

Binary search and other "halving" methods

November 7, 2021 · 27 min · 5555 words · nor

This post was originally written on Codeforces; relevant discussion can be found [here](#).

As a certain legendary grandmaster once said (paraphrased), if you know obscure techniques and are not red yet, go and learn binary search.

The main aim of this tutorial is to collect and explain some ideas that are related to binary search, and collect some great tutorials/blogs/comments that explain related things. I haven't found a comprehensive tutorial for binary search that also explains some intuition about how to get your own implementation right with invariants, as well as some language features for binary search, so I felt like I should write something that covers most of what I know about binary search.

This is mainly geared towards beginners, but some of the topics that I mention in the other resources, optimizing binary search and language-specifics sections might not be well-known to people in the low div-1 range. I will also include some references to pretty high-quality articles/resources with related content, for the interested.

Contents

1. The main idea behind binary search
 1. A small classical example
 2. A more general idea
 3. The generalization



2. Binary searching on the answer
3. Two other common ways of implementing binary search
4. Binary lifting for binary search
5. Language specific ways for binary searching
 1. C++
 2. Python
 3. Java
6. Optimizing binary search
7. Other resources

The main idea behind binary search

The main idea behind binary search is *linear reduction of search space*. We'll elaborate on this below.

A small classical example

Let's start with the classical example of binary search. Suppose you have an array $[1, 7, 9, 12, 19]$, and you're asked to check if 7 is a member of this array or not (and also return its 0-indexed position in this array). A naive approach would be to iterate over all elements, and check if any element is 7 or not. This would take 5 iterations.

Now note that this array is sorted. So if I check some element at position p in this array, we can have one of three possible outcomes:

- The element is equal to 7: this means that we are done, and we can terminate our search!
- The element is less than 7: since the array is sorted, all elements with position $< p$ are less than 7 as well. So it is useless to look at any element with position $\leq p$ now.
- The element is more than 7: in a similar manner to the previous case, it is useless to look at any element with position $\geq p$ now.



So we could do something like this: initially our search space is the range of indices $[0, 4]$. Let's try looking at the middle element of our search space. If we do that, we end up in one of two scenarios:

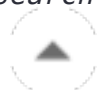
- We terminate if the element at that position is 7
- We reduce the search space to at most half of the original search space each time.

For a concrete example, let's look at the middle element of the search space, which is 9. Since this is more than 7, it is useless to look at indices in the range $[2, 4]$. So our search space reduces to $[0, 1]$. There are two midpoints of this array, let's just use the left one for the sake of uniformity. The element we are now looking at is 1, which is less than 7. So it is useless to look at indices in the range $[0, 0]$. So our search space becomes $[1, 1]$. The middle position is just 1, and since the element there is 7, we can return the position 1 as the answer.

What if there was a 4 instead of 7 in the original array? In the last step of the above dry-run of the algorithm, we would note that it is useless to look at indices in the range $[0, 1]$, so the search space becomes empty! When the search space becomes empty, that means we haven't found it yet. So we should return some special value indicating that the search function resulted in a failure. We'll use the size of the array as the return value for now (we will see why we took this choice later on).

An implementation might look like the following:

```
int find_position(const vector<int>& a, int x) {
    int l = 0;
    int r = (int)a.size() - 1;    // [l, r] is our search space
    while (l <= r) {              // search space is non-empty
        int m = l + (r - l) / 2;  // "middle" position in the range
        if (a[m] == x) return m;  // found!
        else if (a[m] < x) {
            l = m + 1;            // remove all indices <= m from the search
        } else {
            r = m - 1;            // remove all indices >= m from the search
        }
    }
    return n;                    // failure
}
```



Is this more efficient than a linear search? Note that at each step, we reduce the search space by at least half, so in $1 + \log_2 n$ iterations (where n is the size of a), the search space size reduces to < 1 , and since the size is an integer, it becomes 0. It's easy to see how it terminates.

