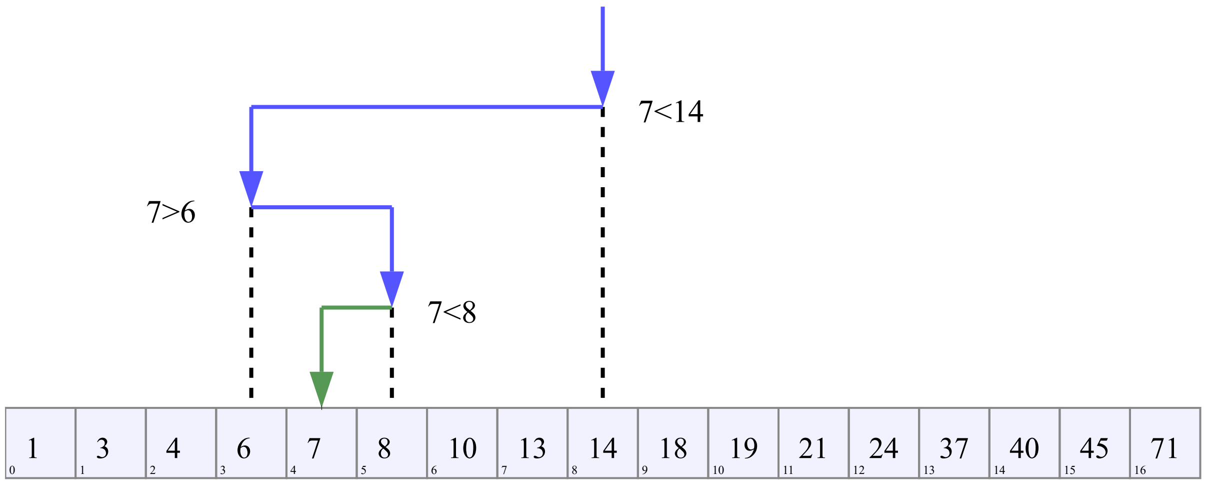
Last update: July 21, 2024 Original

Binary search

Binary search is a method that allows for quicker search of something by splitting the search interval into two. Its most common application is searching values in sorted arrays, however the splitting idea is crucial in many other typical tasks.

Search in sorted arrays

The most typical problem that leads to the binary search is as follows. You're given a sorted array $A_0 \le A_1 \le \dots \le A_{n-1}$, check if k is present within the sequence. The simplest solution would be to check every element one by one and compare it with k (a so-called linear search). This approach works in $O(n)$, but doesn't utilize the fact that the array is sorted.



Binary search of the value 7 in an array.
The image by AlwaysAngry is distributed under CC BY-SA 4.0 license.

Now assume that we know two indices $L < R$ such that $A_L \le k \le A_R$. Because the array is sorted, we can deduce that k either occurs among $A_L, A_{L+1}, \dots, A_R$ or doesn't occur in the array at all. If we pick an arbitrary index M such that $L < M < R$ and check whether k is less or greater than A_M. We have two possible cases:

1. $A_L \le k \le A_M$. In this case, we reduce the problem from $[L, R]$ to $[L, M]$;
2. $A_M \le k \le A_R$. In this case, we reduce the problem from $[L, R]$ to $[M, R]$.

When it is impossible to pick M, that is, when $R = L + 1$, we directly compare k with A_L and A_R. Otherwise we would want to pick M in such manner that it reduces the active segment to a single element as quickly as possible *in the worst case*.

Since in the worst case we will always reduce to larger segment of $[L, M]$ and $[M, R]$. Thus, in the worst case scenario the reduction would be from $R - L$ to $\max(M - L, R - M)$. To minimize this value, we should pick $M \approx \frac{L+R}{2}$, then

$$M - L \approx \frac{R - L}{2} \approx R - M.$$

In other words, from the worst-case scenario perspective it is optimal to always pick M in the middle of $[L, R]$ and split it in half. Thus, the active segment halves on each step until it becomes of size 1. So, if the process needs h steps, in the end it reduces the difference between R and L from $R - L$ to $\frac{R-L}{2^h} \approx 1$, giving us the equation $2^h \approx R - L$.

