UNIVERSITY OF KARACHI
DEPARTMENT OF COMPUTER SCIENCE
UBIT

PROJECT TITLE:
LINEAR SEARCH ALGORITHM

SUBMITTED BY:

TAHRIM BILAL (B21110006I53)

YUMNA MUBEEN (B21110006165)

SUBMITTED TO:

SIR MUHAMMAD SAEED

Algorithm: Linear Search

---

1. Selected Algorithm: Linear Search

Linear Search is a basic searching algorithm that sequentially checks each element of an array until the target element is found or the end of the array is reached.

---

2. Mathematical Model

Let:

- **A** = $[a_1, a_2, ..., a_n]$ be an array of n elements
- **x** = the element to search
- Function:
  LinearSearch(A, x) = *i*, where $A[i] = x$, if $1 \leq i \leq n$; else return -1

Mathematically,

$\exists\ i \in [1, n]$ such that $A[i] = x \Rightarrow$ return i;
If no such i exists, return -1

---

3. Explanation of the Model

- The algorithm begins at the start of the array and compares each element with the target x.
- If a match is found, the index is returned.
- If no match is found after scanning the full array, the function returns -1.
- The average and worst-case time complexity is **O(n)**.
- The best time and space complexity is **O(1).**

---

4. Application Domain

Linear Search is used when:

- The data is **unsorted**.
- The dataset is **small** or performance is not a critical concern.
- Memory overhead needs to be **minimal**.

5. Domains Where Linear Search is Used

Linear Search is commonly used in:

- **Embedded Systems**: Simple logic with low memory constraints.
- **IoT Devices**: Lightweight search operations.
- **Educational Tools**: For teaching search algorithms.
- **Text Processing**: Substring search in unsorted text chunks.
- **Database Queries**: Small unsorted datasets.

---

6. Implementations

Sequential Implementation (C++)

```cpp
#include <iostream>
using namespace std;

int linearSearch(int A[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (A[i] == x)
            return i;
    }
    return -1;
}

int main() {
    int A[] = {4, 2, 7, 9, 5, 8, 3};
    int n = sizeof(A) / sizeof(A[0]);
    int x = 7;

    int result = linearSearch(A, n, x);
    if (result != -1)
        cout << "Element " << x << " found at index " << result << endl;
    else
        cout << "Element " << x << " not found" << endl;

    return 0;
}
```

Output:

```
Element 7 found at index 2
```

Parallel Implementation (Using OpenMP in C++)

```
#include <iostream>
#include <omp.h>
using namespace std;

int parallelLinearSearch(int A[], int n, int x) {
    int index = -1;

    #pragma omp parallel for shared(index)
    for (int i = 0; i < n; i++) {
        if (A[i] == x) {
            #pragma omp critical
            {
                if (index == -1 || i < index)
                    index = i;
            }
        }
    }
    return index;
}

int main() {
    int A[] = {4, 2, 7, 9, 5, 8, 3};
    int n = sizeof(A) / sizeof(A[0]);
    int x = 7;

    int result = parallelLinearSearch(A, n, x);
    if (result != -1)
        cout << "Element " << x << " found at index " << result << endl;
    else
        cout << "Element " << x << " not found" << endl;

    return 0;
}
```

Output:

```
Element 7 found at index 2
```

Distributed Implementation (Using MPI in C++)

```
#include <mpi.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
    int A[] = {4, 2, 7, 9, 5, 8, 3};
    int n = sizeof(A) / sizeof(A[0]);
    int target;
```

```
    MPI_Init(&argc, &argv);
    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Only root process asks for input
    if (world_rank == 0) {
        cout << "Enter the number to search: ";
        cin >> target;
    }

    // Broadcast the target to all processes
    MPI_Bcast(&target, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Determine range for this process
    int local_start = world_rank * n / world_size;
    int local_end = (world_rank + 1) * n / world_size;
    int local_result = -1;

    cout << "Process " << world_rank << " searching indices ["
        << local_start << ", " << local_end - 1 << "]" << endl;

    for (int i = local_start; i < local_end; ++i) {
        if (A[i] == target) {
            local_result = i;
            break;
        }
    }

    if (local_result != -1) {
        cout << "Process " << world_rank << " found the number at index " <<
local_result << endl;
    }

    // Gather all local results at root
    int global_results[world_size];
    MPI_Gather(&local_result, 1, MPI_INT, global_results, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    // Root process prints final result
    if (world_rank == 0) {
        int found_index = -1;
        for (int i = 0; i < world_size; ++i) {
            if (global_results[i] != -1) {
                found_index = global_results[i];
                break;
            }
        }

        if (found_index != -1)
            cout << "Final Result: Number found at index: " << found_index << endl;
        else
            cout << "Final Result: Number not found in array." << endl;
    }
    MPI_Finalize();
```

```
    return 0;
}
```

Output:

```
guest@guest:~$ mpic++ -o mpi_search mpi_search.cpp
guest@guest:~$ mpirun -np 4 ./mpi_search
Enter the number to search: 7
Process 0 searching indices [0, 0]
Process 1 searching indices [1, 2]
Process 1 found the number at index 2
Process 2 searching indices [3, 4]
Process 3 searching indices [5, 6]
Final Result: Number found at index: 2
```

Conclusion

Linear Search is simple yet powerful for specific use cases. Its parallel and distributed implementations significantly improve performance for large datasets, leveraging multiple cores and processors.