Milan's Multi-key Binary Search algorithm and the Related Performance

Július Milan

Red Hat Czech,
Purkynova 115, 621 00 Brno-Medlanky, Czech Republic
jmilan@redhat.com
http://www.redhat.com

Abstract. Searching for a one key over a sorted array of values is usually done by binary search algorithm, well known for its efficiency which in terms of time complexity is logarithmic. The concept of multi key binary search was already introduced several years ago and wide variety of algorithms implementing it was found as shown later. This paper proposes a new elementary search algorithm dedicated for search of multiple keys over a sorted array. The algorithm is based on classical one-key binary search which is used in recursive manner for searching always for the middle key of given (sub)array of keys on given (sub)array of values, taking advantage of the knowledge obtained from its previous results to adjust the boundaries of each subsequent binary searches and so lower their search space. Time complexity of the algorithm is a function of two variables and is logarithmic considering both of them.

Keywords: Multiple keys, Binary Search, Multi-key Binary Search, Recursive Algorithm, Logarithmic Time Complexity

1 Introduction

Computer science research of elementary algorithms as binary search and its variants comes from their wide applicability often also out of scope of computer science itself and potential to significantly improve performance of computation.

To search for multiple keys inside sorted array of values, it is possible to use the classical binary search on all keys one by one with the same result with time complexity $O(M * log(N))$ where M is the amount of keys to search and N is the size of an array to search on. The algorithm proposed in this paper outperforms this sequential approach significantly as shown by measurements and outperforms also other yet known algorithms according to study of author.

The idea of searching for more than one key at a time in a sorted array of values was introduced in 2004 [1]. Later, the same author came with next variations [2] [3], the algorithms improves their performance by reducing search space, however the art is still more theoretical and non implementable in common programming languages (as C, Java, ...) or implementations are significantly limited. The last variation [4] has practical implementation for search of any number of keys lower than size of array of values. Variation [4] reduces its search space by firstly searching for particular amount of keys uniformly distributed over array. Then it continues to search intermediate keys sequentially, but just in intervals determined by first set of searches.

The Milan's Multi-Key Binary Search (further just `mmkbs`) improves its performance by reducing search space as well, but uses a different order in which the keys are searched. Firstly it searches for a middle key (at 1/2 of keys input), then again for middle keys of both halves (1/4 and 3/4 of keys input), then for middle keys of all fourths (1/8, 3/8, 5/8, 7/8) and so on, reducing its search space always to a half for its subsequent searches in average case.

The paper is organized as follows. Section 2 introduces the terms used in the paper. Section 3 and its subsections describes the inputs and outputs of the algorithm, the idea behind the algorithm and its detailed explanation of its behaviour, example implementation and some of its specific features. Section 4 describes how the measurements were done and shows measurements results. Section 5 contains the time complexity of introduced `mmkbs` algorithm and the steps how it was found.

2 Terminology

`bs`	: Binary Search
`mmkbs`	: Milan's Multi-key Binary Search
`m_times_bs`	: sequential implementation to search for multiple keys
`MIDDLE`	: macro or function to determine middle of given interval
arr	: sorted array (or list) of elements to search on
N	: amount of elements in arr, i.e. size of array to search on
$keys$	: sorted array of keys to search on arr
M	: amount of keys to search, i.e. size of keys
$results$	: array representing search output

Variable names used in example implementation of Sect. 3.3, symbols from Figures or standard mathematical symbols might be used in text to help to explain the implementation and functionality of the algorithm.

3 The algorithm

3.1 Input and Output

Inputs of the algorithm are sorted array arr of values to search on and sorted array of keys $keys$ to search for. The example algorithm present in this paper requires both arr and $keys$ sorted in ascending order, however four slightly different variations of `mmkbs` are possible, next one is when both inputs (arr and $keys$) are sorted in descending order and next two variations when one of inputs is sorted in ascending and other in descending order. The limitation that $keys$ have to be sorted as well in practice may require some next effort before the search is performed. However that effort can be moved to non-critical sections or the whole solution which uses the algorithm can be designed to require minimal amount of sorting of values which could be possibly used for searching as keys.

