

Searching Algorithms

Linear Search

Overview from Previous Classes:

Definition: Linear search (also known as sequential search) is a simple searching technique where each element in a list is checked one by one until the desired value is found or the list ends.

Suitable for **small** or **unsorted** datasets.

No issue if the **data is sorted or unsorted**.

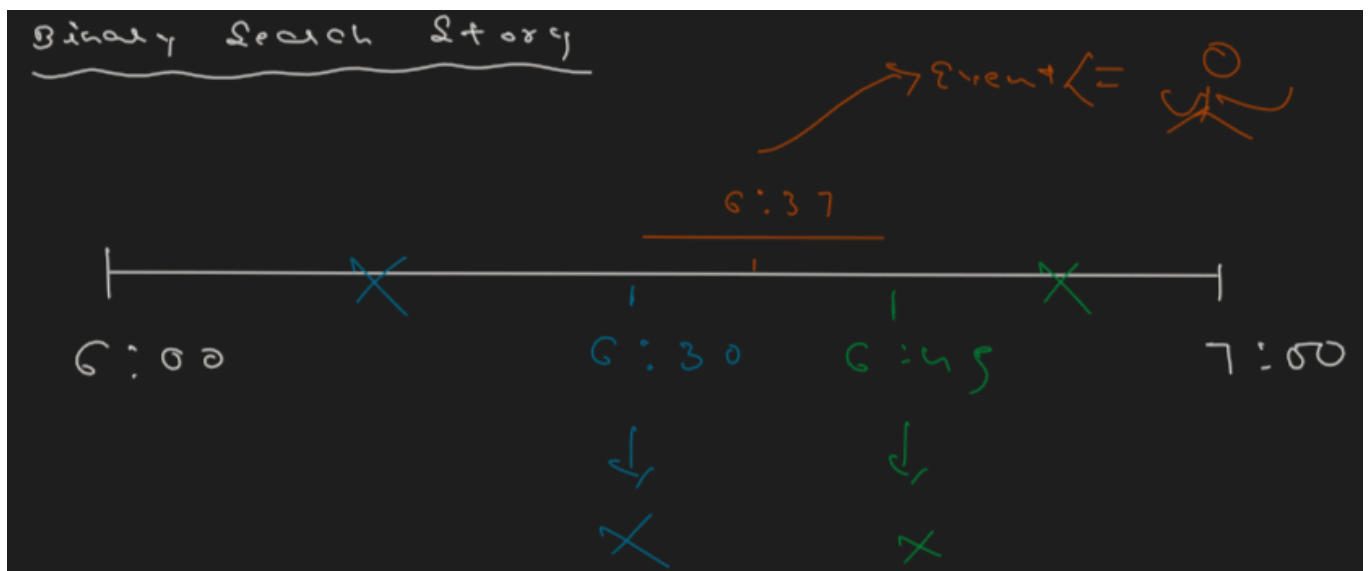
Binary Search

What does *binary* mean? If you remember from previous classes, we studied binary numbers—numbers that are either **0** or **1**, representing **false** or **true**. So basically, you just have **two options**, and you need to choose **one** of them.

If you apply this idea to real life, it's like having two choices: either you **study and do well in life**, or you **scroll through Reels all day**. You can't pick both—you have to choose one. That's what binary means: **two options, one decision**.

In **search algorithms**, this concept applies as well. At every step in **binary search**, you have **two directions** to go: left or right. You choose one direction based on the condition, and you follow that path.

Smart Use of Binary Search in Real Life



Imagine an incident occurred in our area between 6:00 and 7:00 p.m. I need to find the exact moment it happened by watching the CCTV footage—but I'm not as free as someone binge-watching *Panchayat*! I'm smart and efficient, so I'll use binary search.

First, I check the recording at 6:30 p.m. In India, a crowd usually gathers *after* an incident to discuss what happened. At 6:30, there's no crowd—just normalcy. That means nothing significant occurred between 6:00 and 6:30, so I discard that interval.

🔍 My new interval is from 6:30 to 7:00. I check the middle again—this time at 6:45 p.m. Now I see a huge crowd gathered. Clearly, the incident has already happened. Anything after 6:45 is just reaction, not the actual event. So I eliminate the interval from 6:45 to 7:00.

🕒 The narrowed window is 6:30 to 6:45. Once again, I go to the midpoint—6:37 p.m.—and I get lucky! I witness the actual incident: someone fires a gunshot. Found it!

This whole approach saves time and effort. Instead of watching the entire hour-long recording, I pinpointed the moment using binary search—just like how we divide and conquer in algorithms.

