Binary Search

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

Simulation Weblink

https://www.cs.usfca.edu/~galles/visualization/Search.html

1 What is Searching?

Given a collection of k items $\{I_1, I_2, \ldots, I_k\}$, searching is the task of finding whether an item called *key* is present in the collection or not. If present, usually we are required to return an index i such that $I_i = key$. Needless to say that such a fundamental problem has numerous (real-life) applications and certainly needs no further justification as to why we should study this problem.

What we are interested in is arguably the simplest form of searching: given a collection of integers, find whether *key* is present in the collection. Like in so many numerous instances, by a collection of integers, obviously we mean an array of integers.

We arrive at our formal setting: *given an integer array $A[0, n-1]$, find an index i (if exists) such that $A[i] = key$, where key is the integer to be searched.*

2 Linear Search an Array

Obviously, the simplest way to find *key* is simply search the array from 0 to $n-1$, and when *key* is found, return the corresponding index of the array. If no index is found, return -1. This is formalized in the following pseudo-code.

Algorithm 1 Linear Search

```
function linearSearch(int array[n], int key) {
    for (int i = 0; i < n; i++)
        if (array[i] == key)
            return i;
    return -1;
}
```

As we have learned from our complexity analysis lectures, this has a worst-case complexity of $O(n)$. So the real question is: *can we do better?* Alas, if the array is completely random and no additional information is stored, the situation is rather gloomy: linear search is the best we can do.

So, we are going to make a very REASONABLE[1] assumption: *the array is sorted*, i.e., $A[0] \leq$

[1] It is reasonable because searching is a highly used functionality, whereas the array to be searched hardly changes. Therefore, we may assume that our array is initially sorted, if not, we carry out a ONE-time sorting of the array.

$A[1] \leq \cdots A[n-1]$. My claim is that now we can find *key* (or detect that it does not exist) in $O(\log n)$ time. *How?* **Welcome to Binary Search!**

3 Binary Search an Array

As we have discussed so far, we NEED a SORTED array. **Remember** binary search MAY NOT give you the correct result, if your array is not sorted. However, the sorted property is the one and only assumption we make, and it plays a huge role.

Let us look at the **key intuition**. Suppose $key < A[m]$ for some index m in the array, i.e., $0 < m \leq n-1$. Since the array is sorted, we are absolutely sure of one fact – *key* is not present in the sub-array $A[m, n-1]$. So, we are only concerned with the sub-array $A[0, m-1]$. **For example,** consider $A[0, 7] = \{1, 3, 7, 9, 11, 13, 17, 34\}$. If $key = 9$ and $m = 4$, then observe that *key* is not present in $A[4, 7]$. So, we can simply ignore the numbers that lie from index 4 to 7. *Why?* Because the array is SORTED – that is crucial.

To search an element in the array, we define something called a *relevant array* – it is a sub-array of our input array which may contain *key*. In other words, anything outside the relevant array surely does not equal *key*. Now, we carry out the following steps as long as we have a properly defined relevant array, failing which *key* is not present and we return -1.

1. Initially, the entire array $A[0, n-1]$ is relevant.
2. Pick the middle element $A[mid]$ in the relevant array. Compare it with *key*.
3. If $key = A[mid]$ then we have found *key* at index *mid*. Return *mid*.
4. If $key > A[mid]$, then *key* can only be present in the array $A[mid+1, n-1]$, which becomes our new *relevant array*. Repeat the process from Step 2.
5. If $key < A[mid]$, then *key* can only be present in the array $A[0, mid-1]$, which becomes our new *relevant array*. Repeat the process from Step 2.

The above algorithm is formalized in the following pseudo-code.

Algorithm 2 Binary Search

```
function BINARYSEARCH(int array[n], int key) {
    int left = 0, right = n − 1;
    while (left ≤ right)  {
        int mid = (left + right)/2;
        if (array[mid] == key)
            return mid;
        else if (array[mid] > key)
            right = mid − 1;
        else
            left = mid + 1;
    }
    return -1;
}
```

Note that binary search (unlike linear search) may return any occurrence of *key* in the array and not necessarily the first occurrence. Additionally, the relevant array in each iteration of the while-loop is the sub-array $A[left, right]$. A relevant array is *not defined* when $left > right$.

