# Sort and search

IIT Ropar - Minor in AI

13th Feb, 2025

## 1 Efficient Book Lookup in a Digital Library

Imagine you are working in a a university digital library contains millions of books categorized by their ISBN numbers. Your task is implementing a system that efficiently retrieves book details based on ISBN queries. A naive linear search approach would be too slow, making the system inefficient for users.

### 1.1 Challenges

- The dataset is vast, containing millions of books.
- Users expect real-time search results.
- A traditional linear search would result in high computational time.
- The system needs to be scalable for future expansions.

### 1.2 Solution: Implementing Binary Search

Since the ISBN numbers are pre-sorted, Binary Search can be applied effectively. Instead of scanning each book sequentially, Binary Search divides the dataset into halves, reducing the search complexity. This optimization significantly improves query response time.

### 1.3 Implementation Steps

1. Ensure the book database is sorted by ISBN.
2. Use Binary Search to locate the desired ISBN efficiently.
3. Retrieve and display the book information to the user.

### 1.4 Results and Benefits

- Query response time improved significantly from linear to logarithmic scale.
- Enhanced user experience with faster search results.

- Scalability ensured as the system grows with new book additions.

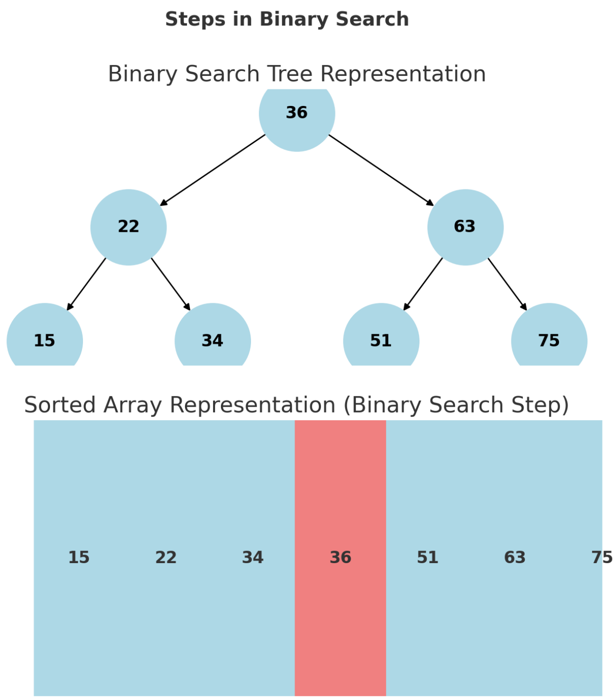


Figure 1: Binary Seacrh

## 2 Binary Search Code Implementation

Below is the Python implementation of Binary Search with detailed comments:

Listing 1: Binary Search

```
# Function to perform Binary Search
def binary_search(arr, target):
    """
    Performs binary search on a sorted array.
    :param arr: List of sorted elements
    :param target: The element to search for
    :return: Index of target if found, else -1
    """
    left, right = 0, len(arr) - 1  # Initialize search bounds

    while left <= right:
        mid = left + (right - left) // 2  # Find the middle index

        if arr[mid] == target:
```

```
            return mid  # Target found, return index
        elif arr[mid] < target:
            left = mid + 1  # Ignore left half
        else:
            right = mid - 1  # Ignore right half

    return -1  # Target not found

# Example usage:
data = [15, 22, 34, 36, 51, 63, 75]
target_value = 36
result = binary_search(data, target_value)
print("Element found at index:" if result != -1 else "Element not found", result)
```

## 2.1 Explanation of Code

- The function accepts a sorted list and a target value.
- It initializes two pointers, `left` and `right`, to mark the search boundaries.
- A `while` loop runs as long as `left` is less than or equal to `right`.
- The middle index `mid` is calculated to divide the search space.
- If the middle element matches the target, its index is returned.
- If the middle element is smaller, we move the left boundary to `mid + 1`.
- If the middle element is larger, we move the right boundary to `mid - 1`.
- If the loop terminates without finding the target, `-1` is returned.

# 3 Sorting Student Attendance Records

A school maintains attendance records of students in small-sized classes. The records are kept in an unordered list, and the administration needs a simple way to sort them based on roll numbers before generating reports.