Definition: Binary search is an efficient algorithm for finding an element in a **sorted** list by repeatedly dividing the search range in half.

🍛 Shaadi Thali Analogy – Finding the Paneer Tikka

0 1 2 3 4 5 6 7 8 9 10 11 12 13

2	3	4	5	7	9	11	13	16	18	20	22	24	26
---	---	---	---	---	---	----	----	----	----	----	----	----	----

Buffet
↓
Spice level

Akash →

Hitesh → Paneer Tikka → 13

You're at a wedding, and the servers bring a massive steel thali with 14 tiny compartments (each filled with one dish). You've heard your favorite dish—**paneer tikka**—is somewhere in it. The compartments are already arranged from least spicy (left) to most spicy (right), matching these spice ratings:

[2, 3, 4, 5, 7, 9, 11, 13, 16, 18, 20, 22, 24, 26]

Paneer tikka has a **spice level of 13**.

Instead of tasting every item left to right like a hungry beginner, you're a smart foodie—you apply binary search:

1. First taste: go for the **middle compartment (index 6)** = spice level **11**. Hmm... it's spicy, but **paneer tikka's spicier**.

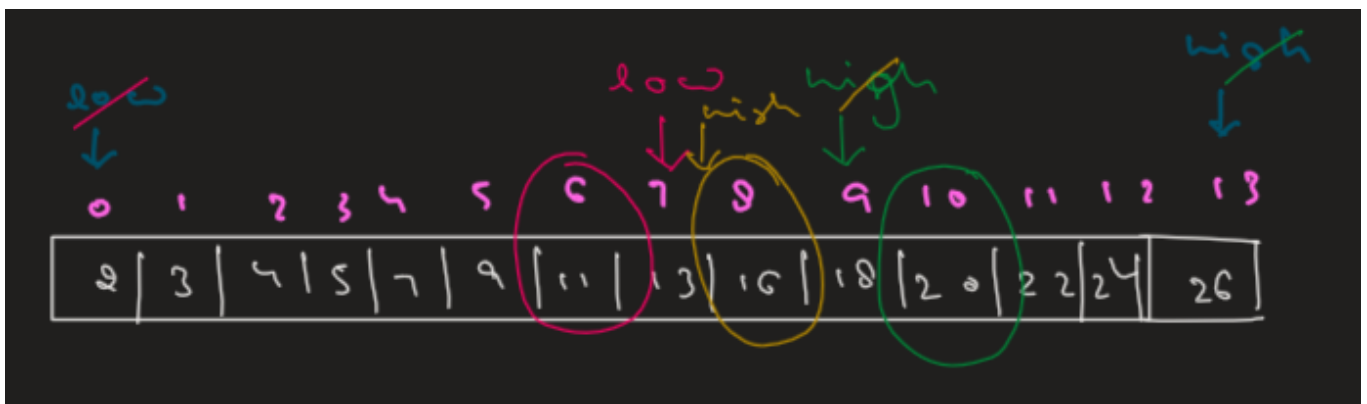
2. Move to the **right half** → now your range is: [13, 16, 18, 20, 22, 24, 26]
3. Next middle = index 9 → spice level **18** Whoa! Too spicy! Your tikka isn't that fiery → move **left**
4. Middle of new range: index 7 = **13** 🌟 Jackpot! That's your beloved **paneer tikka**.

You found it in just **3 bites**, not 14. Your stomach is grateful. That's binary search—**intelligent taste-testing for quick satisfaction**. 😊

Ex: Searching sorted pincodes by amazon delivery guy.

♦ Steps:

1. Find the middle element of the list.
2. If it matches the target, return the index.
3. If the target is smaller, repeat the search in the **left** half.
4. If the target is larger, repeat the search in the **right** half.
5. Continue until the element is found or the sublist is empty.



$$\text{mid} = \frac{(\text{low} + \text{high})}{2}$$

```

if (arr[mid] == key)
    return mid;
else if (arr[mid] > key) } → 1st array
    high = mid - 1;
else if (arr[mid] < key) } → 2nd array
    low = mid + 1;

