to have a great deal of it
2	When I was young I thought that money was the most important thing in life; now that I am old I know that it is.
3	A man is usually more careful of money than he is of his principles.

Table 2.1: Sample document collection

Word	Doc List	Frequency	Positions
a	1, 3	1:1, 3:1	1:(12), 3:(1)
About	1	1:1	1:(7)
am	2	2:1	2:(19)
Careful	3	3:1	3:(6)
deal	1	1:1	1:(16)
great	1	1:1	1:(13)
have	1	1:1	1:(11)
...	...	...	...
money	1, 2, 3	1:2, 2:1, 3:1	1:(8), 2:(8), 3:(9)
more	3	3:1	3:(5)
...	...	...	...
when	2	2:1	2:(1)

Table 2.2: Inverted lists example, most words omitted

3 Intersection Algorithms

In this chapter we are going to see a collection of algorithms to compute the intersection of two **sorted** lists, taken from the chapter six of "Pearl of Algorithm Engineering" by Paolo Ferragina, published by Cambridge University Press (Ferragina, 2023).
We will first look at two of the most commonly used search algorithms, since we cannot intersect without searching.

3.1 Search Algorithms

3.1.1 Binary Search

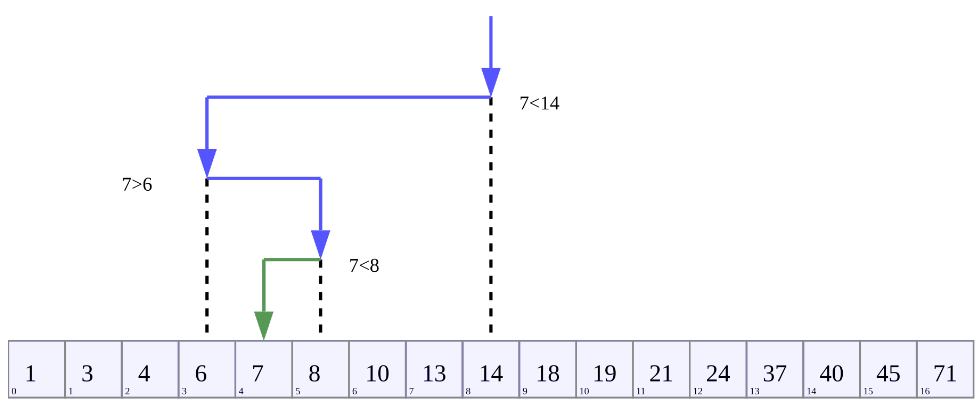

Figure 3.1: Binary search algorithm, source: Wikipedia

Binary search, also known as logarithmic search or binary chop, is a search algorithm that finds the position of a target value within a sorted array: it compares the target to the middle element of the array and, if they are not equal, it eliminates half of the search space by discarding either the left or right half, depending on whether the target value is less than or greater than the middle element. This process is repeated by iteratively searching into the remaining sub-array until the target value is found or the search space is empty. The pseudocode for the algorithm can be seen at *Algorothm* [1].

Binary search runs in logarithmic time in the worst case, doing $O(\log n)$ comparisons, where n is the number of elements in the array, making it much faster than linear search with large arrays thanks to its scaling.

Algorithm 1

Pseudocode for binary search algorithm

looking for element key
let $L = 0$ ▷ First half
let $R = n - 1$ ▷ Second half
while $L \leq R$ **do**
  $m = \lfloor (L + R)/2 \rfloor$
  if $A[M] < key$ **then**
    let $L = m + 1$
  else if $A[M] > key$ **then**
    $R = m - 1$
  else
    return m ▷ Found
  end if
end while
return false ▷ Not found