A more general idea

More often than not, binary search is not used in such a simple use-case in competitive programming (or in real life).

Let's have a closer look at what we were doing in the above algorithm. We were using some kind of ordering (the ordering of integers, in our example) to discard a large part of the search space (in fact, we discarded about half the search space each time).

How do we generalize this further? More importantly, what will be the statement of the generalized problem?

Firstly, we will do something that will make our original problem easier to generalize. Rather than checking if an element is present in the array, we can find the position of the first element that is greater than or equal to the element (if no such element exists, we'll return the size of the array).

Clearly, this is even stronger than our original problem. If the answer to this problem is the index i , we can check if this is the right answer by checking if $i < n$ and $a[i] = x$. If this condition isn't true, we don't have any occurrence of x in a .

Let's try to do away with the ordering, and see how far we go.

Let's mentally (not in code) construct an array b , where the value of $b[i]$ is true if and only if $a[i] < x$.

How does b look like? Clearly, some (possibly empty) prefix of it is filled with "true", and the remaining (possibly empty) suffix is filled with "false".

Then our problem reduces to finding the position of the first false (n if not found) in this kind of an array b . For now, forget all about a and x , and focus



only on b .

It turns out, that for any such array b , we can easily find the first false using the same idea of discarding parts of the search space.

Let's start with some notation. $[l_0, r_0]$ will be the interval we need to search (for our problem it is $[0, n - 1]$). l, r will be indices we shall maintain at each step such that the range of indices $[l_0, l]$ in b consists only of "true" values, and the range of indices $[r, r_0]$ in b consists only of "false" values.

Initially, we have zero information about any element of b , so it is better to force these ranges to be empty ranges. We can do so by setting $l = l_0 - 1$ and $r = r_0 + 1$ initially. We will try to increase l and decrease r so that these two ranges cover the whole search space. So, in the end, l will correspond to the location of the last true, and r will correspond to the location of the first false, and we can simply return r !

Let's suppose that at some point, $r - l > 1$. This means that there is at least one element between the indices l and r that we haven't put in one of these intervals yet. Let's take an index roughly in the middle of this range (l, r) , say $m = \lfloor (l + r)/2 \rfloor$.

- If $b[m]$ is true, then we know that everything to the left of $b[m]$ is true as well, so we can increase the range $[l_0, l]$ to $[l_0, m]$ (which is equivalent to replacing l by m).
- Otherwise, it is false, so everything to the right of $b[m]$ is false as well. So we can increase the range $[r, r_0]$ to $[m, r_0]$.

Each time, we reduce the size of the unexplored range from $r - l - 1$ to $r - m - 1$ or $m - l - 1$, which corresponds to a reduction by at least half. So the number of iterations is at most $\log_2(r_0 - l_0 + 1) + 1$.

An implementation of this would look something like this:



```

int find_first_false(const vector<bool>& b) {
    int l = -1;
    int r = (int)b.size();
    while (r - l > 1) {
        int m = l + (r - l) / 2;
        if (b[m]) {
            l = m;
        } else {
            r = m;
        }
    }
    return r;
}

```

Note that this terminates by our previous observation.

But, we said earlier that we won't really construct b in our code, right? How do we use this same algorithm for solving our own problem?

Note that, by what we said earlier, $b[i]$ is just defined as the truth value of $a[i] < x$, so computing it on the fly is no big deal. So, if we replace $b[m]$ with $a[m] < x$, that would solve our problem.

That was quite a handful, so let's see a brief summary of what we did:

- Rather than explicitly reason using the order $<$ and the value x , we constructed an array b of some very specific form (the first few things in it being true and the rest being false), and found the location of the first false in b .

This very clearly points to our desired generalization:

The generalization

Suppose we are given the following:

- $[l, r]$: a range of integers (in our example, $[0, n - 1]$)
- f : a function from integers to booleans, which satisfies the following property: there exists some integer t such that for all $l \leq x < t$, $f(x)$ is true, and for all $t \leq x \leq r$, $f(x)$ is false.



