BINARY SEARCH

Searching for an element in an array which is totally random and sorted in no particular order can be done with a linear search which runs in an O(N) time. Obviously, this is tedious in case there are multiple test cases or multiple data sets to solve. For a randomized array, there is no other solution for it.

What if the array is sorted? Or probably there is some sort of monotonic function that defines that array?

Binary search is the fastest and one of the most efficient algorithms for searching any data which is sorted based on some monotonic function.

In this document, I'll show you 4 different types of binary searches that are used in programming very frequently. I'm pretty sure you are used to the first of the binary searches that I'm going to show you.

1. Searching for a specific element:

This is the well-known binary search algorithm. The question would probably be like: Given a sorted array, find the index where an element 'k' occurs. If it doesn't occur in the array, then return -1.

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	4	5	6	7	8	9	10

I'll use the above array from now on to describe the binary search examples.

The length of the total array is 11. Indexing of the array starts from 0.

Now, I'll just tell you guys the basic way of writing an algorithm of this type.

Step number 1: Initialize two variables low and high.

The value of low will always be 0 and the value of high will always be length-1.

For convenience, let the length of the array be 'n'.

int low = 0, high = n-1;

Step number 2: What is the condition for the termination of the loop?

Well, for this type of binary search where you want the ‘exact’ position, the condition is going to be (low <= high)

Step number 3: You need to find what must be the value of ‘mid’ at each iteration. And another thing you need to consider is what will be the new values of low and high after each iteration.

I’ll just write the algorithm and make things easier to understand.

Say that the array is ‘a’ and the element to search is ‘k’.

```
int binarySearch(int[] a, int k)
{
    int n = a.length;
    int low = 0, high = n - 1;

    while (low <= high) {

        int mid = (low + high)/2;

        if(a[mid] == k)
            return mid;

        else if(a[mid] < k)
            low = mid + 1;

        else
            high = mid - 1;

    }

    return -1;

}
```

Now that you have the algorithm, let’s search for the element 24 in the array.

In the first iteration, we have

low = 0, high = 10, mid = 5 (since it is (low + high)/2)

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	4	**5**	6	7	8	9	**10**

/* I colored the indices of low, high and mid so that it becomes easier for you to understand */

First thing we do in the iteration is check if a[mid] == k.

Since a[5] = 39 != 24, it evaluates to false.

The next condition it checks is if a[mid] < k.

Since 39 < 24 is FALSE, it doesn't execute the next line either.

Although I did not write the condition, it is understood that a[mid] > key at this point.

We assign high = mid – 1.

Why "mid – 1" and not just "mid"?

In binary search, we always minimize the array as much as possible to remove all redundant elements from the array. Since we know that 39 is definitely not what we're looking for, we can remove that element. So, we assign the new limits to be [low, mid-1].

The question might seem pointless now, but in the next two types of binary searches, you'll realize why I said so.

Let's continue with the next iteration.

low = 0, high = 4, mid = 2 (since (0 + 4)/2)

2	7	13	19	24	39	47	51	59	62	79
0	1	**2**	3	**4**	5	6	7	8	9	10

/* The shaded portion of the array means that we have removed half the search space and we no longer need to look for anything in that part */

First condition evaluates to FALSE as a[mid] = 13 != 24

Second condition evaluates to TRUE since 13 < 24

The assignment here is low = mid + 1, since anything less than or equal to 13 is definitely not what we're looking for.

Iteration 3:

low = 3, high = 4, mid = 3 (since, (3 + 4)/2 = 3)

2	7	13	19	24	39	47	51	59	62	79
0	1	2	**3**	**4**	5	6	7	8	9	10

First condition: FALSE

Second condition: TRUE

New assignment: low = mid + 1

Next iteration:

low = 4, high = 4, mid = 4

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	**4**	5	6	7	8	9	10

First condition: TRUE;
Now that will execute and will return mid; and mid is the position of the occurrence of 24.

Let us see what would have happened if we were searching for 26.

Everything would be the same until the last iteration.

But in the last iteration, the first condition would be false and the second condition would be true. So, the assignment will be low = mid + 1;

Now, low = 5, high = 4.

