Google Developer Student Clubs

Competitive Programming Series present

Binary Search

Plan

1. The idea of binary search
2. Implementation
3. Efficiency
4. Applications in problems

0 1 0 1 0 0 1
1 1 0 1 0 0 1
0 0 1 0 1 0 1

General problem

Given an array of **sorted** values we want to check if it contains a target value **x**

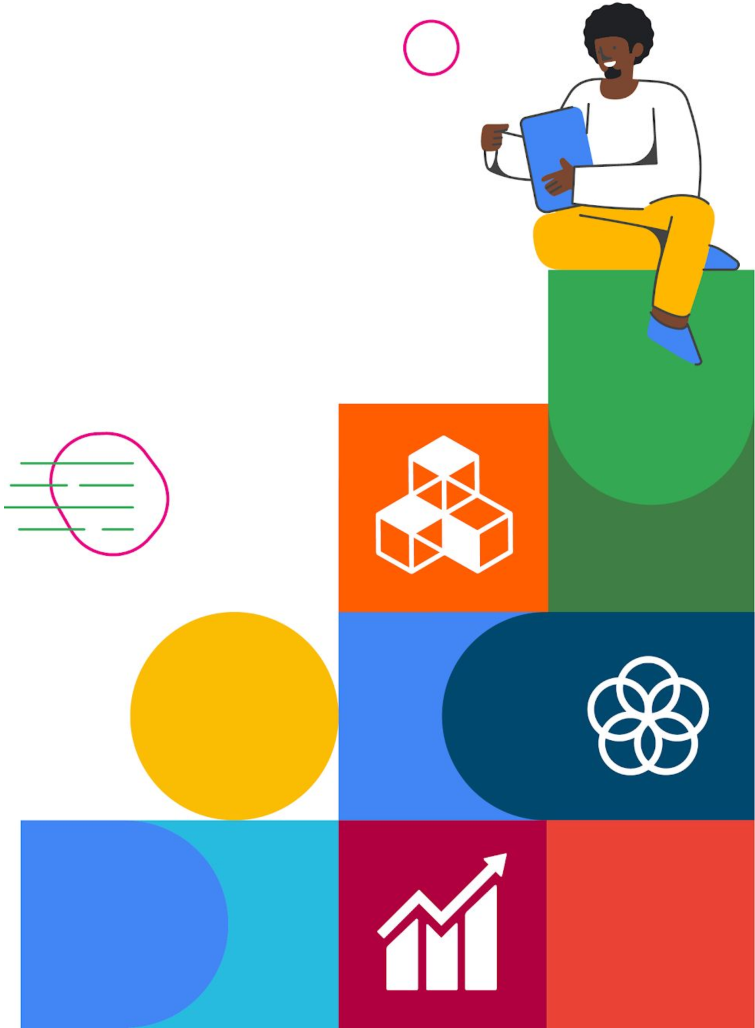
a = [1, 2, 4, 6, 10, 15, 17]

x = 4 *contains?* yes

x = 11 *contains?* no

First Idea: Linear Search

```
bool contains(int[] a, int n, int x) {
    for (int i = 0; i < n; i++) {
        if (a[i] == x) {
            return true;
        }
    }
    return false;
}
```