Then if the time taken to compute f is upper bounded by $T(l, r)$, we can find the value of t (i.e., the first false index) in $O(T(l, r) \log_2(r - l + 1)) + O(1)$ time.

Such an f is called a predicate. In the problem we discussed above, f is called a predicate.

An implementation of this function will be something like the following (ignoring overflow errors):

```
template <class Integer, class F>
Integer find_first_false(Integer l, Integer r, F&& f) {
    --l;
    ++r;
    while (r - l > 1) {
        Integer m = l + (r - l) / 2; // prefer std::midpoint in C++20
        if (f(m)) {
            l = m;
        } else {
            r = m;
        }
    }
    return r;
}
```

Note that this implementation also has the nice property that l is the position of the last true in the corresponding array b , so you can define a function similar to this one that returns l instead.

To use this function to implement our original binary search, we can do something like the following:

```
int find_position(const vector<int>& a, int x) {
    auto f = [&](int i) {
        return a[i] < x;
    };
    int n = (int)a.size();
    int pos = find_first_false(0, n - 1, f);
    if (pos == n || a[pos] != x) return n;
    return pos;
}
```



Note that this abstraction also gives us the following result: we don't really

need a to be sorted at all. The only thing we need is that everything less than x in a should be in a prefix, and everything not less than x should be in the remaining suffix and if x is in the array, the beginning of that suffix should have x at it. This definition also handles possible duplicates of x in the array.

As we'll see later on, this kind of an abstraction allows us to solve a whole variety of problems.

Exercise: What would you do if in b , the first few elements were false, and the rest were true?

Binary searching on the answer

Sometimes, it is much easier to deal with bounds on the answer rather than the exact answer itself.

In other words, sometimes it is much easier to construct a function f that returns true iff the input is \geq the answer to the problem, by running some algorithm that returns a boolean value not by computing the answer, but by considering some properties of the problem at hand and the input.

To make this more concrete, let's consider this problem.

In short, you're given an $n \times m$ multiplication table, and you want to find the k^{th} smallest entry in this table (i.e., if you were to sort all the entries, find the entry at position k).

It is not clear how to directly find this value. But given a "guess" x , we can see if this is at least the answer or not:

Let's find the number of integers smaller than x in each row, and sum them up. This can be done by a simple division for each row.

If the total numbers less than x are $< k$, we will return true, otherwise, we will return false. This predicate works since the number of entries smaller than x is a non-decreasing function of x (more commonly, we compare at $f(x)$, $f(x+1)$ and try to argue that if $f(x)$ is false, then $f(x+1)$ is also false), and hence

the last element that makes the function return true is in fact the k^{th} smallest entry.

This can be found using binary search as we have discussed above.

Two other common ways of implementing binary search

We'll discuss two other ways of implementing usual binary search, and why they seem intuitive to people who use them.

We'll refer to the previous implementation as the (l, r) way, since the invariant in that was equivalent to saying that the remaining search space will always be (l, r) .

This section is just to help out people in reading others' binary search implementations, as well as choose between implementations to find the way you find best. I prefer using the implementation used above, so I'll explain the other two implementations in the context of that, with some intuition as to why people choose to implement things that way.

The $[l, r]$ way

In this type of an implementation, we maintain $l + 1$ and $r - 1$ rather than l and r . The reason behind this is something like the following (independent of the above reasoning):

$[l, r]$ consists of the currently explored search space. Let's maintain an extra integer (*ans*), which stores the "best" answer found so far, i.e., the leftmost false found so far.

This interval is non-empty if $r \geq l$. The midpoint is the same as before, i.e., $m = \lfloor (l + r) / 2 \rfloor$. If $f(m)$ evaluates to false, then the best possible answer so found has to be m , and we reduce the search space from $[l, r]$ to $[l, m - 1]$. Otherwise, we reduce it to $[m + 1, r]$, and do not update the answer.