The while loop's condition will be false and the loop will terminate. After the loop, it will return -1.

Since a negative number has been returned, it means that the number doesn't exist in the array.

That basically concludes the first and basic type of binary search.

According to your class syllabus, that is all you need.

This next little portion about the same binary search is something that I decided to teach you personally. You might find it useful at times.

In the previous example, notice the last value that was present in the variable 'low'.

Although we returned -1, there was 5 stored in 'low'.

What if we were to search the value 21 in the array?

We would return -1, but the value in low would be 4.

What if we search for 1? The value in low is 0.

What about searching for 99? Value in low is 11.

/* Just replace return -1 with return low in the last line if you want to check. */

Take a look at the array, and the above values of low….. Do you notice something?

If the 'key' that you are looking for *was* present in the array, then the value return by 'low' is the index where key would occur.

Meaning, if you search for 99, the value in low = 11, which is the index where 99 would occur if it was present in the array.

This, in many scenarios can be extremely useful for array manipulation.

So, how do you utilize this value of 'low'?

Why not just return low itself?

But that would be a mistake because we will not know if the element is present in the array or not. (Of course you can check with an 'if' condition *outside*, but it must be a part of the algo).

As the return value will have to be a non-negative number, why not return a non-negative value if it is present, and return a negative number if it isn't?

Cool. That seems fine.

So, the last line should be modified to: `return -low;` Right?

Nope. That doesn't solve it either. Because it causes confusion when low = 0.

Why not modify it to `return -(low+1);`? Yeah… this seems alright.

Hence, we return negative of 'low + 1' from it. That seems to solve our problem now.

Negative of 1 added to a number. Does it sound familiar?

Oh..yeah… That's pretty much what 1s compliment is.

To sum up: `return ~low;` is the last line. (Or `return -(low+1);` for readability)

If the function returns a non-negative value, then the value is present. If it returns a negative value, then that value is not present in the array, and in case you want to find the position where it would occur, then take the bitwise NOT (~) of the returned value.

There are other ways of dealing with checking the presence or absence in the array (for eg. `return -low-2;` or `return -low-3;` or whatever). But having it simple is good.

That's the end of the first type of binary search.

The time complexity for binary search is $\log_2(N)$ This is a LOT faster than a usual linear search. If you want to find a contact in a dictionary that has 7.4 billion names in it, then binary search would take only 33 iterations in the worst case to find its location.

2. Searching for the first value that evaluates to TRUE:

This, and the next one are going to be very similar. And these two are definitely the most important types of binary search.

Consider the same array:

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	4	5	6	7	8	9	10

In this type of binary search, the array splits into two portions: A series of **NO**s followed by a series of **YES**s. Let me give an example: In the given array, find the first number which is at least 50.

If that was the question, the array can be split as follows.

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	4	5	6	7	8	9	10

As we need to find the first element that is at least 50, we mark all the elements that are at least 50 as items that can evaluate to TRUE in green.

Step 1: Initialize low and high.

Like I said earlier, low and high will always be the same: low = 0, high = n – 1

```
int binarySearch(int[] a, int val)
{
    int n = a.length;
    int low = 0, high = n - 1;

    while (low < high)
    {
        int mid = (low + high) / 2;

        if (a[mid] >= val)
            high = mid;

        else
            low = mid + 1;
    }

    return low;

}
```

Step 2, condition for while loop:

The while loop condition here is (low < high). This is because we are not finding the exact element. And if we put low <= high, we will get stuck at an infinite loop at one point.

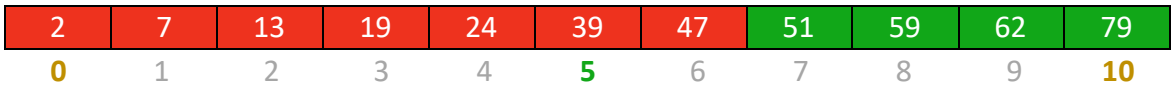
Step 3: The mid in this case will also be the same, i.e. (low + high) /2

Let us find the first occurrence of an element that evaluates to true (first element greater than or equal to 50 in our example)

Let that variable be named ‘val’.

