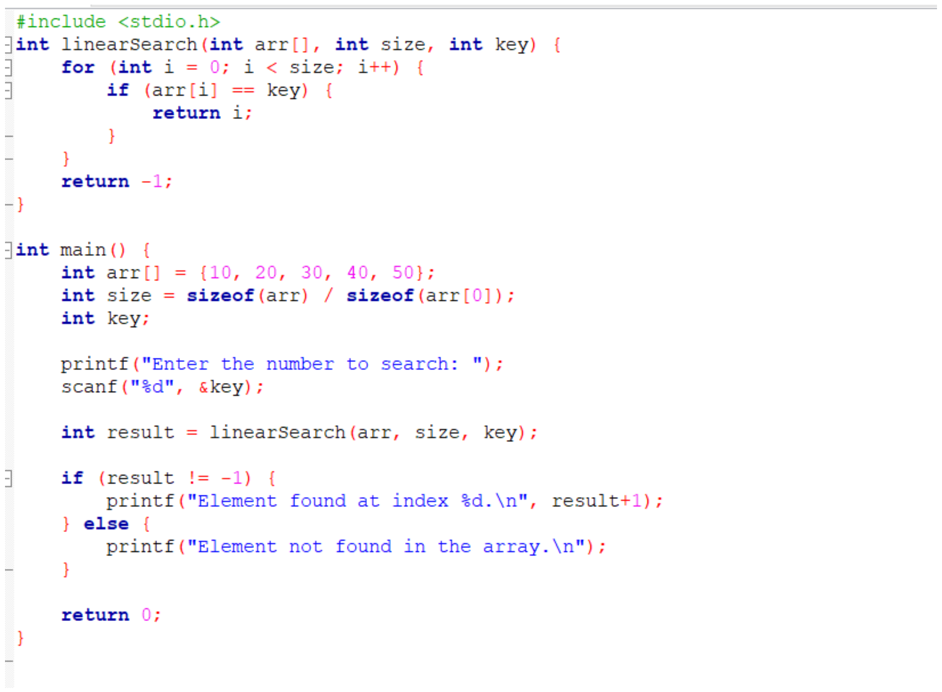
Linear Search

Introduction

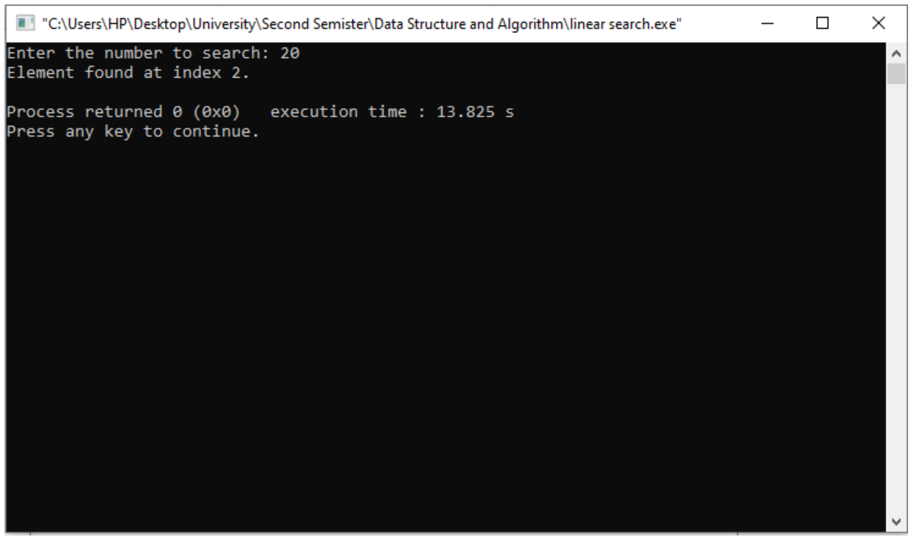
Linear Search is a straightforward technique that examines each item in a list sequentially to locate a specific value. It operates on unsorted data, making it versatile but less optimal for large datasets due to its O(n) time complexity. Simple and intuitive, it's best suited for smaller collections or scenarios requiring minimal setup.