Note that here we would also need to maintain a separate variable *ans*, and


initialize it appropriately, depending on whether you want the first false (as done above), or the last true (which can be done by moving the update of the *ans* variable across the if-branches), which is also why I haven't used it for a very long time. However, people who prefer closed intervals and this explanation seem to gravitate towards this implementation.

The invariant here remains that if l', r' are the l, r in the old implementation, and l, r are the l, r in this implementation, then $l' = l - 1$ and $ans = r' = r + 1$. In the case where you need to find the last true, the invariant becomes $ans = l' = l - 1$ and $r' = r + 1$.

An implementation is as follows:

```
template <class Integer, class F>
Integer find_first_false(Integer l, Integer r, F&& f) {
    Integer ans = n;
    while (l <= r) {
        Integer m = l + (r - l) / 2; // prefer std::midpoint in C++20
        if (f(m)) {
            l = m + 1;
        } else {
            ans = m;
            r = m - 1;
        }
    }
    return ans;
}
```

The $[l, r)$ way

In this type of an implementation, we maintain $l + 1$ and r instead of l and r . The interpretation is that the search space is in $[l, r)$, and people who like using half-open intervals tend to prefer this implementation. The rationale behind this is that when the search space becomes empty, it corresponds to the range $[ans, ans)$ (if you're trying to find the first false, of course). The structure of the intervals doesn't always correspond in this implementation with the other implementations, since the value of m used is usually slightly different (and usually equal to $\lfloor (l + r) / 2 \rfloor$ where $[l, r)$ is the search space at that point,  in the case that there are two range midpoints, this is the one to the right and not to the left).

It's much more illuminating to think of it in this way: Suppose you have a range $[l, r)$, and you need to insert a false just after the last true in the range. What will be the position of the new false?

Another way of looking at it is: everything in the range $[l, ans)$ corresponds to true, and everything in the range $[ans, r)$ corresponds to false. So ans is some sort of a "partition point" for the array. We will see later on how this exact interpretation is used in an implementation of this function in C++'s STL.

Let's check the midpoint of the range $[l, r)$. If $f(m)$ is true, we will never put it at a location $\leq m$, so l needs to be updated to $m + 1$. Otherwise, we have to put it at a position $\leq m$, so the r needs to be updated to m .

```
template <class Integer, class F>
Integer find_first_false(Integer l, Integer r, F&& f) {
    ++r;
    while (l < r) {
        Integer m = l + (r - l) / 2; // prefer std::midpoint in C++20
        if (f(m)) {
            l = m + 1;
        } else {
            r = m;
        }
    }
    return r;
}
```

Binary lifting for binary search

This and the next sections will be much more terse than the previous section, since we will be doing more of an overview of methods instead of explaining something from scratch.

Consider the following implementation:



```

template <class Integer, class F>
Integer find_first_false(Integer l, Integer r, F&& f) {
    if (l > r) return r + 1;
    ++r;
    Integer w = Integer(1) << __lg(r - l);
    --l;
    if (f(l + w)) l = r - w;
    for (w >= 1; w >= Integer(1); w >= 1)
        if (f(l + w)) l += w;
    return l + 1;
}

```

Here, we try to ensure that the interval sizes are always powers of 2. Why we do so will be explained in the section on optimizing binary search.

After we ensure that the interval sizes are always powers of 2, we can just try to reconstruct the binary representation of $ans - 1 - l$, where ans is what we need to return. For that, we can try to go from the highest bit to the lowest bit, and greedy works here for the same reason that binary representations are unique; adding a higher power leads to the predicate being false.

Language specific ways for binary searching

Some of the most used languages in competitive programming fortunately have some inbuilt functions (albeit limited) to do binary search for you.

