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

Július Milan

mathematician, researcher
julikmilan@gmail.com

Abstract. The Milan's Multi-key Binary Search is an algorithm for fast search of multiple keys in a sorted array of values. The algorithm is based on classical, i.e. one-key binary search and uses it as its fundamental part, additionally it takes advantage of the knowledge obtained from its previous results to adjust the boundaries of each next binary search and so lower the amount of values that will be searched by binary search. The algorithm has one limitation, it needs to obtain its *keys* input sorted, otherwise it is not guaranteed and also highly probable that some of the searched *keys* won't be found. 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

The use case for the multi-key search algorithm is very common and frequent or in some cases implementation of the concrete solutions can be designed to use it and profit from its efficiency. Often it is needed to search for more than one key over an array of values and to return a *"not found"* information for every single key separately in case of finding nothing. To achieve this, 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 time complexity function of Milan's Multi-Key Binary Search (further just `mmkbs`) grows logarithmically considering M variable as well. Exact form is shown bellow.

2 Terminology

`bs` : Binary Search
`mmkbs` : Milan's Multi-Key Binary Search
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

Variable names used in example implementation of Sect. 3.3 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 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.

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.

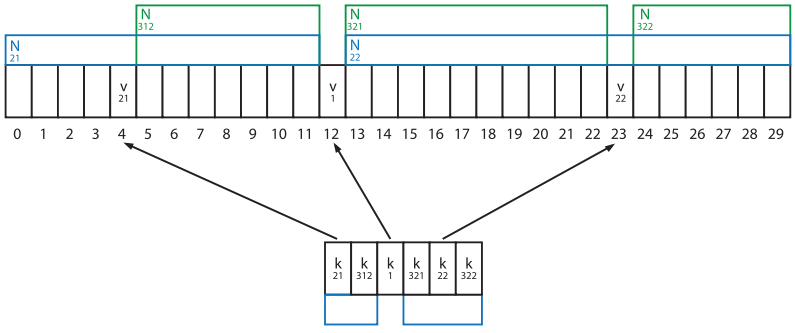

Fig. 1. Example search, shows ranges of next searches, if k_1 was found on position 12 of *arr*, k_{21}-s range to search is shown by upper left blue line and is of size N_{21}.

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)
```

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:

```
1 int bs (const int *arr, int left, int right, int key,
           bool *found)
2 {
3     int middle = MIDDLE(left, right);
4
5     while (left <= right) {
6         if (key < arr[middle])
7             right = middle - 1;
8         else if (key == arr[middle]) {
9             *found = true;
10            return middle;
11        }
12        else
13            left = middle + 1;
14        middle = MIDDLE(left, right);
15    }
16
17    *found = false;
18    /* left points to the position of first bigger element */
19    return left;
20 }
```

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:

```
22 void _mmkbs (const int *arr, int arr_l, int arr_r,
               const int *keys, int M,
               int *results)
23 {
24     if (M <= 0)
25         return;
26
27     int keys_middle = MIDDLE(0, M - 1);
28
29     /* throw away half of keys, if key on keys_middle is out */
30     if (keys[keys_middle] < arr[arr_l]) {
31         _mmkbs(arr, arr_l, arr_r, &keys[keys_middle + 1],
                  M - 1 - keys_middle, &results[keys_middle + 1]);
32         return;
33     }
34     else if (keys[keys_middle] > arr[arr_r]) {
35         _mmkbs(arr, arr_l, arr_r, keys, keys_middle, results);
36         return;
37     }
38
39     bool found;
40     int pos = bs(arr, arr_l, arr_r, keys[keys_middle], &found);
41
42     if (found)
43         results[keys_middle] = pos;
44
45     _mmkbs(arr, arr_l, pos - 1, keys, keys_middle, results);
46     _mmkbs(arr, (found) ? pos + 1 : pos, arr_r,
              &keys[keys_middle + 1], M - 1 - keys_middle,
              &results[keys_middle + 1]);
47 }
48
49 void mmkbs (const int *arr, int N,
              const int *keys, int M,
              int *results)
50 {    _mmkbs(arr, 0, N - 1, keys, M, 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:

```
52 void m_times_bs (const int *arr, int N, const int *keys, int M,
                   int *results)
53 {
54     for (int i = 0; i < M; i++)
55     {
56         bool found;
57         int pos = bs(arr, 0, N - 1, keys[i], &found);
58         if (found)
59             results[i] = pos;
60     }
61 }
```

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.

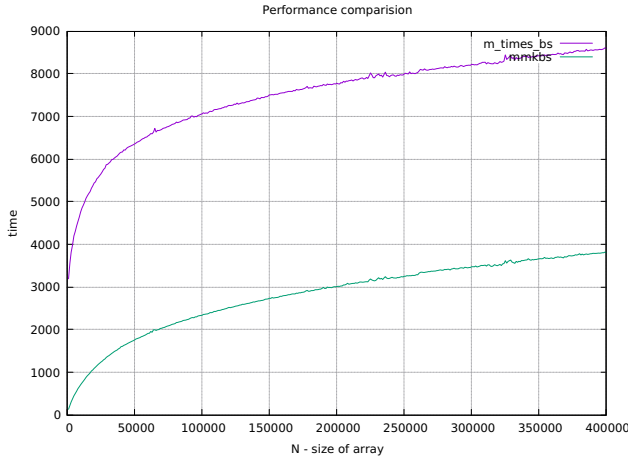

Fig. 2. Performance comparison with fixed M of size 50 000, x-axis represent variable N, y-axis represents time in CPU ticks.

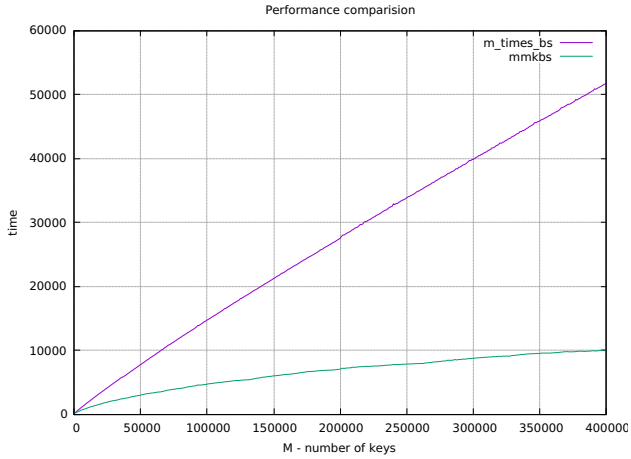

Fig. 3. Performance comparison with fixed N of size 200.000, x-axis represent variable M, y-axis represents time in CPU ticks.

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 is also equal 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

As shown, multi-key binary search can be used for searching for more than one key at once with much better performance than multiple times called binary search. Furthermore for large sets of *keys*, it should also be faster than big amount of hash-table lookups.

Algorithm is able to run searches with $M > N$, i.e. when the size of searched *keys* is bigger than the size of searched *arr* values. Furthermore performance will not suffer even if $M > N$ thanks to mechanism for throwing away redundant keys and time consumed grows further logarithmically.