Code:



```c
#include <stdio.h>
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the number to search: ");
    scanf("%d", &key);

    int result = linearSearch(arr, size, key);

    if (result != -1) {
        printf("Element found at index %d.\n", result+1);
    } else {
        printf("Element not found in the array.\n");
    }

    return 0;
}
```

Output:


"C:\Users\HP\Desktop\University\Second Semister\Data Structure and Algorithm\linear search.exe"

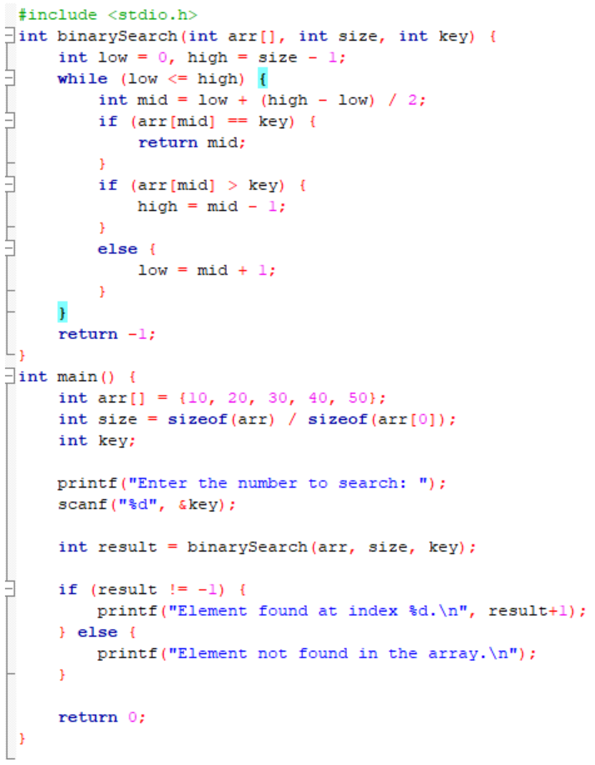
```
Enter the number to search: 20
Element found at index 2.

Process returned 0 (0x0)   execution time : 13.825 s
Press any key to continue.
```

BINATY SEARCH

Introduction

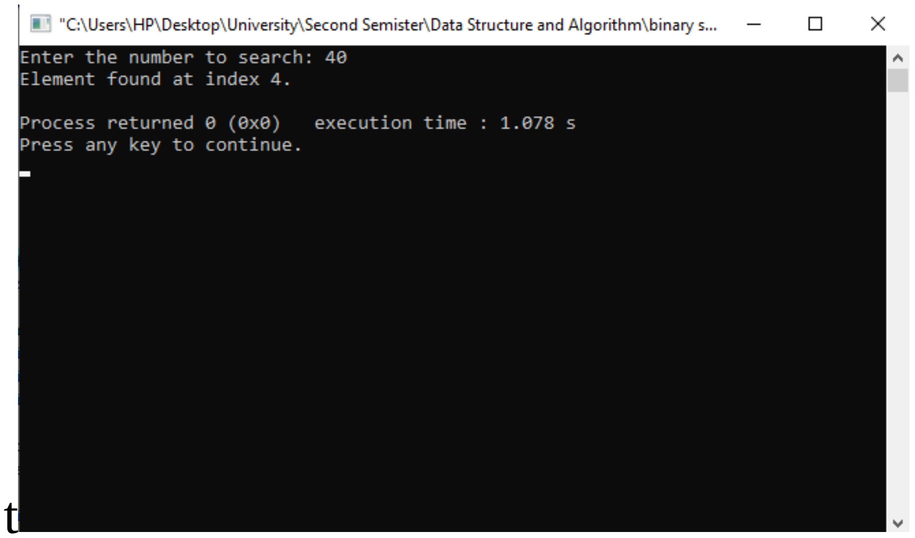
Binary Search is an efficient algorithm to find an element in a sorted list. It works by repeatedly dividing the search interval in half, comparing the target with the middle element, and narrowing the range based on the comparison. With a time complexity of O(log n), it's faster than linear search for large, sorted datasets.