As an output, algorithm writes its findings into array for *results* which is of size M, i.e. the same size as size of *keys*. Each result represents the position in *arr* where the particular key was found. The example implementation bellow doesn't store any information to *results* in case of finding nothing, next variation would be to write some default value in such cases.

3.2 Idea

The idea behind the algorithm is to take advantage of the fact that keys are ordered as well as values in *arr*. As a consequence that means that every key's k predecessor p (the key one to left) has also its corresponding value in *arr*, if any, located somewhere to the left from the value which corresponds to the k. The same principle applies for right side too. This is well visible on Fig. 1 below.

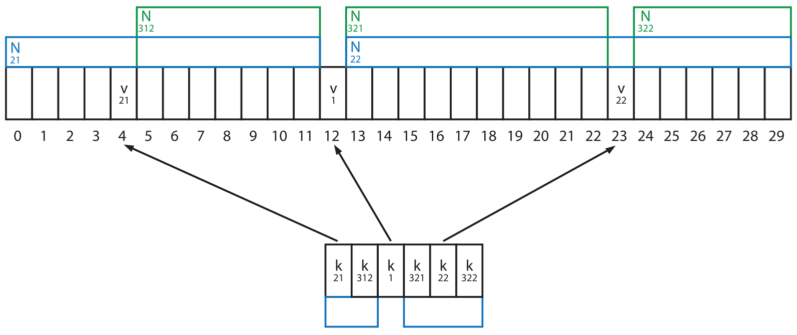

Fig. 1. Example search, shows ranges of next searches, if k_1 was found on position 12 of *arr*, k_{21}-s range to search is shown by upper left blue line and is of size N_{21}.

From Fig. 1 it is obvious that if the key $k1$ is found on the position $v1$ then lower keys (in this case $k21$ and $k312$) can be located on *arr* only to the left from $v1$. Algorithm can continue in the same way for both halves with updated boundaries. Left half boundaries becomes original left boundary (*arr_l* in example code) and $v1 - 1$ and new left half middle key is determined from keys $k21$, $k312$ (so $k21$), when C macro:

```
#define MIDDLE(left, right) (((left) + (right)) / 2)
```

or its optimized version:

```
#define MIDDLE(left, right) (((left) + (right)) >> 1)
```

is used for middle determination.

For right side it is analogical, its boundaries become $v1 + 1$ and original right boundary (*arr_r* in example code) and new right half middle key is determined from keys $k321$, $k22$, $k322$ (so $k22$). This is repeated recursively until list of keys obtained on input of recursive call is not empty.

3.3 Example implementation in C

Used binary search:

```c
int bs (const int *arr, int left, int right, int key,
         bool *found)
{
    int middle = MIDDLE(left, right);

    while (left <= right) {
        if (key < arr[middle])
            right = middle - 1;
        else if (key == arr[middle]) {
            *found = true;
            return middle;
        }
        else
            left = middle + 1;
        middle = MIDDLE(left, right);
    }

    *found = false;
    /* left points to the position of first bigger element */
    return left;
}
```

Above, there is a well known binary search. Only its interface is slightly specific.

If the element was found, the procedure passes *true* to its output parameter $found$ and returns the position of the found element.

If the element was not found, it passes *false* to the $found$ and returns the position of the smallest element bigger than the *key* found in array arr. This information is useful because according to it, it is possible to determine boundaries for next searches even if the element was not found.

Implementation of the mmkbs:

```
void _mmkbs (const int *arr, int arr_l, int arr_r,
             const int *keys, int keys_l, int keys_r,
             int *results)
{
    if (keys_r - keys_l < 0)
        return;

    int keys_middle = MIDDLE(keys_l, keys_r);

    /* throw away half of keys if the middle key is out */
    if (keys[keys_middle] < arr[arr_l]) {
        _mmkbs(arr, arr_l, arr_r,
               keys, keys_middle + 1, keys_r, results);
        return;
    }
    if (keys[keys_middle] > arr[arr_r]) {
        _mmkbs(arr, arr_l, arr_r,
               keys, keys_l, keys_middle - 1, results);
        return;
    }

    bool found;
    int pos = bs(arr, arr_l, arr_r, keys[keys_middle], &found);

    if (found)
        results[keys_middle] = pos;

    _mmkbs(arr, arr_l, pos - 1,
          keys, keys_l, keys_middle - 1, results);
    _mmkbs(arr, (found) ? pos + 1 : pos, arr_r,
          keys, keys_middle + 1, keys_r, results);
}

void mmkbs (const int *arr, int N,
            const int *keys, int M,
            int *results)
{   _mmkbs(arr, 0, N - 1, keys, 0, M - 1, results);   }
```