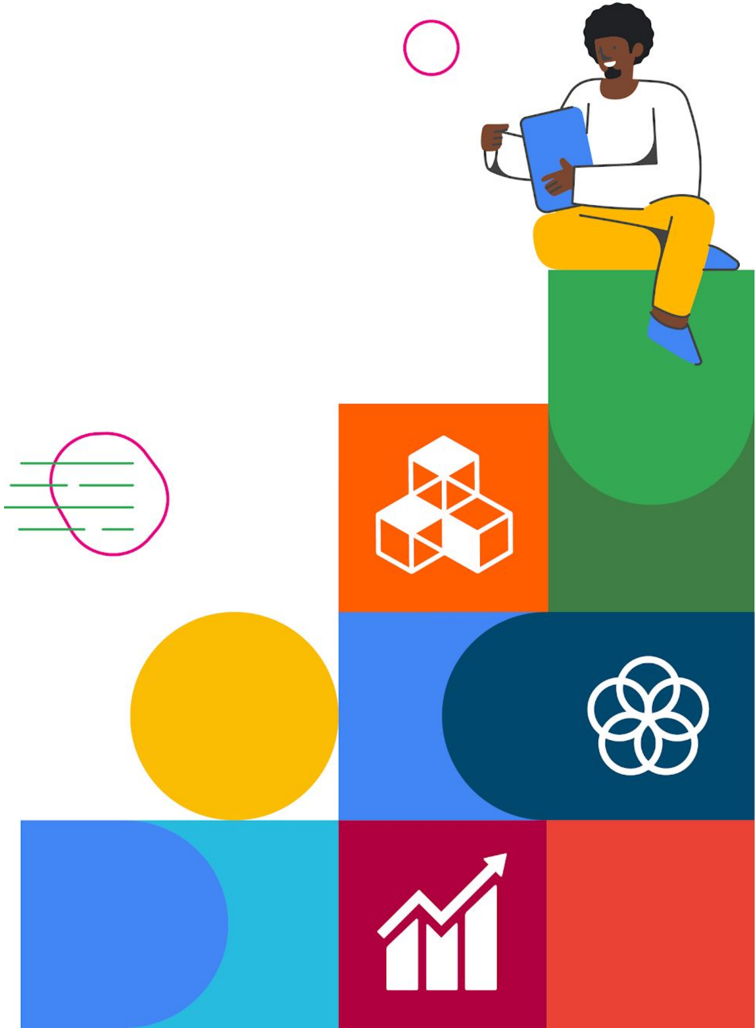


First Idea: Linear Search

```
bool contains(int[] a, int n, int x) {
    for (int i = 0; i < n; i++) {
        if (a[i] == x) {
            return true;
        }
    }
    return false;
}
```

Time complexity O(n)

Too slow for us!



Second Idea: Binary Search

Let’s compare the middle value of the array and **x**. We have 3 cases:

- **a[mid] == x** -> return true
- **a[mid] > x** -> repeat in left half
- **a[mid] < x** -> repeat in right half

Example

Initial values:

a = [1, 2, 4, 6, 10, 15, 17]

x = 4

Example

Initial values:

a = [1, 2, 4, 6, 10, 15, 17]

x = 4

First step:

a[mid] = 6 > 4 -> a = [1, 2, 4]

Example

Initial values:

a = [1, 2, 4, 6, 10, 15, 17]

x = 4

First step:

a[mid] = 6 > 4 -> a = [1, 2, 4]

Second step:

a[mid] = 2 < 4 -> a = [4]

Example

Initial values:

a = [1, 2, 4, 6, 10, 15, 17]
x = 4

First step:

a[mid] = 6 > 4 -> a = [1, 2, 4]

Second step:

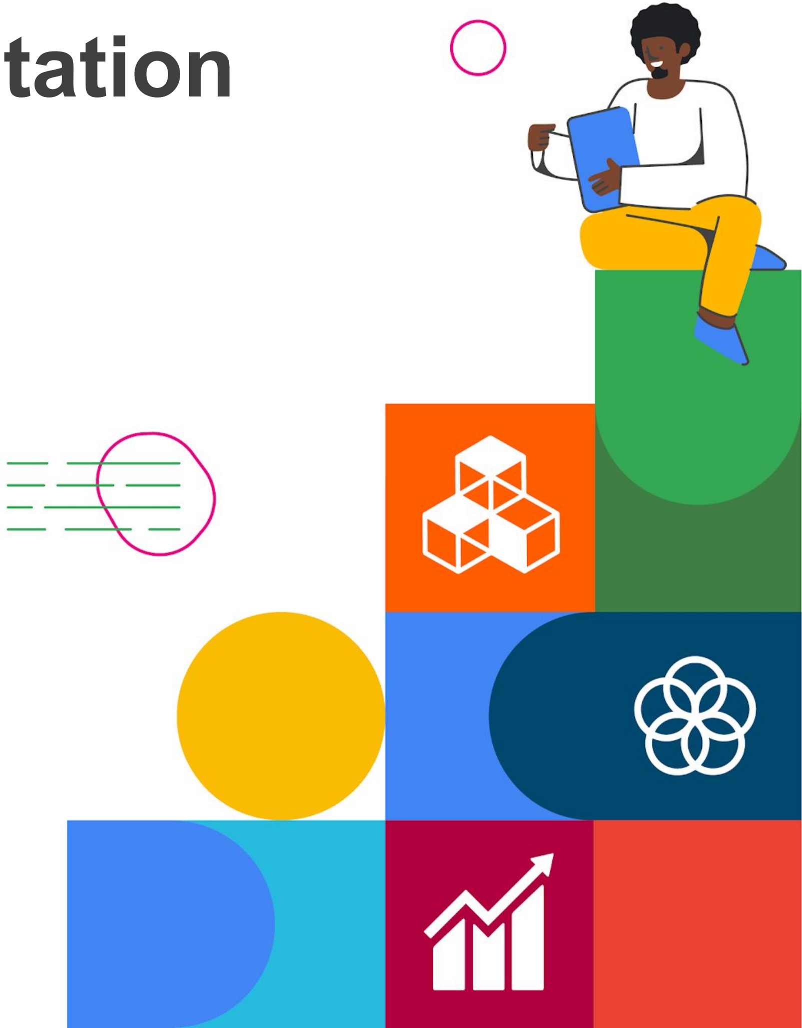
a[mid] = 2 < 4 -> a = [4]