Code



```c
#include <stdio.h>
int binarySearch(int arr[], int size, int key) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) {
            return mid;
        }
        if (arr[mid] > key) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return -1;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the number to search: ");
    scanf("%d", &key);

    int result = binarySearch(arr, size, key);

    if (result != -1) {
        printf("Element found at index %d.\n", result+1);
    } else {
        printf("Element not found in the array.\n");
    }

    return 0;
}
```

Output



"C:\Users\HP\Desktop\University\Second Semister\Data Structure and Algorithm\binary s...

```
Enter the number to search: 40
Element found at index 4.

Process returned 0 (0x0)   execution time : 1.078 s
Press any key to continue.
```

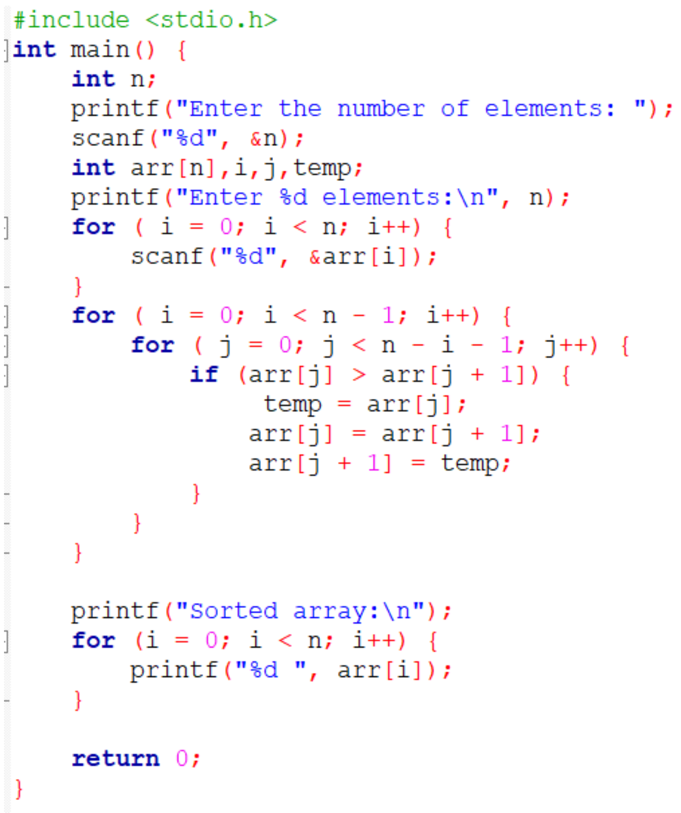
t

Bubble Sort

Introduction

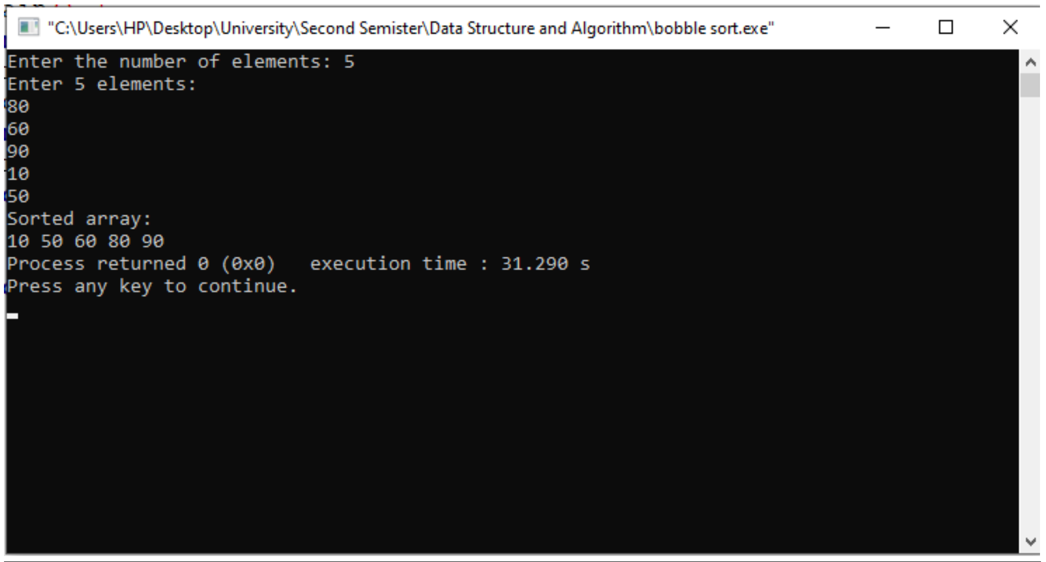
Bubble Sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is fully sorted. It has a time complexity of $O(n^2)$ and is best suited for small datasets or when simplicity is more important than efficiency.