```

low → ~~0~~ mid + 1 → 6 + 1 → 7
 high → ~~13~~ mid - 1 → 10 - 1 → ~~9~~
 key → **13**
 mid - 1 → 7

$$m = \frac{(0+13)}{2} = 6$$

$$\text{arr}[mid] < \text{key}$$

$$11 < 13$$

$$\text{low} = 7$$

$$\text{high} = 13$$

$$m = \frac{(7+13)}{2}$$

$$m = 10$$

$$\text{arr}[mid] > \text{key}$$

$$20 > 13$$

$$\text{low} = 7$$

$$\text{high} = 9$$

$$m = \frac{(7+9)}{2}$$

$$m = 8$$

$$\text{arr}[mid] > \text{key}$$

$$16 > 13$$

$$\text{low} = 7$$

$$\text{high} = 7$$

$$m = \frac{(7+7)}{2}$$

$$m = 7$$

$$\text{arr}[mid] = \text{key}$$

$$13 = 13$$

```
public class Main {



    public static void main(String[] args) {
        int[] arr = { 2, 3, 4, 5, 6, 7, 8, 11, 13, 16, 17, 18 };
        int item = 13;
        System.out.println(binarySearch(arr, item));
    }

    public static int binarySearch(int[] arr, int item) {
        int n = arr.length;
        int low = 0;
        int high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == item) {
                return mid;
            } else if (arr[mid] > item) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return -1;
    }



}
```

How to Identify When to Use Binary Search

✓ Ideal Conditions:

-  The list must be sorted in ascending or descending order.
-  You need fast and efficient search performance.

Ideal when:

-  Data is static (not changing frequently). (**Ex-** Air quality index)
-  Searching is more frequent than inserting or deleting. (**Ex-** Students record)

Find the Floor of the K-th Root

You are given two integers:

- n – a positive integer,
- k – another positive integer.

Your task is to **find the greatest integer x such that: $x^k \leq n$**

This means you're finding the **floor of the k -th root of n** .

✓ Explanation with Example:

$n = 149$, $k = 3$

You want to find the **largest integer x such that $x^3 \leq 149$** .

Let's evaluate:

- $5^3 = 125$ ✓
- $6^3 = 216$ ✗ (greater than 149)

So, the greatest integer x that satisfies $x^3 \leq 149$ is **5**, Therefore, the output of the code is: **5**

➡ Linear Search Solution

We try each number starting from **1** upwards, checking whether its k -th power is less than or equal to n . As soon as it exceeds n , we stop — and return the **last value** that worked.

```
public class Main {  
  
    public static void main(String[] args) {  
        int n = 149;  
        int k = 3;  
        System.out.println(kthRootLinear(n, k));  
    }  
  
    public static int kthRootLinear(int n, int k) {  
        int ans = 0;  
        for (int i = 1; i <= n; i++) {  
            double power = Math.pow(i, k);  
            if (power <= n) {
```

```

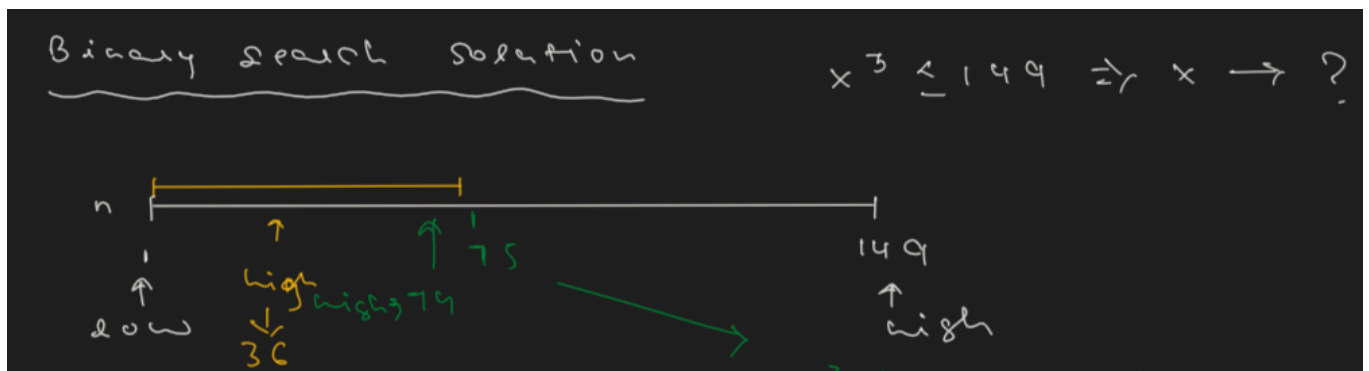
        ans = i;
    } else {
        break;
    }
}
return ans;
}
}

```

⚙️ Binary Search Solution

The function `kthRoot(int n, int k)` uses **binary search** in the range `[1, n]` to find the correct integer root efficiently.