Iteration 1:

low = 0, high = 10, mid = 5

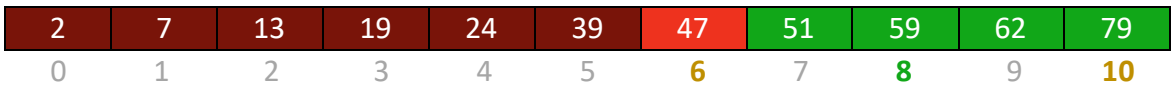


The condition check is (a[mid] >= val) which is FALSE, since 39 >= 50 == false

We move to the else condition and set the new low = mid + 1, because we know that 39, or any value below 39 will evaluate to FALSE. Thus, we eliminate those values.

Iteration 2:

low = 6, high = 10, mid = 8



/* The darkened portion is not there in our search space anymore */

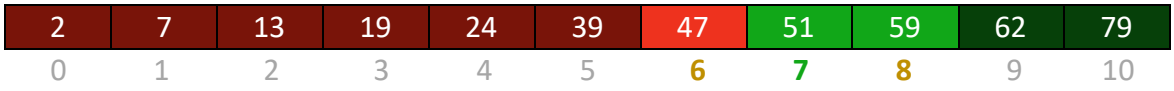
The first condition (a[mid] >= val) is TRUE. [59 >= 50]

Therefore, high = mid.

Why “mid” and not “mid – 1”? I told earlier that we need to minimize the search space as much as possible. Since the current value of ‘mid’ may possibly be the required answer, we can eliminate the portion of the array AFTER mid, but not mid itself.

Iteration 3:

low = 6, high = 8, mid = 7

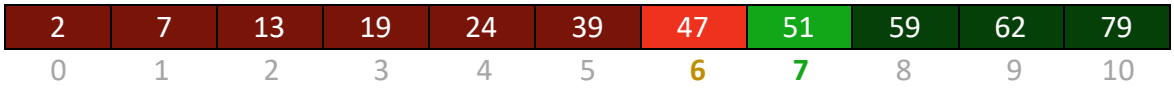


First condition is TRUE, so high = mid.

The explanation is again the same as the one I gave earlier.

Iteration 4:

low = 6, high = 7, mid = 6



First condition evaluates to FALSE, so we execute the instruction in the second line, i.e.

low = mid + 1

The iterations will stop as low < high no longer holds good.

The value in low = 7.

And remember this, the last value stored in low is the answer.

What if the entire array is filled with values that don't meet the condition?

In other words, the entire array is red in color. You will notice that the last value of low will be the last index. But that is also not the answer.

How we take care of that is just by putting an if condition before the return statement.

If (a[low] doesn't satisfy the condition)

    Throw an Error or something of that sort;

Else

    Return low;

Notice that in this type of binary search, there is no return statement inside the while loop unlike the binary search of the first type.

The return statement must be kept outside the while loop with a checker condition before returning it.

Aaaand we're done with the second type of binary search! The third is almost the same as this one. Just a few minor changes.

3. Searching for the last value that evaluates to TRUE:

Same array will be used for this example as well.

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	4	5	6	7	8	9	10

Let the question be, find the greatest value that is at most 50.

Making this array as a True/False array, we have:

2	7	13	19	24	39	47	51	59	62	79
0	1	2	3	4	5	6	7	8	9	10

The code for it is just like the previous one.

```
int binarySearch(int[] a, int val)
{
    int n = a.length;
    int low = 0, high = n - 1;

    while(low < high)
    {
        int mid = (low + high + 1) / 2;

        if(a[mid] <= val)
            low = mid;

        else
            high = mid - 1;

    }

    return low;
}
```

Here, if the current value of mid is a possible answer, we set 'low' as mid. Otherwise, high will become mid – 1 since the current value of mid is definitely not in the required search space.

NOTE: You can see that the value of mid is calculated as **(low + high +1) / 2** instead of **(low + high) / 2**. The reason for that is to avoid an infinite loop. To understand why, consider the question given above with a two-element array [48, 49]. In the first iteration, if we put mid = (low + high) / 2, mid will be index 0 which will evaluate the condition to true and reassign low to 0 again causing an infinite loop. Hence, we just do (low + high + 1) / 2 instead.

