**EXPERIMENT NO. 1**

| **Objective(s):**  To develop efficient search algorithms that locate the position of a target value within a sorted array using binary search and interpolation search techniques. |
| --- |
| **Outcome:**  Implementation of binary search and interpolation search algorithms that can quickly find the index of a target element in a sorted array, improving search efficiency compared to linear search. |
| **Problem Statement:**  Implement binary search and interpolation search. |
| **Background Study:**  **1. Binary Search**  **Objective:** To locate the position of a target value within a sorted array efficiently by repeatedly dividing the search interval in half.  **How Binary Search Works**   * 1. **Initial Setup:**      + Start with two pointers: low at the beginning (index 0) and high at the end (index n-1) of the array.   2. **Midpoint Calculation:**      + Calculate the midpoint (mid) using the formula: mid = low + (high - low) / 2.   3. **Comparison:**      + Compare the target value with the element at the midpoint (array[mid]).        - If the target is equal to array[mid], return the index mid.        - If the target is less than array[mid], narrow the search to the left half by setting high = mid - 1.        - If the target is greater than array[mid], narrow the search to the right half by setting low = mid + 1.   4. **Repeat:**      + Repeat the process until low exceeds high or the target is found.      * **Time Complexity**    1. **Best Case:** ${O(1)}$, when the target is at the midpoint on the first check.   2. **Average and Worst Case:** ${O(log\ n)}$, because the search interval is halved in each step. * **Space Complexity**    1. **Iterative Version:** ${O(1)}$, as it uses a constant amount of space.   2. **Recursive Version:** ${O(log\ n)}$, due to the recursion stack. * **Use Cases**   1. Binary search is effective for searching in large sorted datasets where the cost of sorting the data is justified by the number of searches performed. |
| **2. Interpolation Search**  **Objective:** To improve the efficiency of search operations for uniformly distributed sorted arrays by estimating the position of the target value.  **How Interpolation Search Work**   1. **Initial Setup:**    * + Similar to binary search, start with two pointers: low at the beginning (index 0) and high at the end (index n-1) of the array.    1. **Position Calculation:**       * Calculate the estimated position (pos) of the target value using the formula:         + $pos = low + \frac{(target - array[low]) \times (high - low)}{array[high] - array[low]}$    2. **Comparison:**       * Compare the target value with the element at the estimated position (array[pos]).         + If the target is equal to array[pos], return the index pos.         + If the target is less than array[pos], narrow the search to the left side by setting high = pos - 1.         + If the target is greater than array[pos], narrow the search to the right side by setting low = pos + 1.    3. **Repeat:**       * Repeat the process until low exceeds high or the target is found.      * **Time Complexity**    1. **Best Case:** $O(1)$, when the target is at the estimated position on the first check.   2. **Average Case:** $O(\log \log n)$ for uniformly distributed data.   3. **Worst Case:** $O(n)$, when the distribution of elements is skewed. * **Space Complexity**    1. **Iterative Version:** $O(1)$, as it uses a constant amount of space.   2. **Recursive Version:** $(O(\log \log n))$, due to the recursion stack. * **Use Cases**    1. Interpolation search is particularly effective for large, uniformly distributed datasets where the values are spread evenly. * **Comparison: Binary Search vs. Interpolation Search**    1. **Binary Search:**      + Works well with any sorted dataset.      + Time complexity of \(O(\log n)\).      + Simple to implement.   2. **Interpolation Search:**      + Optimized for uniformly distributed data.      + Can achieve \(O(\log \log n)\) time complexity in the best case.      + Slightly more complex to implement due to the estimation formula. * **Practical Implementation Considerations**    1. **Data Distribution:**      + Use binary search for general-purpose searching in sorted arrays.      + Use interpolation search for specific cases where data is uniformly distributed.   2. **Array Size:**      + For small to moderately sized arrays, the difference in performance might be negligible.      + For very large arrays, interpolation search can provide significant performance improvements if the data distribution is suitable.   3. **Dynamic Data:**      + Both searches require the array to be sorted. Any dynamic updates (insertions or deletions) may necessitate re-sorting, impacting overall performance. * Understanding these search algorithms and their appropriate use cases ensures that you can select the most efficient algorithm for a given problem, optimizing search operations in various scenarios. |