3.1.2 Exponential Search

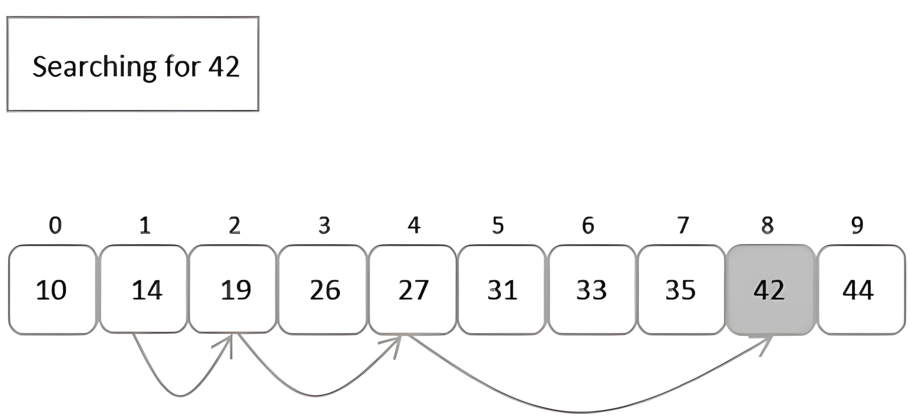

Figure 3.2: Exponential search algorithm, source: Tutorialspoint

Exponential search, also called doubling or galloping search, is an algorithm for searching sorted, unbounded lists: there are numerous implementations, most common being determining a sub-array into which the `key` may resides in and performing a binary search [3.1.1] within its range.
To be more precise: we go trough the list in exponentially increasing steps, with a factor of 2^k such that we first look into `list[0]`, then `list[1]`, then `list[2]`, `list[4]`, `list[8]`, following with *16*, *32*, *64*, *128* and so on until we find a value that is greater than the `key`. Once we find it, we perform a binary search between the previous jump and the current (or the end of the array): $2^{k-1} \leq key \leq min(2^k, n)$.

The algorithm can be more efficient than binary search, as it runs in $O(\log i)$ time, where i is the index of the element being searched for, which could be half if not less than n. The pseudocode can be seen at *Algorithm* [2].

Algorithm 2

Pseudocode for exponential search algorithm

looking for element key
let $i = 0$
let $k = 0$
while $(key > list[i + 2^k]$ and $i < n)$ **do**
    $i = i + 2^k$ ▷ Gallop to next step
    $k = k + 1$ ▷ Increment exponent
end while
if $i < n$ **then**
    binary_search($list$, key, i, $min(i + 2^k, n)$)
else
    return false ▷ Not found
end if

3.1.3 Extrapolation and Intrapolation

TODO maye or maybe not

3.2 Intersection Algorithms

3.2.1 Brute Force

The first idea that would come to mind when thinking about intersecting two lists is to simply iterate through both of them and check if the elements are equal: this is the *brute force* approach, which is simple but inefficient.
With a time complexity of $O(m \cdot n)$, assuming lists sizes n and m to be around 10^6, and assuming a modern computer able to do 10^9 operations per second, this algorithm would need ten minutes to compute a 2-words query, which is less than ideal.
The (very short) pseudocode can be seen at *Algorithm* [3].

Algorithm 3

Pseudocode for brute force algorithm

```
for all i = 0 to n − 1 do
    for all j = 0 to m − 1 do
        if A[i] == B[j] then
            add A[i] to result
        end if
    end for
end for
```

3.2.2 Bunny Race

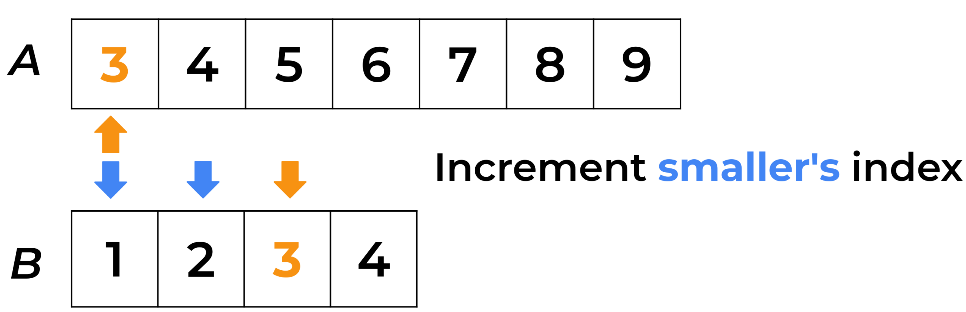

Figure 3.3: Bunny race algorithm

This approach, often called merge-based, is simple, elegant and fast: the main idea is to have two indices pointing at the two list running after each other by comparing elements each time and incrementing the index pointing at the smallest one (or incrementing both if they are equal).

To clarify: lets say we have two lists, A and B, of size n and m respectively. We start with two pointers, i and j, both set to zero.

We compare the elements at these indices, $A[i]$ and $B[j]$: if they are equal, we add the element to the result and increment both pointers.

If $A[i] < B[j]$, we increment i, while if $A[i] > B[j]$, we increment j.

This process continues until one of the pointers reaches the end of its respective list.

The correctness can be proven inductively, exploiting the following observation: if $A[i] < B[j]$ then $A[i]$ is smaller than all elements following $B[j]$ in B since its ordered, so $A[i] \notin B$. The other case is symmetric.

In regards to time complexity, we just need to note that at each step the algorithm executes one comparison and advances at least one iterator, thus, given that $n = |A|$ and $m = |B|$, the algorithm runs in no more than $O(n + m)$ time.

This time complexity is significantly better than the *brute force* [3] approach, since it can compute a 2-word query in 10^{-3} seconds.