Taking $\log_2$ on both sides, we get $h \approx \log_2(R - L) \in O(\log n)$.

Logarithmic number of steps is drastically better than that of linear search. For example, for $n \approx 2^{20} \approx 10^6$ you'd need to make approximately a million operations for linear search, but only around 20 operations with the binary search.

Lower bound and upper bound

It is often convenient to find the position of the first element that is greater or equal than k (called the lower bound of k in the array) or the position of the first element that is greater than k (called the upper bound of k) rather than the exact position of the element.

Together, lower and upper bounds produce a possibly empty half-interval of the array elements that are equal to k. To check whether k is present in the array it's enough to find its lower bound and check if the corresponding element equates to k.

Implementation

The explanation above provides a rough description of the algorithm. For the implementation details, we'd need to be more precise.

We will maintain a pair $L < R$ such that $A_L \leq k < A_R$. Meaning that the active search interval is $[L, R)$. We use half-interval here instead of a segment $[L, R]$ as it turns out to require less corner case work.

When $R = L + 1$, we can deduce from definitions above that R is the upper bound of k. It is convenient to initialize R with past-the-end index, that is $R = n$ and L with before-the-beginning index, that is $L = -1$. It is fine as long as we never evaluate A_L and A_R in our algorithm directly, formally treating it as $A_L = -\infty$ and $A_R = +\infty$.

Finally, to be specific about the value of M we pick, we will stick with $M = \lfloor \frac{L+R}{2} \rfloor$.

Then the implementation could look like this:

```
... // a sorted array is stored as a[0], a[1], ..., a[n-1]
int l = -1, r = n;
while (r - l > 1) {
    int m = (l + r) / 2;
    if (k < a[m]) {
        r = m; // a[l] <= k < a[m] <= a[r]
    } else {
        l = m; // a[l] <= a[m] <= k < a[r]
    }
}
```

During the execution of the algorithm, we never evaluate neither A_L nor A_R, as $L < M < R$. In the end, L will be the index of the last element that is not greater than k (or -1 if there is no such element) and R will be the index of the first element larger than k (or n if there is no such element).

Note. Calculating `m` as `m = (r + l) / 2` can lead to overflow if `l` and `r` are two positive integers, and this error lived about 9 years in JDK as described in the blogpost. Some alternative approaches include e.g. writing `m = l + (r - l) / 2` which always works for positive integer `l` and `r`, but might still overflow if `l` is a negative number. If you use C++20, it offers an alternative solution in the form of `m = std::midpoint(l, r)` which always works correctly.

Search on arbitrary predicate

Let $f : \{0, 1, \dots, n-1\} \to \{0, 1\}$ be a boolean function defined on $0, 1, \dots, n-1$ such that it is monotonously increasing, that is

$$f(0) \leq f(1) \leq \dots \leq f(n-1).$$

The binary search, the way it is described above, finds the partition of the array by the predicate $f(M)$, holding the boolean value of $k < A_M$ expression. It is possible to use arbitrary monotonous predicate instead of $k < A_M$. It is particularly useful when the computation of $f(k)$ is requires too much time to actually compute it for every possible value. In other words, binary search finds the unique index L such that $f(L) = 0$ and $f(R) = f(L+1) = 1$ if such a *transition point* exists, or gives us $L = n-1$ if $f(0) = \dots = f(n-1) = 0$ or $L = -1$ if $f(0) = \dots = f(n-1) = 1$.