Third step:

a[mid] = 4 == 4 -> return true

Binary Search: Implementation

```
bool contains(int[] a, int n, int x) {
    int l = 0;
    int r = n-1;
    while (l < r) {
        int mid = (l + r) / 2;
        if (a[mid] == x) {
            return true;
        }
        else if (a[mid] > x) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return a[l] == x;
}
```



Efficiency

On each step of the algorithm the length of the considered range (which is actually equal to r - l + 1) is reduced by half:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, **4, 5, 6, 7,** 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, **4, 5**, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, **5**, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Efficiency

On each step of algorithm the length of the considered range (which is actually equal to r - l + 1) is reduced by half:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, **4, 5, 6, 7,** 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, **4, 5**, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, **5**, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

=> O(log(n))

When to use binary search?

When to use binary search?

In fact, binary search can be applied whenever the array is **sorted.** However, you should always pay attention to the details such as:

- Initial values of l and r
- Exit condition in loop

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky

- Donald Knuth

Applications

Guess the number

How to guess a number from 1 to 100 if you can ask **YES/NO questions**?

What is the smallest number of questions you need?

What would your first question be?

Guess the number

Let’s try to use binary search idea.

If we start with a question “Is your number greater than 50?”, we will know that number is either in interval from 1 to 50 or from 51 to 100.

So one questions makes interval **twice smaller**!

Guess the number

```
int guess() {
    int l = 1;
    int r = 100;
    while (l < r) {
        int mid = (l + r) / 2;
        if (is_bigger_then(mid)) { //ask another player
            l = mid + 1;
        }
        else {
            r = mid;
        }
    }
    return l; // "return r" would also be fine
}
```

Square Root

Given an integer *x*, which is **a perfect square**, can you find it’s square root?

Of course, you CAN NOT use built in *sqrt* function. 😁

Square Root

Simple solution would be to every iterate over all numbers smaller than *x* and check them.

```c
int sqrt(int x) {
    int ans;
    for (int i = 1; i <= x; ++i) {
        if (i * i == x)
            ans = i;
    }
    return ans;
}
```

Square Root

This approach would be linear. Time complexity is O(x).

Can we do better?

Maybe binary search will again help us?

Square Root

Let's look at the following example:

$x = 16$

i : [1, 2, 3, **4**, 5, 6, 7, 8 … 16]

$i * i$: [1, 4, 9, **16**, 25, 36, 49, 64 … 256]

We can use binary search again!

0 1 0 1 0 0 1
1 1 0 1 0 0 1
0 0 1 0 1 0 1

Square Root

```
int sqrt(int x) {
    int l = 1, r = x;
    while (l < r) {
        int mid = (l + r) / 2;
        if (mid * mid == x) {
            return mid;
        }
        else if (mid * mid > x) {
            l = mid + 1;
        }
        else {
            r = mid - 1;
        }
    }
    return l;
}
```