- It keeps trying the middle value (`mid`) of the current range.
- If `midk ≤ n`, it moves the lower bound up (because a potentially larger answer might exist).
- Otherwise, it moves the upper bound down.
- It stores the last valid `mid` (i.e., `midk ≤ n`) in the variable `ans`.



```

if (midk ≤ n) {
    ans = mid;
    low = mid + 1;
}
else {
    high = mid - 1;
}

```

low = 1
high = ~~149~~ 36

$$l = 1 \quad m = \frac{(1 + 149)}{2} = 75$$

$$h = 149$$

$$75^3 \leq 149 \rightarrow \times$$

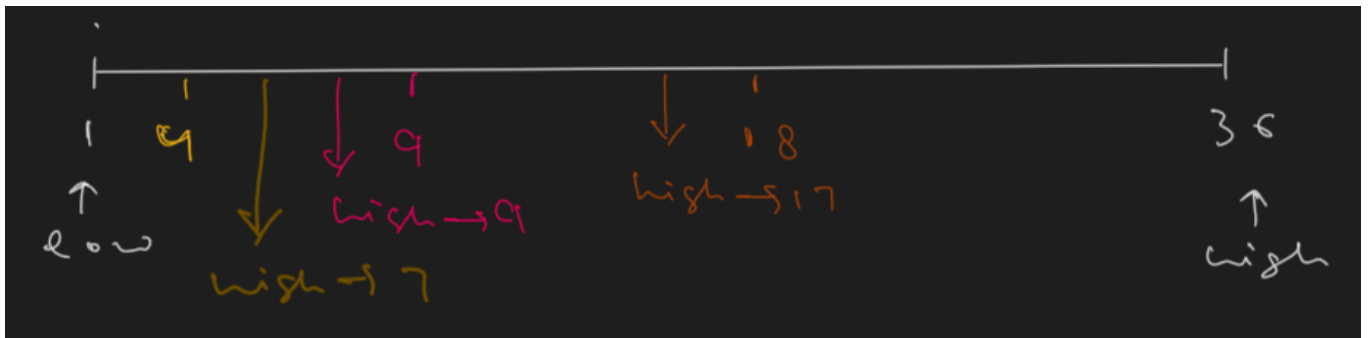
$$75^3 > 149 \rightarrow \checkmark$$

$$l = 1 \quad m = \frac{(1 + 74)}{2} = 37$$

$$h = 74$$

$$37^3 \leq 149 \rightarrow \times$$

$$37^3 > 149 \rightarrow \checkmark$$



$$l = 15$$

$$h = 36 \quad 17 \quad 18 \quad 5$$

$$a \sim 5 = 0 \quad 4 \quad 5$$

$$l = 1 \quad m = \frac{(1 + 36)}{2} = 18$$

$$h = 36$$

$$18^3 \leq 149 \rightarrow \times$$

$$18^3 > 149 \rightarrow \checkmark$$

$$l = 1 \quad m = \frac{(1 + 17)}{2} = 9$$

$$h = 17$$

$$9^3 \leq 149 \rightarrow \times$$

$$9^3 > 149 \rightarrow \checkmark$$

$$l = 1 \quad m = \frac{(1 + 8)}{2} = 4$$

$$h = 8$$

$$4^3 \leq 149 \rightarrow \checkmark$$

$$l = 5 \quad m = \frac{(5 + 8)}{2} = 6$$

$$h = 8$$

$$6^3 \leq 149 \rightarrow \times$$

$$6^3 > 149 \rightarrow \checkmark$$

$$l = 5 \quad m = \frac{(5 + 5)}{2} = 5$$

$$h = 5$$

$$5^3 \leq 149 \rightarrow \checkmark$$

```

public class Main {

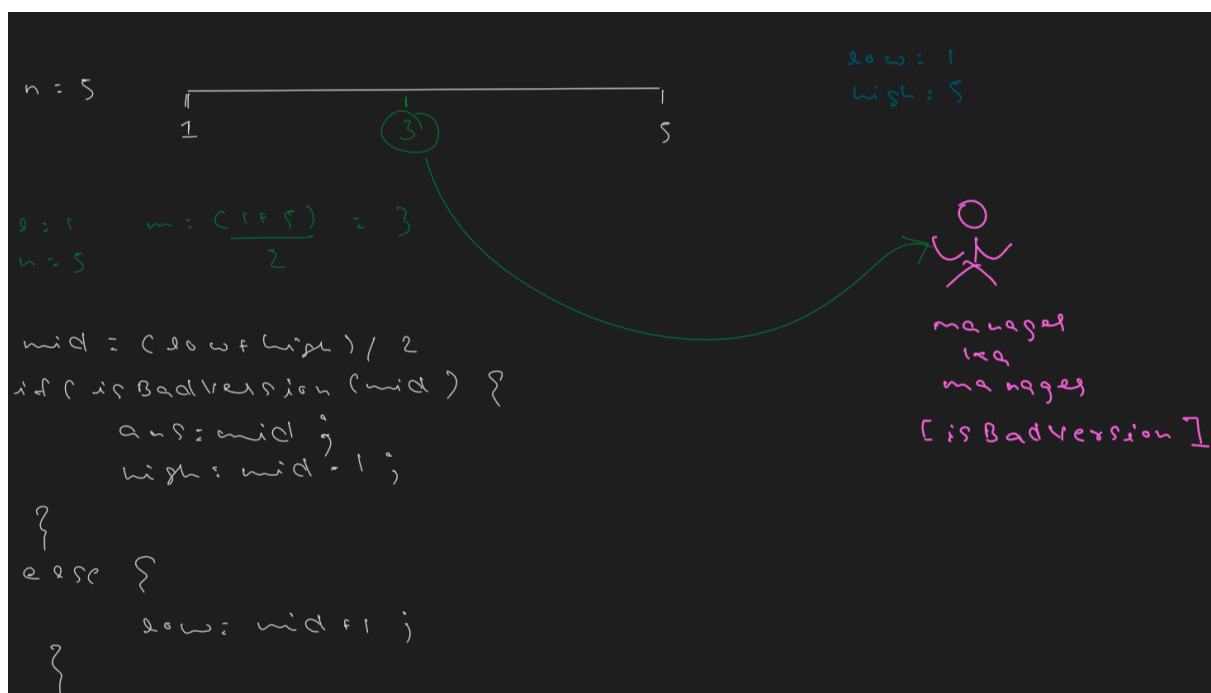
    public static void main(String[] args) {
        int n = 149;
        int k = 3;
        System.out.println(kthRootBinary(n, k));
    }

    public static int kthRootBinary(int n, int k) {
        int low = 1;
        int high = n;
        int ans = 0;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (Math.pow(mid, k) <= n) {
                ans = mid;
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return ans;
    }
}

```

🎯 First Bad Version

<https://leetcode.com/problems/first-bad-version/>




```

/* The isBadVersion API is defined in the parent class VersionControl.
