

Searching Algorithms

Consider the problem of searching an array of n integers to find the one with a particular value K . There are two basic approaches:

- sequential search that requires $O(n)$ time;
- binary search that requires $O(\log_2 n)$ time.

Binary search is much faster than sequential search, but it works only on sorted arrays.

1. Sequential Search

The algorithm begins at the first element in the array and looks at each element in turn until K is found.

```

ALGORITHM sequentialSearch(A[0..n-1], K)
// Input: an array A[0..n-1] and a search key K
// Output: the index of the element of A that matches K,
// or -1, if there is no such element

i ← 0
while i ≤ n-1 and A[i] ≠ K do
    i ← i+1
if i < n return i           // i is the index of key K
else return -1             // key K is not found

```

The time complexity of sequential search is $O(n)$ since in the worst case the algorithm performs n comparisons (if key K is the last element in the array).

Observe, that in the best case, one comparison is done (if key K is the first element in the array).

In the average case, $n/2$ comparisons are done.

2. Binary Search

Binary search is based on using the famous divide-and-conquer technique, which consists of the following three main steps.

1. **Divide**: a problem's instance is divided into two smaller instances.
2. **Recur**: the smaller instances are solved recursively.
3. **Conquer**: if necessary, the solutions obtained for the smaller instances are combined in order to get a solution to the original problem.

These ideas can be successfully applied to the problem of finding a key K in an ordered array of n integers. Suppose the array is sequenced in **ascending order**. Then the main steps of divide-and-conquer can be described as follows.

1. **Divide** the array into two sub-arrays about half as large. If key K is smaller than the middle element, choose the left sub-array. If K is larger, choose the right sub-array.
2. **Recur**: apply the same method to solve a smaller instance, i.e., to search a sub-array to find key K . The algorithm stops when either the search key is found or the sub-array is "sufficiently small".
3. **Conquer**: since only one half of the array is considered in each step and the other half is discarded, there is no need to combine the solutions.

```

ALGORITHM binarySearch(A[l..r], K)
//The algorithm implements iterative binary search
//Input:   an array A[l..r] sorted in ascending order, defined by its
//          left and right indices l and r
//          a search key K
//Output: an index of the array's element that is equal to K
//          or -1 if there is no such element
while l ≤ r do
    mid ← ⌊(l+r)/2⌋
    if K == A[mid] return mid
    else
        if K < A[mid] r ← mid-1
        else l ← mid+1
return -1 //Search key not in array

```

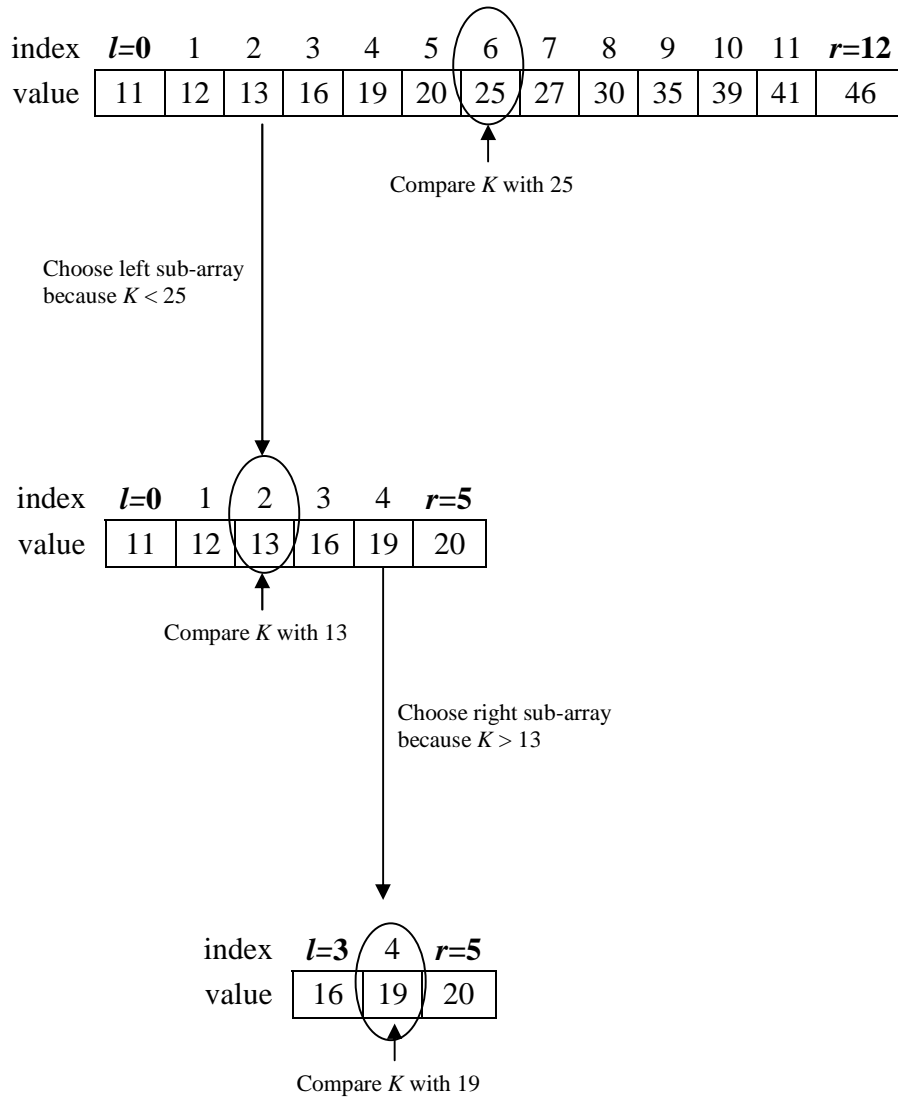
If at some stage we consider a part of array with indices $l, l+1, \dots, r$, $0 \leq l \leq r \leq n-1$, then the middle element is found as $\lfloor (l+r)/2 \rfloor$, which is the largest integer smaller than or equal to $(l+r)/2$. It is the integer part of $(l+r)/2$ (the fractional part is discarded).