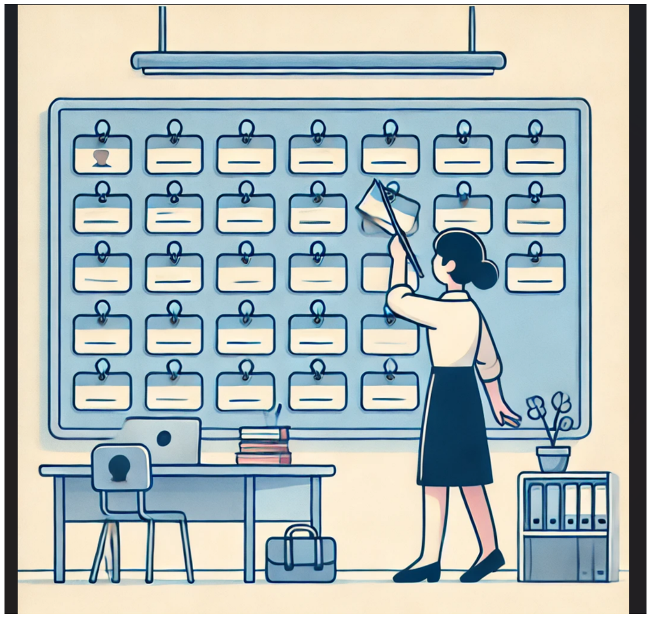

Figure 2: Sort Attendance records

### 3.1 Challenges

- The number of students per class is relatively small (typically 30-50).
- The sorting process needs to be implemented quickly without complex algorithms.
- The school's system has limited computational resources.
- Teachers without technical expertise should be able to understand and apply the sorting process.

### 3.2 Solution: Implementing Bubble Sort

Since the dataset is small, Bubble Sort is a viable choice due to its straightforward implementation. The algorithm repeatedly compares adjacent elements and swaps them if they are in the wrong order. Despite its $O(n^2)$ complexity, it performs adequately for small lists.

### 3.3 Implementation Steps

1. Retrieve the list of student attendance records.
2. Apply Bubble Sort to arrange the records by roll number.
3. Store the sorted records for generating reports.

## 3.4 Results and Benefits

- Simple implementation requiring minimal programming knowledge.
- Efficient sorting for small datasets without additional computational overhead.
- Easy to debug and modify if needed.
- Suitable for educational institutions with basic IT infrastructure.

# 4 Algorithm Process and Example

The Bubble Sort algorithm follows these steps:

1. Start from the first element.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Move to the next element and repeat steps 2 and 3 until the last element.
5. Repeat the process for $n - 1$ passes, where $n$ is the length of the list.

## 4.1 Example

Consider the unsorted list: $[5, 3, 8, 4, 2]$

## 4.2 Step-by-Step Execution

We perform multiple passes over the list, where in each pass, the largest unsorted element bubbles up to its correct position.

### 4.2.1 Pass 1

| | | | | |
|---|---|---|---|---|
| 5 | 3 | 8 | 4 | 2 |

1. Compare 5 and 3, swap: $\rightarrow [3, 5, 8, 4, 2]$
2. Compare 5 and 8, no swap: $\rightarrow [3, 5, 8, 4, 2]$
3. Compare 8 and 4, swap: $\rightarrow [3, 5, 4, 8, 2]$
4. Compare 8 and 2, swap: $\rightarrow [3, 5, 4, 2, 8]$

Total comparisons: 4

#### 4.2.2 Pass 2

1. Compare 3 and 5, no swap: $\rightarrow [3, 5, 4, 2, 8]$
2. Compare 5 and 4, swap: $\rightarrow [3, 4, 5, 2, 8]$
3. Compare 5 and 2, swap: $\rightarrow [3, 4, 2, 5, 8]$

Total comparisons: 3

#### 4.2.3 Pass 3

1. Compare 3 and 4, no swap: $\rightarrow [3, 4, 2, 5, 8]$
2. Compare 4 and 2, swap: $\rightarrow [3, 2, 4, 5, 8]$

Total comparisons: 2

#### 4.2.4 Pass 4

1. Compare 3 and 2, swap: $\rightarrow [2, 3, 4, 5, 8]$

Total comparisons: 1
After 4 passes, the list is sorted: $[2, 3, 4, 5, 8]$

## 5 Code Implementation and Explanation

Below is a Python implementation of the Bubble Sort algorithm with comments explaining each step:

Listing 2: Bubble Sort Algorithm

```python
def bubble_sort(arr):
    """
    Function to perform Bubble Sort on a given list.
    :param arr: List of elements to be sorted
    """
    n = len(arr)

    # Traverse through all elements in the list
    for i in range(n - 1):
        swapped = False  # Track if any swaps occur

        # Last i elements are already in place
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swap if
                    elements are in wrong order
                swapped = True

        # If no swaps occurred, the array is already sorted
        if not swapped:
            break
```

```python

# Example usage:
data = [5, 3, 8, 4, 2]
bubble_sort(data)
print("Sorted array:", data)
```

**Explanation:**

- The function takes an array as input and sorts it in ascending order.
- It iterates through the array multiple times.
- In each pass, adjacent elements are compared, and if needed, swapped.
- The process is optimized by introducing a `swapped` flag, which stops the algorithm early if the list becomes sorted before completing all passes.

# 6 Summary

Sorting algorithms are essential for efficient data retrieval and management in different applications. Binary Search is highly effective for large-scale applications like digital library systems, providing efficient, scalable, and real-time book retrieval to enhance performance and user satisfaction. On the other hand, Bubble Sort remains a practical choice for small-scale applications such as school attendance management, where ease of implementation is prioritized over efficiency. Both algorithms demonstrate the importance of selecting appropriate sorting techniques based on the specific needs of an application.