| **Algorithm (Student Work Area):**  **Algorithms for Binary Search and Interpolation Search**  **1. Binary Search**  **Objective:** Efficiently locate the position of a target value within a sorted array.   * **Algorithm:**    1. **Initialization:**      + Set low to the starting index of the array (0).      + Set high to the ending index of the array (n-1).   2. **Loop:**      + While low is less than or equal to high:        1. Calculate the midpoint: mid = low + (high - low) / 2.        2. Compare the target value with array[mid]:           - If the target equals array[mid], return mid.           - If the target is less than array[mid], set high = mid - 1.           - If the target is greater than array[mid], set low = mid + 1.   3. **Completion:**      + If the loop ends without finding the target, return an indication that the target is not present (e.g., -1 or null).   **2. Interpolation Search**  **Objective:** Efficiently locate the position of a target value within a uniformly distributed sorted array.   * **Algorithm:**    1. **Initialization:**      + Set low to the starting index of the array (0).      + Set high to the ending index of the array (n-1).   2. **Loop:**      + While low is less than or equal to high and the target is within the range of array[low] to array[high]:        1. Estimate the position: pos = low + ((target - array[low]) \* (high - low)) / (array[high] - array[low]).        2. Compare the target value with array[pos]:           - If the target equals array[pos], return pos.           - If the target is less than array[pos], set high = pos - 1.           - If the target is greater than array[pos], set low = pos + 1.   3. **Completion:**      + If the loop ends without finding the target, return an indication that the target is not present (e.g., -1 or null). * These algorithms provide a structured approach to efficiently search for a target value in a sorted array, with binary search being more universally applicable and interpolation search offering potential performance benefits for uniformly distributed data. |
| --- |

| **Code:**  **Binary Search Implementation**  #include <stdio.h>  **// Function to perform binary search on a sorted array**  int binarySearch(int array[], int size, int target) {  int low = 0;  int high = size - 1;    while (low <= high) {  int mid = low + (high - low) / 2;    // Check if target is present at mid  if (array[mid] == target) {  return mid;  }  // If target is greater, ignore left half  else if (array[mid] < target) {  low = mid + 1;  }  // If target is smaller, ignore right half  else {  high = mid - 1;  }  }    // Target not found  return -1;  }  **// Example usage**  int main() {  int array[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};  int size = sizeof(array) / sizeof(array[0]);  int target = 23;    int result = binarySearch(array, size, target);    if (result != -1) {  printf("Element found at index %d\n", result);  } else {  printf("Element not found\n");  }    return 0;  }  **Interpolation Search Implementation**  #include <stdio.h>  **// Function to perform interpolation search on a sorted array**  int interpolationSearch(int array[], int size, int target) {  int low = 0;  int high = size - 1;    while (low <= high && target >= array[low] && target <= array[high]) {  // Estimate the position  int pos = low + ((double)(high - low) / (array[high] - array[low])) \* (target - array[low]);    // Check if target is present at pos  if (array[pos] == target) {  return pos;  }  // If target is greater, ignore left half  else if (array[pos] < target) {  low = pos + 1;  }  // If target is smaller, ignore right half  else {  high = pos - 1;  }  }    // Target not found  return -1;  }  **// Example usage**  int main() {  int array[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};  int size = sizeof(array) / sizeof(array[0]);  int target = 23;    int result = interpolationSearch(array, size, target);    if (result != -1) {  printf("Element found at index %d\n", result);  } else {  printf("Element not found\n");  }    return 0;  } |
| --- |
| **OUTPUT :** |