C++

- `binary_search` : This function returns a bool that tells you if there is an element equal to the queried element between two iterators (with an optional comparator).
- `lower_bound` , `upper_bound` : If the range occupied by elements comparing equal to x is defined by iterators $[it_l, it_r)$ in a given range of iterators $[input_it_l, input_it_r)$, then `lower_bound` and `upper_bound` return it_l and it_r . In other words, `lower_bound(it1, it2, x, comp)` returns the iterator to the first element in the range of iterators $[it1, it2)$ which is $\geq x$ according to `comp` (which is optional and defaults to the usual comparison), and

`upper_bound` does the same for $> x$ instead.

- `equal_range` : This returns both `lower_bound` and `upper_bound` .
- `partition_point` : This works exactly like the binary search function we wrote in the $[l, r)$ way. In C++20, with `ranges::views::iota` , we can use it to do binary search on the answer as well.

Python

- The `bisect` module has useful functions like `bisect` , `bisect_left` , `bisect_right` that do similar things to `lower_bound` and `upper_bound` .

Java

- `Collections.binarySearch` and `Arrays.binarySearch` do something similar to finding an index/element (not necessarily the first equal or the last equal) using binary search.

Optimizing binary search

In our first implementation, we were returning early when we found the element. Sometimes, it's faster to stop the binary search early, and in those cases, this is sometimes a constant factor optimization.

The binary lifting implemented for binary search above is also a constant factor optimization on some architectures if unrolled manually (which is possible due to the simple nature of the increments, and the possibility of hardcoding the constants), according to John Bentley's Programming Pearls. It is an improvement in some cases, where the locations you binary search on are few in number and repeated in successive binary search applications, leading to better cache usage. An example of doing that optimization to the extreme would be to implement multiple functions (each with a different power of 2), and store the function pointers in an array, and use computed jumps (for instance, by computing the power of 2 needed) to get to the correct function at runtime, which still leads to some speedup on certain architectures.



For instance, for certain types of queries (where either l or r are more or less

the same), the speedup is pretty significant. However, for other types of queries, the simpler version is more efficient. To show the kind of speedups you can get for certain types of queries, I did some benchmarks on queries where the left bound is always the same. For the benchmarking code below, the results are as follows:

► Benchmarking code

Results:

```
-----  
Simple binary search  
744175  
Time: 2184 ms  
-----  
Binary lifting  
744175  
Time: 1504 ms  
-----  
Binary lifting with unrolling  
744175  
Time: 1407 ms  
-----
```

<https://algorithmica.org/en/eytzinger> explains quite a nice method of exploiting the cache and speeding up binary search in a certain setup by a pretty impressive factor.

<https://codeforces.com/blog/entry/76182> explains a variation of binary search which divides the ranges in an uneven order, which can lead to a change at the complexity level as well.

Other resources

As promised, here's a collection of resources that I felt are pretty high-quality, that pertain to topics related to binary search.

A great resource is the Codeforces EDU tutorial (and problems) on binary search.



A great video that also explains binary search is linked in the following post:
<https://codeforces.com/blog/entry/67509>.

There is also an extension to vanilla binary search, called the parallel binary search, and <https://codeforces.com/blog/entry/89694> links to a great video tutorial for this extension.

Binary lifting is a pretty general idea. A tutorial on binary lifting on fenwick trees can be found at <https://codeforces.com/blog/entry/61364>, a great one on binary lifting on trees can be found at <https://usaco.guide/plat/binary-jump?lang=cpp> (which also has links to other resources).

Binary search on segment trees is a quite useful technique too, and a working implementation (with some intuition on how it works) for recursive segment tree can be found at <https://codeforces.com/blog/entry/83883>. An implementation for the iterative segment tree can be found at <https://github.com/atcoder/ac-library/blob/master/atcoder/lazysegtree.hpp>.

A rough idea of how it works (based off a conversation where I was explaining this code to someone) is as follows. Some parts of it might not make sense, so please let me know if there's something unclear.

► **Spoiler**

Please let me know if you find any bugs or typos in this post!

UPD: Added benchmarking code for binary lifting (with and without unrolling) and binary search in a few special cases to show speedups.

Algorithms

Binary-Search

© 2024 [nor's blog](#)

>