The pseudocode can be seen at *Algorithm* [4].

In the case that $n = \Theta(m)$ this algorithm is optimal, because we need to process the smallest set, thus $\Omega(min(n, m))$ is an obvious lower bound. Moreover, this procedure is also optimal in the disk model since it takes $O(\frac{n}{B})$ I/Os.

In the case that $n \ll m$ the classic *binary search* can be helpful since we can design an algorithm that search in A for each elements of B in $O(m \log n)$ time, which is better than $O(n + m)$ when $m = o(\frac{n}{\log n})$.

Algorithm 4

Pseudocode for bunny race algorithm

let $i = 0$
let $j = 0$
while $i < n$ and $j < m$ **do**
    if $A[i] < B[j]$ **then**
        $i = i + 1$ ▷ Increment first
    else if $A[i] > B[j]$ **then**
        $j = j + 1$ ▷ Increment second
    else
        add $A[i]$ to result ▷ Found
        $i = i + 1$ ▷ Increment both
        $j = j + 1$
    end if
end while

3.2.3 Divide and Search

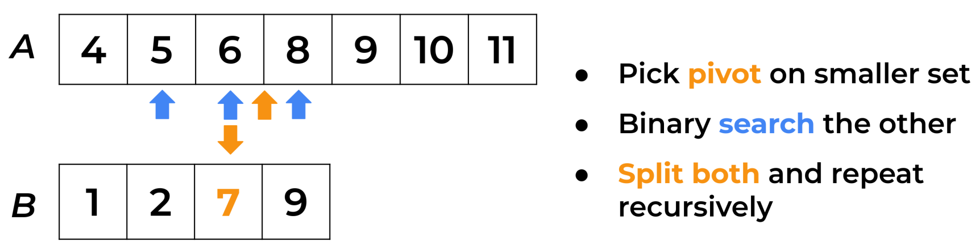

Figure 3.4: Divide and search algorithm

Also called *mutual partitioning*, this approach adopt a classic algorithmic paradigm, namely *divide and conquer*, famously used to design the *quick sort* algorithm (which can be visualized here).

Let us assume $m = |B| \leq n = |A|$. We select the median element of B, $b_{m/2}$, as a *pivot* and search for it in the longer sequence A using the *binary search* [3.1.1] algorithm.

Two cases may occur:

i. *pivot* is one of the elements fo the intersection;

ii. $b_{m/2} \notin A$, e.g, $A[j] < b_{m/2} < A[j + 1]$.

In both cases the algorithm proceeds *recursively* by calling itself on the two sub-lists in which each list (A and B) has been split according to the *pivot* element, thus computing the following intersections:

- $A[1, j] \cap B[1, m/2 - 1]$
- $A[j + 1, n] \cap B[m/2 + 1, m]$.

In simpler terms: we pick the middle element of the smaller list, search for it in the bigger list, split both lists and repeat the process recursively on the remaining sub-lists, then again, then again, then again, until we reach the base case where each list is of size one. The pseudocode of the algorithm can be seen at *Algorithm* [5].

Correctness follows, while for evaluating time complexity we need to identify the worst case.
Let us start with the case where *pivot* falls outside A, meaning that one of the two parts is empty and thus the corresponding half of B can be ignored. So, one *binary search* [3.1.1] over A, costing $O(\log n)$ time, has discarded half of B.
If this keeps occurring in all recursive calls, the total number of them will be $O(\log m)$, which leads us to a time complexity of $O(\log m \log n)$.
On the other hand, if we have both balanced partitions so that $b_{m/2}$ not only falls inside A but coincides with the median element $a_{n/2}$, the time complexity can be expressed via the recurrence relation $T(n, m) = O(\log n) + 2T(\frac{n}{2}, \frac{m}{2})$, with the base case $T(n, m) = O(1)$ whenever $n, m \leq 1$.
This recurrence has the solution $T(n, m) = O(m(1 + \log \frac{n}{m}))\forall m \leq n$, which is an optimal time complexity in the comparison model.

That being said, despite its optimal time complexity, the mutual-partitioning paradigm is heavily based on recursive calls and binary searching, and both paradigms offer poor performance in a disk-based setting when sequences are long hence requiring a large number of both dynamic memory allocations (recursive calls) and random memory access (*binary search* steps).