For example,

- if we have an array of 13 elements with the left index $l=0$ and right index $r=12$, then $\lfloor (l+r)/2 \rfloor = \lfloor (0+12)/2 \rfloor = 6$.
- if we have an array of 12 elements with the left index $l=0$ and right index $r=11$, then $\lfloor (l+r)/2 \rfloor = \lfloor (0+11)/2 \rfloor = 5$.

Example.

Consider an array of 13 elements and search key $K = 19$. The operation of binary search is illustrated in the figure below.



The search key is found.
Return its index **4**.

In-class exercise: Fill in the tracing table

l	r	mid	$K == A[mid]?$ (true/false)	$K < A[mid]?$ (true/false)
0	12			

Analysis of Binary Search

In each iteration binary search discards approximately one half of the array by comparing the search key with the middle element. To estimate the number of iterations, suppose first that n is a power of 2, i.e.,

$$n = 2^q \quad (1)$$

for some q .

The search requires the following steps:

- inspect the middle element of an array of size n ;
- inspect the middle element of an array of size $\frac{n}{2}$;
- inspect the middle element of an array of size $\frac{n}{2^2}$, and so on.

If we divide an array of n elements in half, then divide one of those halves in half, and continue dividing halves until only one item remains, we should perform q divisions (because $\frac{n}{2^q} = 1$ according to our assumption (1)). Thus in the worst case the algorithm performs q iterations and, therefore, q comparisons. Due to (1),

$$q = \log_2 n.$$

It means that the time complexity of the algorithm is $O(\log_2 n)$ in the worst case.

Suppose now that n is not a power of 2. We can always find an integer q such that

$$2^{q-1} < n < 2^q.$$

For example, if $n=10$, then $q=4$, because

$$2^3 < 10 < 2^4.$$

The algorithm still performs at most q iterations to obtain a sub-array with one element. Now it follows that

$$q-1 < \log_2 n < q,$$

$$q < \log_2 n + 1 < q+1,$$

$$q = \lfloor \log_2 n \rfloor + 1.$$

Thus the algorithm is still $O(\log_2 n)$ in the worst case when $n \neq 2^q$.

In general, the binary search algorithm is $O(\log_2 n)$ in the worst case for any n .

Binary search is much better than sequential search. For example, $\lfloor \log_2 1,000,000 \rfloor = 19$, so a sequential search of one million sorted elements can require one million comparisons while a binary search will require at most 20 comparisons. For large arrays, the binary search has a huge advantage over a sequential search.