   boolean isBadVersion(int version); */

public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int low = 1;
        int high = n;
        int ans = -1; // Variable to store the first bad version

        // Binary search to find the first bad version
        while (low <= high) {
            // int mid = (low + high) / 2; // Calculate the midpoint
            int mid = low + (high - low)/2; // Calculate the midpoint
            (efficiently)

            if (isBadVersion(mid)) {
                ans = mid; // mid is bad, so it could be the first bad version
                high = mid - 1; // Search left half to check if there's an
                earlier bad version
            } else {
                low = mid + 1; // mid is good, so the first bad version must be on
                the right
            }
        }

        return ans; // Return the first bad version found
    }
}

```

Cost to output

$$1 \leq \text{bad} \leq n \leq 2^{31} - 1$$

$$\text{max value of } n = 2^{31} - 1$$

$$l: 2^{31} - 6$$

$$h: 2^{31} - 1$$

$$\text{mid} = \frac{\text{low} + \text{high}}{2} = \frac{2^{31} - 6 + 2^{31} - 1}{2}$$

$$= \frac{2^{31} + 2^{31} - 7}{2}$$

$$2^2 + 2^2 = 2^3$$

$$\downarrow$$

$$2 \cdot (2^2) = 2^3$$

$$= \frac{2^{32} - 7}{2}$$

→ out of range

$$\text{mid} = \frac{\text{low} + (\text{high} - \text{low})}{2}$$

$$\frac{2 \cdot \text{low} + \text{high} - \text{low}}{2} = \frac{\text{low} + \text{high}}{2}$$

In binary search, we use the formula $mid = low + (high - low) / 2$ to safely calculate the middle index between `low` and `high`. This is an improved version of the simpler $mid = (low + high) / 2$, which can cause **integer overflow** if `low` and `high` are both large values (especially in programming languages like C++ or Java). By subtracting first $(high - low)$, we ensure the intermediate result stays within a safe range before adding it back to `low`.

Search in Rotated Sorted Array

<https://leetcode.com/problems/search-in-rotated-sorted-array/>