3. Binary Search

3.1 Concept

Binary Search is a fast search algorithm that works on sorted arrays by repeatedly dividing the search interval in half.

Prerequisites:

- Array must be sorted
- Random access to elements (arrays work well)

How it works:

1. Compare target with middle element
2. If match found, return position
3. If target is smaller, search left half
4. If target is larger, search right half
5. Repeat until found or search space is empty

Visual Representation:

```
Sorted Array: [10, 15, 25, 30, 35, 40, 50]
Target: 35

Step 1: Check middle (30)
[10, 15, 25, 30, | 35, 40, 50]
               ^
35 > 30, search right half

Step 2: Check middle of right half (40)
[35, 40, 50]
     ^
35 < 40, search left half

Step 3: Check middle of remaining (35)
[35]
 ^
35 = 35, Found at index 4!
```

3.2 Implementation

Iterative Binary Search

```
#include <stdio.h>

int binarySearch(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;  // Avoid overflow

        // Check if target is at mid
        if (arr[mid] == target) {
            return mid;
        }

        // If target is greater, ignore left half
        if (arr[mid] < target) {
            left = mid + 1;
        }
        // If target is smaller, ignore right half
        else {
            right = mid - 1;
        }
    }

    return -1;  // Element not found
}

int main() {
    int arr[] = {10, 15, 25, 30, 35, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 35;

    int result = binarySearch(arr, n, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
```

```
    } else {
        printf("Element %d not found\n", target);
    }

    return 0;
}
```

Why `mid = left + (right - left) / 2`?

```
// This can overflow for large values:
int mid = (left + right) / 2;

// This is safer:
int mid = left + (right - left) / 2;

// Example of overflow:
// left = 2^30, right = 2^30
// (left + right) = 2^31 → overflow in 32-bit int
// left + (right - left) / 2 = no overflow
```

Recursive Binary Search

```
int binarySearchRecursive(int arr[], int left, int right, int target) {
    // Base case: element not found
    if (left > right) {
        return -1;
    }

    int mid = left + (right - left) / 2;

    // Element found at mid
    if (arr[mid] == target) {
        return mid;
    }

    // Search in left half
    if (arr[mid] > target) {
        return binarySearchRecursive(arr, left, mid - 1, target);
    }
```

```
    // Search in right half
    return binarySearchRecursive(arr, mid + 1, right, target);
}

// Wrapper function
int binarySearch(int arr[], int n, int target) {
    return binarySearchRecursive(arr, 0, n - 1, target);
}
```

Binary Search with Comparison Count

```
int binarySearchCount(int arr[], int n, int target, int *comparisons) {
    int left = 0;
    int right = n - 1;
    *comparisons = 0;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        (*comparisons)++;

        if (arr[mid] == target) {
            return mid;
        }

        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
    int arr[] = {10, 15, 25, 30, 35, 40, 50, 60, 70, 80};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 35;
    int comparisons = 0;
```

```
    int result = binarySearchCount(arr, n, target, &comparisons);

    printf("Result: %d\n", result);
    printf("Comparisons: %d\n", comparisons);
    printf("Linear search would need: %d comparisons\n", result + 1);

    return 0;
}
```

3.3 Complexity Analysis

Case	Time Complexity	Description
Best Case	O(1)	Element found at middle
Average Case	O(log n)	Typical search
Worst Case	O(log n)	Element at edge or not found
Space Complexity	O(1) iterative, O(log n) recursive	Stack space for recursion

Comparison with Linear Search:

```
Array size: 1,000,000 elements

Linear Search:
- Average: 500,000 comparisons
- Worst: 1,000,000 comparisons

Binary Search:
- Average: 20 comparisons
- Worst: 20 comparisons

Speed improvement: ~50,000x faster!
```

Growth Comparison:

```
n       Linear Search    Binary Search
10      10               4
100     100              7
1,000   1,000            10
10,000  10,000           14
100,000 100,000          17
```

3.4 Binary Search Variations

Find First Occurrence

```
int findFirstOccurrence(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            result = mid;
            right = mid - 1;  // Continue searching in left half
        }
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return result;
}
```

Visual Example:

```
Array: [10, 20, 20, 20, 30, 40]
Target: 20

Regular binary search might return index 1, 2, or 3
First occurrence will always return index 1
```

Find Last Occurrence

```
int findLastOccurrence(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;
```

```
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            result = mid;
            left = mid + 1;  // Continue searching in right half
        }
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return result;
}
```

Count Occurrences

```
int countOccurrences(int arr[], int n, int target) {
    int first = findFirstOccurrence(arr, n, target);

    if (first == -1) {
        return 0;  // Element not found
    }

    int last = findLastOccurrence(arr, n, target);

    return last - first + 1;
}

int main() {
    int arr[] = {10, 20, 20, 20, 30, 40};
    int n = sizeof(arr) / sizeof(arr[0]);

    int count = countOccurrences(arr, n, 20);
    printf("20 occurs %d times\n", count);
```

```
    // Output: 20 occurs 3 times

    return 0;
}
```

Find Insert Position

```
int searchInsertPosition(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        }
        else if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }

    return left;  // Position where target should be inserted
}

int main() {
    int arr[] = {10, 20, 30, 50, 60};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Insert 25 at position: %d\n", searchInsertPosition(arr, n, 25));
    printf("Insert 35 at position: %d\n", searchInsertPosition(arr, n, 35));
    printf("Insert 70 at position: %d\n", searchInsertPosition(arr, n, 70));

    return 0;
}
```

3.5 When to Use Binary Search

Use Binary Search when:

- Data is sorted
- Need fast searching (large datasets)
- Data doesn't change frequently
- Random access is available (arrays)

Don't use Binary Search when:

- Data is unsorted (sorting overhead)
- Small datasets (linear search is simpler)
- Data changes frequently
- Sequential access only (linked lists)

Revision #1
Created 2025-11-03 02:02:36 UTC by DS
Updated 2025-11-03 02:02:55 UTC by DS