

SCS1308 Foundations of Algorithms

Tutorial - 07

Searching and sorting Algorithms.

Searching Algorithms

- Algorithms to retrieve information efficiently.
- Applications: Databases, file systems, web search engines, and AI.

Types of Searching Algorithms

1. Linear Search: Sequentially checks each element.
2. Binary Search: Divides search space in half.
3. Jump Search: Jumps fixed steps in sorted data.
4. Interpolation Search: Estimates the probable position.

01. Linear Search

- Sequential method.
- Time Complexity: $O(n)$.
- Used for unsorted/small datasets.
- Example: Searching a name in an attendance list.

Algorithm:

Step1: Start from the leftmost element of array and one by one compare x with each element of array.

Step2: If x matches with an element, return the index.

Step3: If x doesn't match with any of elements, return-1.

```
// C program to implement linear search using loop
#include <stdio.h>

int linearSearch(int* arr, int n, int key) {

    // Starting the loop and looking for the key in arr
    for (int i = 0; i < n; i++) {

        // If key is found, return key
        if (arr[i] == key) {
            return i;
        }
    }

    // If key is not found, return some value to indicate
    // end
    return -1;
}
```

02. Binary Search

- Works on sorted datasets.
- Time Complexity: $O(\log n)$.
- Example: Finding a word in a sorted dictionary.

Algorithm:

- Step1: Compare x with the middle element.
- Step2: If x matches with middle element, we return the mid index.
- Step3: Else If x is greater than the mid element, search on right half.
- Step4: Else If x is smaller than the mid element. search on left half.

```
// C Program to implement binary search using iteration
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int key) {

    // Loop will run till left > right. It means that there
    // are no elements to consider in the given subarray
    while (left <= right) {

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

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

        // If key greater than arr[mid], ignore left half
        if (arr[mid] < key) {
            left = mid + 1;
        }

        // If key is smaller than or equal to arr[mid],
        // ignore right half
        else {
            right = mid - 1;
        }
    }

    // If we reach here, then element was not present
    return -1;
}
```



03. Jump Search

- Optimized for sorted data.
- Time Complexity: $O(\sqrt{n})$.
- Example: Finding books in sorted library shelves.

Algorithm

- Step1: Calculate Jump size
- Step2: Jump from index i to index $i+jump$
- Step3: If $x == arr[i+jump]$ return x
Else jump back a step
- Step4: Perform linear search

```
#include<stdio.h>
#include<math.h>
int min(int a, int b){
    if(b>a)
        return a;
    else
        return b;
}
int jumpsearch(int arr[], int x, int n)
{
    // Finding block size to be jumped
    int step = sqrt(n);

    // Finding the block where element is
    // present (if it is present)
    int prev = 0;
    while (arr[min(step, n)-1] < x)
    {
        prev = step;
        step += sqrt(n);
        if (prev >= n)
            return -1;
    }
}
```

```
// Doing a linear search for x in block
// beginning with prev.
while (arr[prev] < x)
{
    prev++;

    // If we reached next block or end of
    // array, element is not present.
    if (prev == min(step, n))
        return -1;
}
// If element is found
if (arr[prev] == x)
    return prev;

return -1;
}
```



04. Interpolation Search

- Best for uniformly distributed sorted data.
- Time Complexity: $O(\log \log n)$.
- Example: Searching numeric keys in a database.

- Step1: In a loop, calculate the value of “pos” using the position formula.
- Step2: If it is a match, return the index of the item, and exit.
- Step3: If the item is less than arr[pos], calculate the position of the left sub-array. Otherwise calculate the same in the right sub-array.
- Step4: Repeat until a match is found or the sub-array reduces to zero.

$$pos = low + \left(\frac{\text{target} - \text{arr}[low]}{\text{arr}[high] - \text{arr}[low]} \times (high - low) \right)$$

- Target = element to be search
- arr[] - array
- Low - start arr[] index
- high - end arr[] index

```
// with recursion
#include <stdio.h>

// If x is present in arr[0..n-1], then returns
// index of it, else returns -1.
int interpolationSearch(int arr[], int lo, int hi, int x)
{
    int pos;
    // Since array is sorted, an element present
    // in array must be in range defined by corner
    if (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
        // Probing the position with keeping
        // uniform distribution in mind.
        pos = lo
            + (((double)(hi - lo) / (arr[hi] - arr[lo]))
            * (x - arr[lo]));
        // Condition of target found
        if (arr[pos] == x)
            return pos;

        // If x is larger, x is in right sub array
        if (arr[pos] < x)
            return interpolationSearch(arr, pos + 1, hi, x);

        // If x is smaller, x is in left sub array
        if (arr[pos] > x)
            return interpolationSearch(arr, lo, pos - 1, x);
    }
    return -1;
}
```



Sorting Algorithms

- Used to organize data for efficient searching and optimization.
- Types: Shell Sort, Radix Sort.

01. Shell Sort

- Compares distant elements first.
- Time Complexity: Best: $O(n \log n)$, Worst: $O(n^2)$.

```
// Shell Sort in C programming

#include <stdio.h>

// Shell sort
void shellSort(int array[], int n) {
    // Rearrange elements at each n/2, n/4, n/8, ... intervals
    for (int interval = n / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < n; i += 1) {
            int temp = array[i];
            int j;
            for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
                array[j] = array[j - interval];
            }
            array[j] = temp;
        }
    }
}

// Print an array
void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
}
```

02. Radix Sort

- Groups data by digits or keys.
- Time Complexity: $O(d \times (n + k))$.
- Example: Sorting ZIP codes, phone numbers.

```
// Radix Sort in C Programming
#include <stdio.h>
// Function to get the maximum value in the array
int getMax(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

```
// Using counting sort to sort the elements based on significant places
void countingSort(int array[], int n, int place) {
    int output[n];
    int count[10] = {0};
    // Calculate count of elements
    for (int i = 0; i < n; i++) {
        int index = (array[i] / place) % 10;
        count[index]++;
    }
    // Calculate cumulative count
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    // Place the elements in sorted order
    for (int i = n - 1; i >= 0; i--) {
        int index = (array[i] / place) % 10;
        output[count[index] - 1] = array[i];
        count[index]--;
    }
    // Copy the sorted elements into original array
    for (int i = 0; i < n; i++) {
        array[i] = output[i];
    }
}
```

```
// Main function to implement radix sort
void radixSort(int array[], int n) {
    // Get maximum element
    int maxElement = getMax(array, n);
    // Apply counting sort to sort elements based on place value
    for (int place = 1; maxElement / place > 0; place *= 10) {
        countingSort(array, n, place);
    }
}
```