Proof of correctness supposing a transition point exists, that is $f(0) = 0$ and $f(n-1) = 1$: The implementation maintaints the *loop invariant* $f(l) = 0, f(r) = 1$. When $r - l > 1$, the choice of m means $r - l$ will always decrease. The loop terminates when $r - l = 1$, giving us our desired transition point.

```
... // f(i) is a boolean function such that f(0) <= ... <= f(n-1)
int l = -1, r = n;
while (r - l > 1) {
    int m = (l + r) / 2;
    if (f(m)) {
        r = m; // 0 = f(l) < f(m) = 1
    } else {
        l = m; // 0 = f(m) < f(r) = 1
    }
}
```

Binary search on the answer

Such situation often occurs when we're asked to compute some value, but we're only capable of checking whether this value is at least i. For example, you're given an array $a_1, \dots, a_n$ and you're asked to find the maximum floored average sum

$$\left\lfloor \frac{a_l + a_{l+1} + \dots + a_r}{r - l + 1} \right\rfloor$$

among all possible pairs of l, r such that $r - l \geq x$. One of simple ways to solve this problem is to check whether the answer is at least λ, that is if there is a pair l, r such that the following is true:

$$\frac{a_l + a_{l+1} + \dots + a_r}{r - l + 1} \geq \lambda.$$

Equivalently, it rewrites as

$$(a_l - \lambda) + (a_{l+1} - \lambda) + \dots + (a_r - \lambda) \geq 0,$$

so now we need to check whether there is a subarray of a new array $a_i - \lambda$ of length at least $x + 1$ with non-negative sum, which is doable with some prefix sums.

Continuous search

Let $f : \mathbb{R} \to \mathbb{R}$ be a real-valued function that is continuous on a segment $[L, R]$.

Without loss of generality assume that $f(L) \leq f(R)$. From intermediate value theorem it follows that for any $y \in [f(L), f(R)]$ there is $x \in [L, R]$ such that $f(x) = y$. Note that, unlike previous paragraphs, the function is *not* required to be monotonous.

The value x could be approximated up to $\pm\delta$ in $O\left(\log \frac{R-L}{\delta}\right)$ time for any specific value of δ. The idea is essentially the same, if we take $M \in (L, R)$ then we would be able to reduce the search interval to either $[L, M]$ or $[M, R]$ depending on whether $f(M)$ is larger than y. One common example here would be finding roots of odd-degree polynomials.

For example, let $f(x) = x^3 + ax^2 + bx + c$. Then $f(L) \to -\infty$ and $f(R) \to +\infty$ with $L \to -\infty$ and $R \to +\infty$. Which means that it is always possible to find sufficiently small L and sufficiently large R such that $f(L) < 0$ and $f(R) > 0$. Then, it is possible to find with binary search arbitrarily small interval containing x such that $f(x) = 0$.

Search with powers of 2

Another noteworthy way to do binary search is, instead of maintaining an active segment, to maintain the current pointer i and the current power k. The pointer starts at $i = L$ and then on each iteration one tests the predicate at point $i + 2^k$. If the predicate is still 0, the pointer is advanced from i to $i + 2^k$, otherwise it stays the same, then the power k is decreased by 1.

This paradigm is widely used in tasks around trees, such as finding lowest common ancestor of two vertices or finding an ancestor of a specific vertex that has a certain height. It could also be adapted to e.g. find the k-th non-zero element in a Fenwick tree.

Practice Problems

- LeetCode - Find First and Last Position of Element in Sorted Array
- LeetCode - Search Insert Position
- LeetCode - First Bad Version
- LeetCode - Valid Perfect Square
- LeetCode - Find Peak Element
- LeetCode - Search in Rotated Sorted Array
- LeetCode - Find Right Interval
- Codeforces - Interesting Drink
- Codeforces - Magic Powder - 1
- Codeforces - Another Problem on Strings
- Codeforces - Frodo and pillows
- Codeforces - GukiZ hates Boxes
- Codeforces - Enduring Exodus
- Codeforces - Chip 'n Dale Rescue Rangers

Contributors:
adamant-pwn (80.77%) jakobkogler (7.69%) RomanSteinberg (3.21%) jxu (2.56%) 100godmoon (1.92%) iAngkur (1.92%) Draac0 (0.64%) mbordia-eightfold (0.64%) twfksh (0.64%)