SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

COURSE NAME : 23ITT101- PROBLEM SOLVING & C PROGRAMMING

I YEAR /I SEMESTER

Unit II – ARRAYS AND STRINGS

Topic : Sorting and Searching

Sorting

- Sorting is the process of arranging elements of a collection (such as an array or list) in a particular order, typically in ascending or descending order.
- Sorting is a fundamental operation in computer science and is used in various applications such as searching, data compression, and optimizing algorithms.

Common Sorting Techniques

- Bubble Sort: Simple but inefficient for large datasets. It compares adjacent elements and swaps them if necessary, pushing the largest elements to the end. Not efficient for large datasets due to $O(n^2)$ time complexity.
- Selection Sort: Similar to Bubble Sort in time complexity. It repeatedly finds the minimum element in the unsorted part and places it in the correct position. It is also inefficient with $O(n^2)$ time complexity.

Common Sorting Techniques

- Insertion Sort: Insertion Sort is much better for small arrays or nearly sorted arrays. It works by building the sorted portion of the array one element at a time.
- Merge Sort: A divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges them back. It performs consistently well with O(n log n) time complexity, but it requires additional space.

Common Sorting Techniques

- Quick Sort: Another divide-and-conquer algorithm that works by selecting a "pivot" element and partitioning the array into two parts: one less than the pivot and one greater. It is very efficient in practice but can degrade to $O(n^2)$ in the worst case.
- Heap Sort: A comparison-based sorting algorithm that builds a max-heap (or min-heap) and repeatedly extracts the largest (or smallest) element to build the sorted array. It is efficient with O(n log n) time complexity and uses constant space.

Common Sorting Techniques

- Radix Sort: A non-comparative sorting algorithm that sorts numbers digit by digit. It works well when sorting integers or strings. The time complexity depends on the number of digits (k) in the largest number and the size of the input (n).
- Counting Sort: Works well for integers or categorical data within a limited range. It counts the occurrences of each element and uses these counts to determine the correct positions of elements. It is stable and very efficient when the range of data (k) is small relative to the number of elements (n).

Common Sorting Techniques

- Bucket Sort: Divides the range of input values into a set of "buckets," sorts each bucket individually, and then combines the results. It is efficient when the input is uniformly distributed.
- Shell Sort: An extension of insertion sort that allows elements to be swapped even if they are far apart. The algorithm gradually reduces the gap between elements being compared.

Example : Bubble sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

```
Original array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

Choosing the Right Sorting Algorithm

- For small arrays or nearly sorted data, Insertion Sort or Bubble Sort might work well.
- For large datasets, Merge Sort, Quick Sort, or Heap Sort are more efficient.
- For sorting integers with a limited range, Counting Sort, Radix Sort, or Bucket Sort may be the best options.

Searching

- Searching techniques are algorithms used to find a specific element or value in a collection, such as an array, list, or database. There are several different searching techniques, and the choice of which one to use depends on the characteristics of the data being searched (e.g., whether it is sorted or unsorted, the size of the dataset, etc.).

Common searching techniques

- Linear Search

Best for: Unsorted data or when you need to check every element.

Algorithm: Starts from the first element and checks each one sequentially until the target element is found or the end of the collection is reached.

- Binary Search

Best for: Sorted data.

Algorithm: Divides the search interval in half. If the value of the target is less than the element in the middle, the search continues in the lower half. If the target is greater, the search continues in the upper half.

Common searching techniques

- Jump Search

Best for: Sorted data.

Algorithm: Jumps ahead by a fixed block size, and when the target is found, a linear search is performed to find the exact position.

- Exponential Search

Best for: Sorted data, especially large datasets.

Algorithm: First checks an exponentially increasing interval. Once a range is found, binary search is used within that range.

Common searching techniques

- Interpolation Search

Best for: Sorted arrays with uniformly distributed numeric data.

Algorithm: An extension of binary search that uses a heuristic to calculate the "mid" value based on the data values.

- Fibonacci Search

Best for: Sorted data.

Algorithm: Similar to binary search, but uses Fibonacci numbers to divide the array, making it more suitable for certain types of data.

Example: Linear search

```
#include <stdio.h>

int main() {
    int arr[] = {12, 34, 54, 2, 3};  // Sample array
  // Calculate the number of elements
  int n = sizeof(arr) / sizeof(arr[0]);
  int target = 54;  // Element to search for
    int i;
    int found = -1;  // Flag to indicate if the element is found

    // Linear search implementation in main
    for (i = 0; i < n; i++) {
        if (arr[i] == target) {
            found = i;  // Element found, store index
            break;  // Exit loop once the element is found
        }
    }
```

```
    // Output the result
    if (found != -1) {
     // Print the index
   printf("Element found at index %d\n", found);
} else {
      // If not found, print a message
  printf("Element not found\n");
}

    return 0;
}
```

Output:

Element found at index 2

Choosing the Right Search Algorithm

- Linear Search: Best for small or unsorted datasets where the overhead of sorting is not justifiable.
- Binary Search: Best for sorted datasets where fast searching is required.
- Jump Search: An improvement over linear search, works well for sorted data where you can divide the search into blocks.

Choosing the Right Search Algorithm

- Exponential Search: Effective for large sorted arrays when you don't know the bounds of the dataset.
- Interpolation Search: Best for uniformly distributed data where values are spread evenly.
- Fibonacci Search: Similar to binary search but uses Fibonacci numbers, and is efficient for large sorted datasets.


Thank
you