3.4 Specific feature to consider

In some cases, the example implementation of `mmkbs` can bring different results than multiple times called `bs`. The non standard case, which can cause different results occurs when *keys* array contains the duplicates, in that case `bs` called multiple times will always find the key and always on the same position since inputs are always the same, but `mmkbs` will either find the duplicate key on different position or won't find it at all.

Although in practical use cases it makes no sense to search for the same key more than once, it is easy to add to `mmkbs` functionality to always check neighbour keys to manage this difference if needed.

3.5 Throwing away redundant keys

The mechanism for throwing away outstanding keys, i.e. keys that are bigger (resp. smaller) than boundary values of *arr* is implemented on lines 30 to 37 of example code. Its purpose is to efficiently get rid of keys which are out of range of searched array *arr* or its sub-intervals. This ensures good performance and valid functionality also for cases when *keys* array is bigger then *arr* and/or contains values out of range of *arr*.

4 Measurements

Measurements were done on idle machine for big sizes of *arr* and *keys*. Values in *arr* and *keys* were generated by pseudo-random generator. The same inputs were processed by both `mmkbs` and `m_times_bs` and the execution time was compared.

Simple m_times_bs - multiple times called bs:

```
void m_times_bs (const int *arr, int N, const int *keys, int M,
                 int *results)
{
    for (int i = 0; i < M; i++)
    {
        bool found;
        int pos = bs(arr, 0, N - 1, keys[i], &found);
        if (found)
            results[i] = pos;
    }
}
```

In principle there are two approaches to do measurements in order to see meaningful results. One is with floating *arr* size, i.e. N and fixed *keys* size, i.e. M. Second is reversed.

4.1 Measurement 1 - N floating, M fixed

First set of measurements was done with M of fixed size 50 000 and N varying from 0 to 400 000 and its graph is on Fig. 2. With floating N, `mmkbs` ran whole time faster then `m_times_bs` about 2.5 times and the ratio of those two didn't change very much.

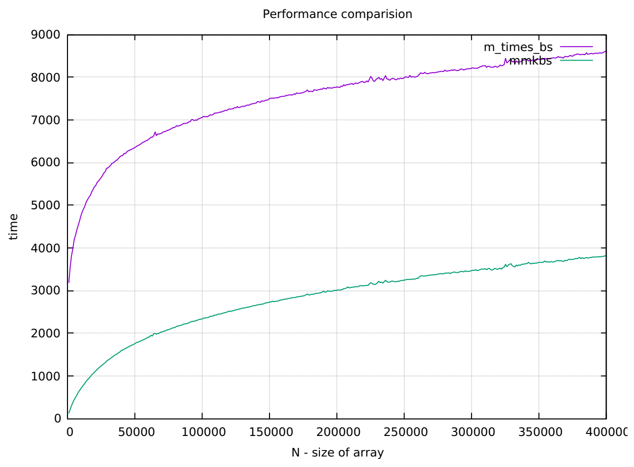

Fig. 2. Performance comparison with fixed M of size 50 000, x-axis represent variable N, y-axis represents time in CPU ticks.

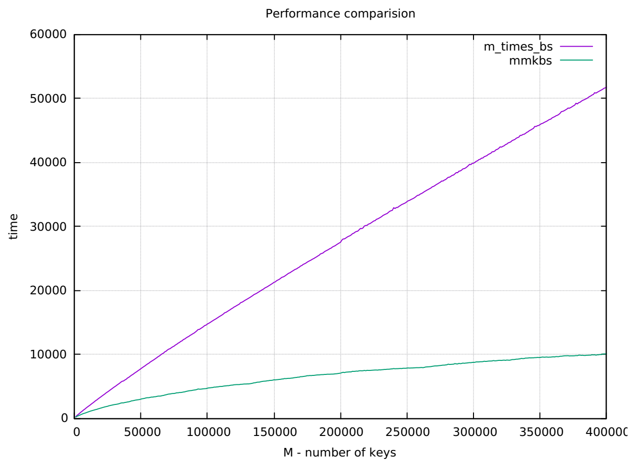

Fig. 3. Performance comparison with fixed N of size 200.000, x-axis represent variable M, y-axis represents time in CPU ticks.