Code:



```c
#include <stdio.h>
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n],i,j,temp;
    printf("Enter %d elements:\n", n);
    for ( i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    for ( i = 0; i < n - 1; i++) {
        for ( j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                 temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Code:



"C:\Users\HP\Desktop\University\Second Semister\Data Structure and Algorithm\bobble sort.exe"

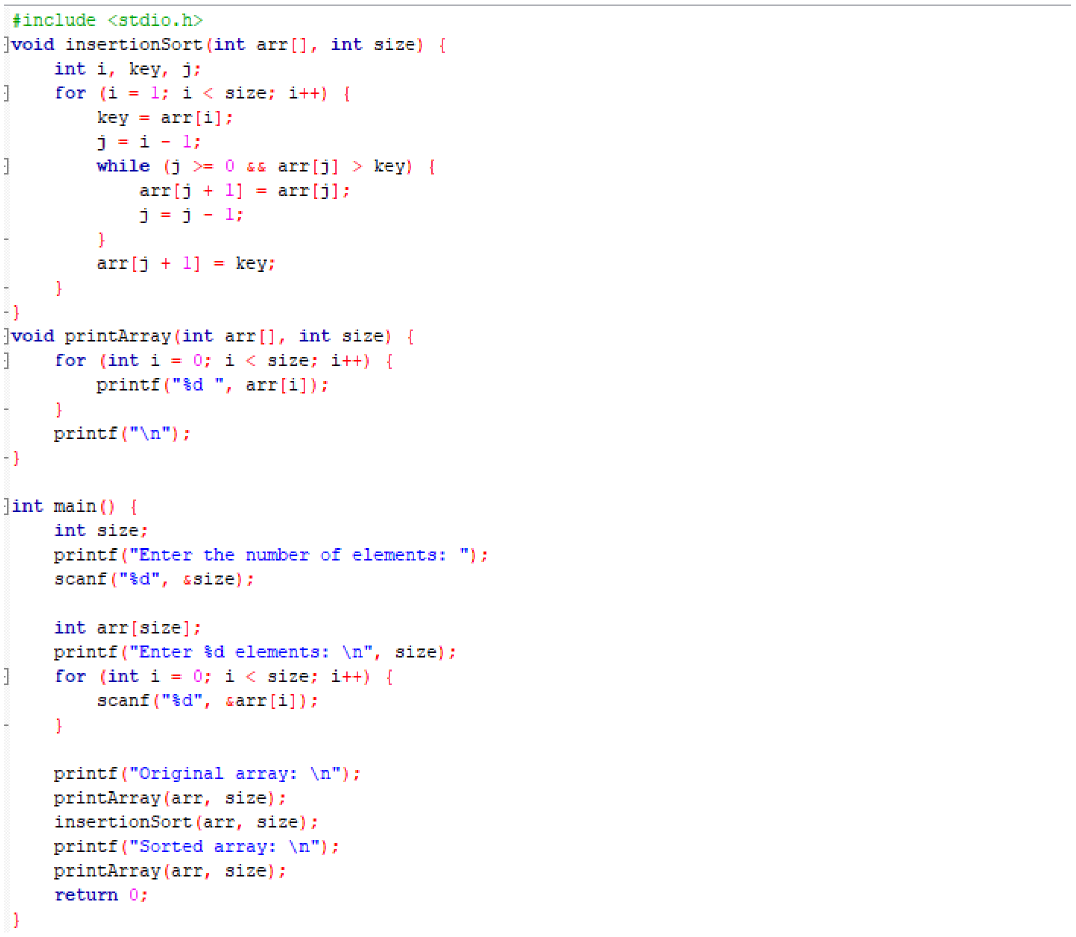
```
Enter the number of elements: 5
Enter 5 elements:
80
60
90
10
50
Sorted array:
10 50 60 80 90
Process returned 0 (0x0)   execution time : 31.290 s
Press any key to continue.
```

Insertion Sort

Introduction

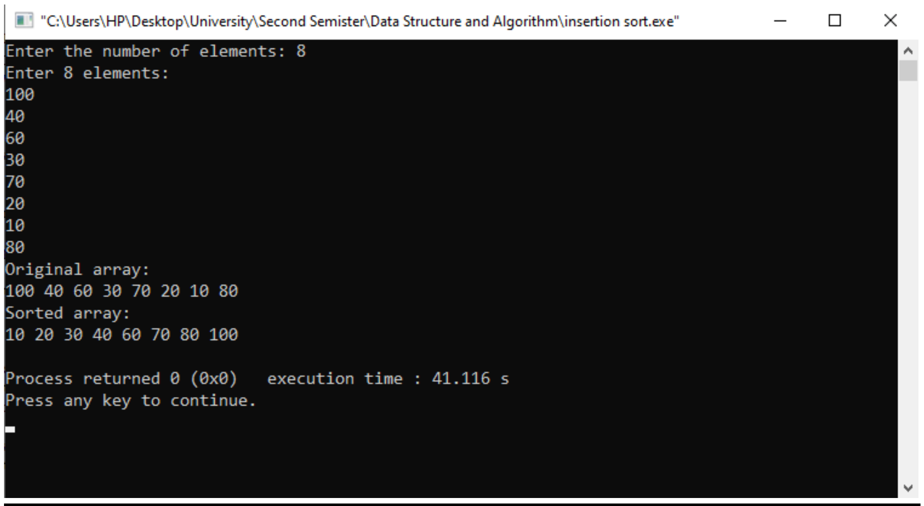
Insertion Sort is a simple algorithm that sorts a list by dividing it into sorted and unsorted parts. Elements from the unsorted section are picked and inserted into the correct position in the sorted section. It's efficient for small or nearly sorted datasets, with a worst-case time complexity of $O(n^2)$.

Code:



```c
#include <stdio.h>
void insertionSort(int arr[], int size) {
    int i, key, j;
    for (i = 1; i < size; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int size;
    printf("Enter the number of elements: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter %d elements: \n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original array: \n");
    printArray(arr, size);
    insertionSort(arr, size);
    printf("Sorted array: \n");
    printArray(arr, size);
    return 0;
}
```

Output:



"C:\Users\HP\Desktop\University\Second Semister\Data Structure and Algorithm\insertion sort.exe"

```
Enter the number of elements: 8
Enter 8 elements:
100
40
60
30
70
20
10
80
Original array:
100 40 60 30 70 20 10 80
Sorted array:
10 20 30 40 60 70 80 100

Process returned 0 (0x0)   execution time : 41.116 s
Press any key to continue.
```