UNIVERSITY OF KARACHI
DEPARTMENT OF COMPUTER SCIENCE
UBIT

PROJECT FILE

PROJECT TITLE:
BINARY SEARCH ALGORITHM

SUBMITTED BY:
MEHAK FATIMA (B21110006057)
RIMSHA LARAIB (B21110006107)

SUBMITTED TO:
SIR MUHAMMAD SAEED

INTRODUCTION:

Binary Search Algorithm is a searching algorithm used in a sorted array by **r**epeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

MATHEMATICAL MODEL:

Let:

A be a **sorted array** of n elements: A[0],A[1],...,A[n−1]A[0], A[1], ..., A[n-1]

x be the **target** element to find.

Algorithm:

- Divide the search space into two halves by **finding the middle index "mid"**.
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
  - If the **key** is smaller than the middle element, then the **left** side is used for next search.
  - If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

APPLICATIONS:

Binary Search is used in:

- **Databases** – Searching sorted indexe
- **Computer Networks** – Routing tables, IP lookup
- **Operating Systems** – Process scheduling, memory paging
- **Machine Learning** – Parameter tuning (grid search)
- **Game Development** – AI decision trees
- **Compiler Design** – Symbol table lookup

IMPLEMENTATIONS

1. SEQUENTIAL IMPLEMENTATION (ITERATIVE - C++)

Code:

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

// Sequential Binary Search
int binarySearch(int arr[], int n, int toSearch)
{
    int low = 0, high = n - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2; // mid
calculation
        if (arr[mid] == toSearch)   // check if it is
found
        {
            return mid;
        }
        else if (arr[mid] < toSearch) // if the value
is less than change the low
        {
            low = mid + 1;
        }
        else // otherwise change the high
```

```
        {
            high = mid - 1;
        }
    }
    return -1;
}
int main(int argc, char **argv)
{
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int n = sizeof(arr) / sizeof(arr[0]); //
calculate the size of array
    int toSearch;                          // value to
search
    cout << "Sequential Binary Search\n";
    cout << "Enter the value you want to search: ";
    cin >> toSearch;
    int result = binarySearch(arr, n, toSearch); //
store the result
    if (result != -1)                              //
check if value is found or not
    {
        cout << "Found " << toSearch << " at index: "
<< result << endl;
    }
    else
   {
      cout << "Value not found!" << endl;
   }
   return 0;
}
```

Output:

```
Sequential Binary Search
Enter the value you want to search: 15
Found 15 at index: 7
```

```
Sequential Binary Search
Enter the value you want to search: 12
Value not found!
```

2. PARALLEL IMPLEMENTATION (OPENMP - C++)

Code:

```
#include <iostream>
#include <omp.h> // For OpenMP parallel
implementation
using namespace std;

void parallelBinarySearch(int arr[], int n, int
queries[], int qCount, int results[]) {
    #pragma omp parallel for
    for (int i = 0; i < qCount; i++) {
        int low = 0, high = n - 1;
        int key = queries[i];
        results[i] = -1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == key) {
                results[i] = mid;
                break;
            } else if (arr[mid] < key) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }
}
int main() {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int n = sizeof(arr) / sizeof(arr[0]);
    int queries[] = {3, 15, 19, 4};
    int qCount = sizeof(queries) / sizeof(queries[0]);
    int results[qCount];
    parallelBinarySearch(arr, n, queries, qCount,
results);
    for (int i = 0; i < qCount; i++) {
        if(results[i]!=-1){
        cout << "Query " << queries[i] << " found at
index: " << results[i] << std::endl;
        }
        else{
```

```
            cout<<"Query " << queries[i] <<" not
found!"<<endl;
        }
    }
    return 0;
}
```

Output:

```
Query 3 found at index: 1
Query 15 found at index: 7
Query 19 found at index: 9
Query 4 not found!
```

3. DISTRIBUTED IMPLEMENTATION (MPI)

Code:

```
binary search:
#include <iostream>
#include <mpi.h>
using namespace std;

void mpiBinarySearch(int argc, char** argv, int arr[],
int n, int key) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int chunkSize = n / size;
    int start = rank * chunkSize;
    int end = (rank == size - 1) ? (n - 1) : (start +
chunkSize - 1);
    // Show which part this process is responsible
for
    cout << "Process " << rank << " searching indices
[" << start << ", " << end << "]" << endl;
    int foundIndex = -1;
    for (int i = start; i <= end; i++) {
        if (arr[i] == key) {
            foundIndex = i;
            break;
        }
```

```
    }
    if (foundIndex != -1) {
        cout << "Process " << rank << " found the
number at index " << foundIndex << endl;
    }
    // Use MPI_Reduce to get the max index (i.e.,
valid index if found)
    int globalIndex = -1;
    MPI_Reduce(&foundIndex, &globalIndex, 1, MPI_INT,
MPI_MAX, 0, MPI_COMM_WORLD)
    if (rank == 0) {
        cout << "\nFinal Result: ";
        if (globalIndex != -1)
            cout << "Key " << key << " found at index:
" << globalIndex << endl;
        else
            cout << "Key " << key << " not found in
array." << endl;
    }
    MPI_Finalize();
}
int main(int argc, char** argv) {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 13;  // You can change this or take
from user input
    mpiBinarySearch(argc, argv, arr, n, key);
    return 0;
}
```

Output:

```
guest@guest:~$ mpic++ -o binary_mpi binary_mpi.cpp
guest@guest:~$ mpirun -np 4 ./binary_mpi
Process 1 searching indices [2, 3]
Process 2 searching indices [4, 5]
Process 3 searching indices [6, 9]
Process 3 found the number at index 6
Process 0 searching indices [0, 1]

Final Result: Key 13 found at index: 6
```