Instead of thinking about what equation to put for mid, just write the if-else statements. If you assign **low = mid**, only then will the mid be (low + high + 1) / 2. Otherwise, it is the normal equation.

I said that binary search can be applied to any monotonic function. I'll state a sentence to make things a little clearer.

If I can do **K** amount of work in less than time **T**, then any amount of work less than **K** can be done in time less than **T**. This statement refers to a monotonically increasing function.

If **M** people can do some amount of work in **T** time, then any number of people greater than **M** will take lesser or equal amount of time as **T**. This is a monotonically decreasing function.

Give the above two statements a thought and understand how they'd work.

Usually, our binary search won't be on plain arrays and our questions most likely won't be "find the certain element". The questions could be like: what is the minimum value where a certain condition 'x' is true? That question can be thought of "What is the first value where 'x' evaluates to true?" That is the second type of binary search which I explained.

Here's the pseudo code (for type 2):

```
int l = 0, r = LAST_VALUE_IN_SEARCH_SPACE; // r needs to be some decent approximation of last value

while (l < r)
{
    int mid = (l + r) / 2;

    if (someAlgorithmAssumingMidIsTheAnswer(min))
        r = mid; // mid can be a possible answer for the algorithm
    else
        l = mid + 1; // algorithm fails when mid is the answer
}

if (someAlgorithmAssumingMidIsTheAnswer(l))
{
    // answer will ALWAYS be stored in l if the answer is in the search space
    return l;
}

// if you reached here, the answer doesn't exist in search space
```

An example for a better understanding:

Q. Find floor($\sqrt{x}$)

Explanation: Find the last value y, such that y * y <= x

This is an example of type 3 binary search. You'll find things much simpler with the code below:

```
int l = 0, r = n;
// the reason for putting r = n is that sqrt(n) can't be > n.
// However, you can put 2*n, n^2 or anything as long as you definitely
// know that your answer will be in [l, r]

while (l < r)
{
    int mid = (l + r + 1) / 2;

    if (mid * mid <= n)
        l = mid;
    else
        r = mid - 1;
}

// because you're 100% sure that the answer will have existed in the initial
// search space, it's not necessary to put that additional if condition
// therefore we simply just
return l;


// lel
```

4. Binary Search for floating-point values:

Sometimes, our answer might be a floating-point value. The difficult part is we don't know when to terminate the loop. Might seem easy to just do: `while(abs(r - l) <= eps)` where **eps** is some small value which determines how accurate we want our answer.

That's perfectly fine. But I do it in a slightly different manner.

You already know by now that after each iteration in a binary search, the search space is halved. So, if I do **n** iterations, the search spaced is reduced by 2^n. Now suppose I want my answer's accuracy to be up to 10^{-6}, I'll find x which satisfies the equation:

$$\text{Length of Search Space} / 2^x <= 10^{-6}$$

Here, x is the number of iteration's I need to have an accuracy till 6 decimal places.

Finding this value x is essentially going to give you the minimum iterations you need. I usually just take x to be 80 or 100 based on the fact that my overall complexity of the entire solution isn't too high.

An example problem will clear this up for you:

Q. Find the square root of a number **n**.

```
double l = 0, r = n;
// since we're dealing with floating points, they're declared as double

for(int iters = 0; iters < 100; iters++)
{
    double mid = (l + r) / 2;

    if (l * l <= n)
        l = mid;
    else
        r = mid;
}

// since the difference between l and r isn't much, you can return
// either of them. But for the sake of consistency I'll just
return l;
```

That sums up all types of binary searches.

Just be sure to know when to use type 2 and type 3 binary searches. Especially when you find mid with (l + r) / 2 and when with (l + r + 1) / 2. Btw, these numberings of the searches are something which I've done. It's not standardized or anything.

You'll get used to it by practicing just a few questions of each type. But be sure that the function is definitely monotonic before proceeding.

PM in case you have any doubts. Coz that's the last line of every doc I've ever written.