4.2 Measurement 2 - N fixed, M floating

Second set of measurements was done with N of fixed size 200 000 and M varying from 0 to 400 000. On the Fig. 3 a big difference is visible, while `m_times_bs` time grows linearly due to change of linear factor in its quasilinear time complexity $O(M * log(N))$, `mmkbs` grows logarithmically.

Interesting is also the milestone when value of M exceeds 200 000, because N also equals 200 000. After this milestone, the $keys$ became bigger than arr, but performance of `mmkbs` doesn't suffer at all and grows further slowly in its logarithmic manner.

5 Time Complexity

As visible on Fig. 1, the ranges of consequent searches are divided by result of preceding search. To quantify the overall complexity, lets start from the first search. Search of the first key k_1 has time complexity of $O(log(N))$. Lets call this set of searches $level_1$ searches.

Next two keys k_{21} and k_{22} searches will operate on arr subintervals of size N_{21} and N_{22}. And since $N_{21} + N_{22} = N - 1$, for following purpose it is possible to estimate $N_{21} \approx N_{22} \approx N/2$. Then the time complexity of those next two searches could be expressed as $log(N/2) + log(N/2)$ which equals $2log(N/2)$. Lets call this set of searches $level_2$ searches.

The same could be applied for next potential foursome of keys k_{311}, k_{312}, k_{321} and k_{322}. Then their time complexity together could be expressed as $4log(N/4)$. Lets call this set of searches $level_3$ searches.

Overall time complexity of infinite amount of $keys$ has form

$$log(N) + 2log(N/2) + 4log(N/4) + 8log(N/8) + ... \tag{1}$$

Which equals

$$\sum_{n=0}^{\infty} 2^n * log(N/2^n) \tag{2}$$

Lastly it is needed to find upper bound of the sum (2), considering searched amount of $keys$. Since $level_1$ search involves 1 key, $level_2$ searches involve 2 keys, $level_3$ searches involve 4 keys, $level_4$ searches involve 8 keys and so on, its obvious that amount of keys involved in every next search doubles. Thereof number of $level$-s for particular amount of $keys$ is $log(M)$, where M is amount of $keys$.

So the worst case time complexity is

$$O(\sum_{n=0}^{log(M)} 2^n * log(N/2^n)) \tag{3}$$

6 Conclusion and Future Research

The paper proposes a new elementary algorithm for search of multiple keys over a sorted array of values, the newly introduced algorithm represents a next variation of multi-key binary search algorithm with several advantages over its predecessors including: performance, simple implementation and no limit on size of searched keys, in other words array of searched keys can be larger than array of values, furthermore without any negative impact on performance. The last is ensured by mechanism for throwing away redundant keys. The paper also shows related measurements and comparison to sequential method. Further the paper explores time complexity of the algorithm.

The future research will be focused on non-recursive implementation of the algorithm for its potential to optimize computational efficiency as well as potential to reduce its memory consumption in form of procedure call stack.

References

1. Tarek, A.: A Logarithmic Algorithm for Multi-Key Binary Search. Proc. SCI'04, 2004, vol. II, pp. 222-227
2. Tarek, A.: A New Approach for Multiple Element Binary Search in Database Applications. NAUN INTERNATIONAL JOURNAL OF COMPUTERS, vol. 1, pp. 269279, Issue 4, 2007
3. Tarek, A.: Multi-key Binary Search and the Related Performance. Proc. WSEAS American Conference on Applied Mathematics, Univer-sity of HARVARD, Cambridge, 2008, pp. 104-109
4. Tarek, A.: A New Algorithm for Tiered Binary Search. Proc. FCS'17 Int'l Conf. Foundations of Computer Science, 2017, pp. 25-31