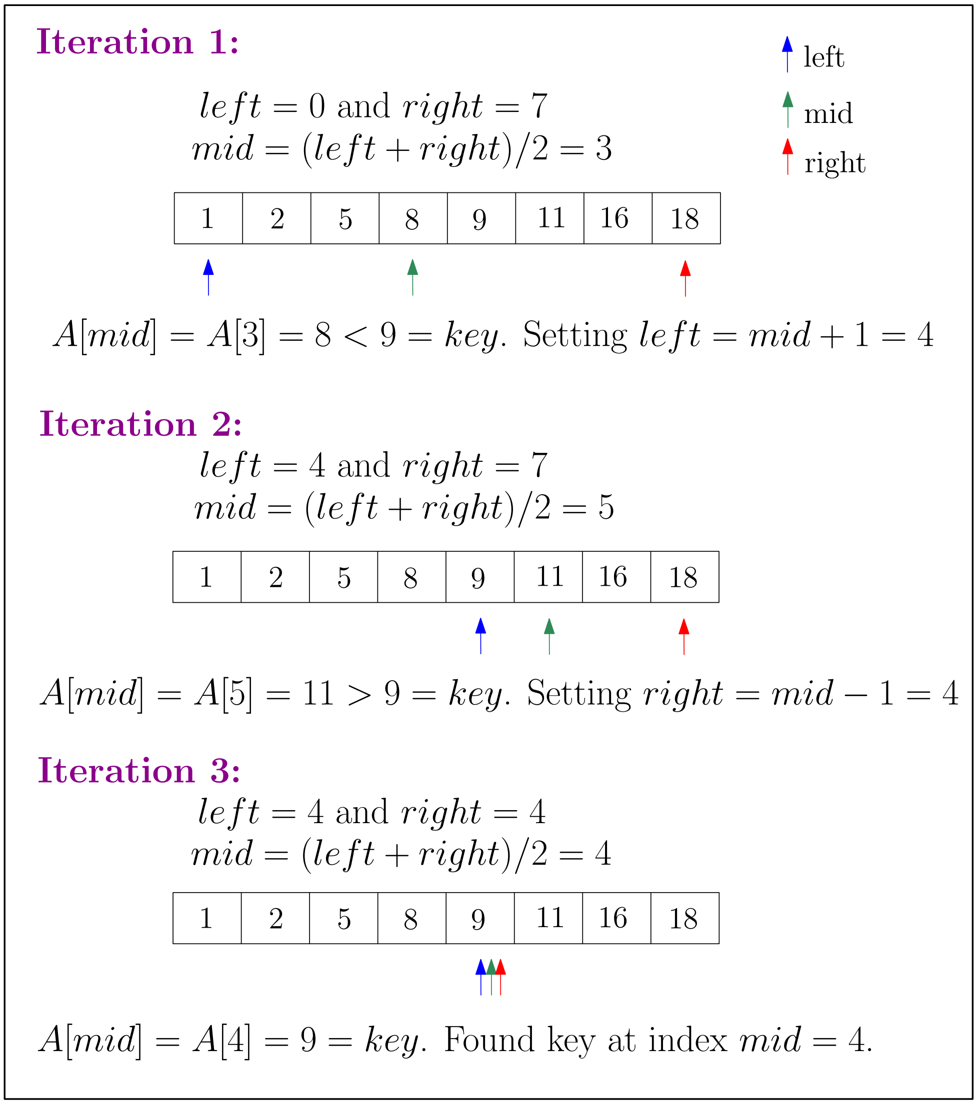

Figure 1: Binary-searching $key = 9$ in $\{1, 2, 5, 8, 9, 11, 16, 18\}$

Let us now use the pseudo-code above to find $key = 9$ in the array $\{1, 2, 5, 8, 9, 11, 16, 18\}$.

Observe that we only need 3 iterations of the while-loop to find key, whereas a linear-search would have needed 5 iterations (to get to position 4) in the array. In fact, for this array, you will always find key (or detect there is none), in at most 3 iterations.

Let us now use the binary search pseudo-code to find $key = 12$ in the array $\{1, 2, 5, 8, 9, 11, 16, 18\}$, i.e., the case when key is not present in the array.

4 Analysis of Binary Search

Now, we come to the final part of our discussion – establishing why binary is search is faster than linear search. To understand this, first you need to know a couple of things:

- $\log 2^y = y$, where log is logarithm in base 2.

Iteration 1:

left · mid · right

$$left = 0 \text{ and } right = 7$$
$$mid = (left + right)/2 = 3$$

1	2	5	8	9	11	16	18

$$A[mid] = A[3] = 8 < 12 = key. \text{ Setting } left = mid + 1 = 4$$

Iteration 2:

$$left = 4 \text{ and } right = 7$$
$$mid = (left + right)/2 = 5$$

1	2	5	8	9	11	16	18

$$A[mid] = A[5] = 11 < 12 = key. \text{ Setting } left = mid + 1 = 6$$

Iteration 3:

$$left = 6 \text{ and } right = 7$$
$$mid = (left + right)/2 = 6$$

1	2	5	8	9	11	16	18

$$A[mid] = A[6] = 16 > 12 = key. \text{ Setting } right = mid - 1 = 5 < left.$$

Since $right < left$, relevant array is not defined. Key is not present.

Figure 2: Binary-searching $key = 12$ in $\{1, 2, 5, 8, 9, 11, 16, 18\}$

- We can successively divide (integer division) an integer n by 2 at most $\lceil \log n \rceil$ times before it reaches 1.[2] For example, $16 \xrightarrow{/2} 8 \xrightarrow{/2} 4 \xrightarrow{/2} 2 \xrightarrow{/2} 1$ requires $4 = \log 16$ divisions.

Now observe what we are doing in binary search – simply halving the relevant array each time inside the while-loop (or returning that key is found at mid). Hence, if we start with an array of length n, we will terminate in at most $\lceil \log n \rceil$ iterations of the while-loop. Since, we have only constant number of operations inside and outside the while loop, our complexity is $O(\log n)$, which is much faster than $O(n)$ required for linear search.

[2]In case you are not aware, $\lceil\ \rceil$ is the ceiling function, i.e., $\lceil x \rceil = x$ if x is an integer, else it equals the next higher integer. Thus $\lceil 5.34 \rceil = 6$ and $\lceil 6 \rceil = 6$. Note that $\lceil x \rceil < x + 1$.