Algorithm 5

Pseudocode for divide and search algorithm

Let $m = |B| \leq n = |A|$
Pick *pivot* $p = b_{\lfloor m/2 \rfloor}$
Binary search for p in A ▷ Say $a_j \leq p < a_{j+1}$
Divide and search on $A[1, j] \cap B[1, m/2 - 1]$
if $p = a_j$ **then**
    Add p to result
end if
Divide and search on $A[j + 1, n] \cap B[m/2 + 1, m]$

3.2.4 Doubling Search

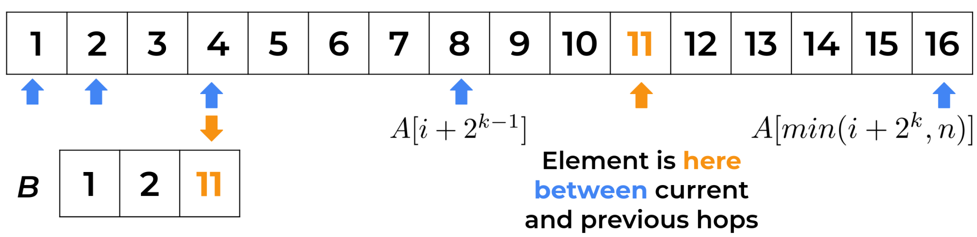

Figure 3.5: Doubling search algorithm

Also called *exponential search* or *galloping search*, this paradigm is more or less what we presented in the search algorithm *exponential search* [3.1.2]: assuming $m = |B| \leq n = |A|$, we search of each element b_j of B into A by jumping trough it with exponentially bigger steps that increase by a factor of 2^k, meaning that we compare b_j with $A[0]$, $A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, $A[32]$, and so on, until we find that either $b_j < A[i + 2^k]$ for some k, or we have jumped out of the array since $i + 2^k > n$.
Then we perform a *binary search* [3.1.1] for b_j between $A[i + 2^{k-1}]$ and $A[min(i + 2^k, n)]$. $A[0, i + 2^{k-1}]$ will be discarded from the subsequent searches.
The pseudocode of the algorithm can be seen at *Algorithm* [6], which is a bit different from the one we saw in *exponential search* section [3.1.2], so it can be seen as an extra resource. Both works very similarly.

Correctness is again immediate, while deriving time complexity will require some reasoning: we denote with Δ_i the size of the sub-array of A where b_j could be found. We then say that:

- $b_j \geq i + 2^{k-1}$ i.e. previous step
- $b_j < min(i + 2^k, n)$ i.e. current step or end of A

We can therefore write $2^{k-1} \leq i - (i-1)$ where i and *(i-1)* are current and previous step respectively, and combining this inequality with Δ_i we get $\Delta_i \leq 2^{k-1} \leq i - (i-1)$.
At this point we can estimate the total lengt of search sub-arrays of A: $\sum_{i=1}^{m} \Delta_i \leq \sum_{i=1}^{m}(i-(i-1)) \leq n$, because the latter is a telescopic sum in wich consecutive terms cancel out.
For every i, the algorithm executes $O(1 + \log \Delta_i)$ steps, thus summing for $i = 1, 2, ..., m$ (since $m = |B|$) we get a total time complexity of $\sum_{i=1}^{m} O(1 + \log \Delta_i) = O(\sum_{i=1}^{m}(1 + log\Delta_i)) = O(m + m \log \sum_{i=1}^{m} \frac{\Delta_i}{m}) = O(m(1 + \log \frac{n}{m}))$.
This is the same time complexity we got for the *divide and search* [5] algorithm, but this present algorithm paradigm is iterative, thus does not require dynamic memory allocation. Moreover, it calls the *binary search* [3.1.1] on a sub-array of A needing less disk accesses.

Algorithm 6

Pseudocode for doubling search algorithm

Let $m = |B| \leq n = |A|$
Let $i = 0$
for all $j = 0$ to $m - 1$ **do**
    Let $k = 0$
    while $B[j] > A[i + 2^k]$ and $i + 2^k \leq n$ **do**
        $k = k + 1$ ▷ Increment exponent
    end while
    $i' =$ *binary search* into $A[i + 2^{k-1} + 1, min(i + 2^k), n]$
    if $a_{i'} = b_j$ **then**
        Add b_j to result
    end if
    $i = i'$ ▷ Update i to the last position
end for

4 Discussion

5 Conclusions

Bibliography

Ferragina, P. (2023). *Set Intersection*. Cambridge University Press, pp. 72–81.

Mahapatra, A. K. and Biswas, S. (July 2011). “Inverted indexes: Types and techniques”. In: *International Journal of Computer Science Issues*, 8.

Nah, F. (Jan. 2004). “A study on tolerable waiting time: how long are Web users willing to wait? Citation: Nah, F. (2004), A study on tolerable waiting time: how long are Web users willing to wait? Behaviour & Information Technology, forthcoming”. In: *Behaviour & IT*, 23.

Appendix A Sample Appendix

You can add one or more appendices to your thesis.