Practical 1A

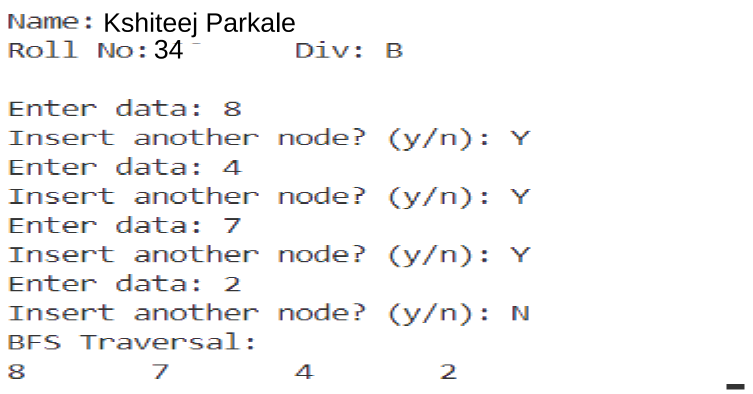
```
#include <iostream>

#include <queue>

#include <omp.h> // Required for OpenMP directives

using namespace std;

class Node

{

public:

Node *left, *right;

int data;

};

class BreadthFS

{

public:

Node *insert(Node *root, int data); // Insert a node into the tree

void bfs(Node *root);

 };

              //

 Perform parallel BFS

// Insert a new node using level-order insertion

Node *BreadthFS::insert(Node *root, int data)

{

if (!root)

{

}

root = new Node;

root->left = NULL;

root->right = NULL;

root->data = data;

return root;

    std::queue<Node *> q;
```

```
q.push(root);

while (!q.empty())
{
   Node *current = q.front();
   q.pop();

   if (!current->left)
   {
      current->left = new Node;
      current->left->left = NULL;
      current->left->right = NULL;
      current->left->data = data;
      return root;
   }
   else
   {
      q.push(current->left);
   }

   if (!current->right)
   {
      current->right = new Node;
      current->right->left = NULL;
      current->right->right = NULL;
      current->right->data = data;
      return root;
   }
   else
   {
      q.push(current->right);
```

```
}

}

return root; // Ensures all control paths return a value

}

// Parallel BFS using OpenMP

void BreadthFS::bfs(Node *root)

{

if (!root)

return;

queue<Node *> q;

q.push(root);

while (!q.empty())

{

int level_size = q.size();

#pragma omp parallel for // Parallelize processing of nodes at the current level

for (int i = 0; i < level_size; i++)

{

Node *current = NULL;

#pragma omp critical // Thread-safe access to the queue

{

}

current = q.front();

q.pop();

cout << current->data << "\t";

#pragma omp critical // Thread-safe insertion of children

{

if (current->left)

q.push(current->left);

if (current->right)

q.push(current->right);

}
```

```
} } }
int main()
{
}
BreadthFS bfs;
Node *root = NULL;
int data;
char choice;
cout << "\n\nName:Krishna S.Kabra\nRoll No:23 \t Div:B\n\n";
do
{
cout << "Enter data: ";
cin >> data;
root = bfs.insert(root, data);
cout << "Insert another node? (y/n): ";
cin >> choice;
} while (choice == 'y' || choice == 'Y');
cout << "BFS Traversal:\n";
bfs.bfs(root);
return 0;
```

Output



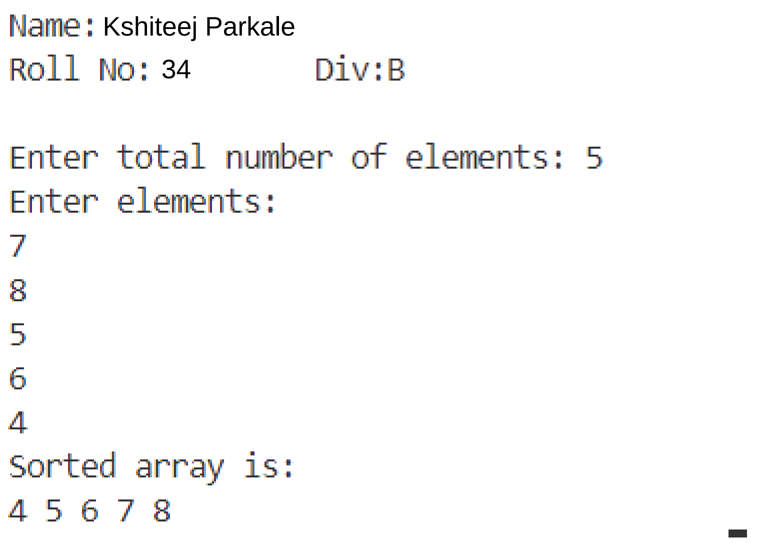
```
Name: Kshiteej Parkale
Roll No: 34       Div: B

Enter data: 8
Insert another node? (y/n): Y
Enter data: 4
Insert another node? (y/n): Y
Enter data: 7
Insert another node? (y/n): Y
Enter data: 2
Insert another node? (y/n): N
BFS Traversal:
8       7       4       2
```

Practical 2A

```
#include <iostream>
#include <cstdlib>
#include <omp.h>
using namespace std;
void bubble(int *, int);
void swap(int &, int &);
void bubble(int *a, int n)
{
for (int i = 0; i < n; i++)
{
int first = i % 2;
#pragma omp parallel for shared(a, first)
for (int j = first; j < n - 1; j += 2)
{
}
}
}
if (a[j] > a[j + 1])
{
}
swap(a[j], a[j + 1]);
void swap(int &a, int &b)
{
}
int temp;
temp = a;
a = b;
b = temp;
int main()
{
```

```
cout << "\n\nName: Krishna S.Kabra\nRoll No:23\t Div:B\n\n";
int *a, n;
cout << "\nEnter total number of elements: ";
cin >> n;
a = new int[n];
cout << "\nEnter elements: ";
for (int i = 0; i < n; i++)
{
}
cin >> a[i];
bubble(a, n);
cout << "\nSorted array is:\n";
for (int i = 0; i < n; i++) {
cout << a[i] << " ";}
cout << endl;
delete[] a;
return 0;
}
```

Output



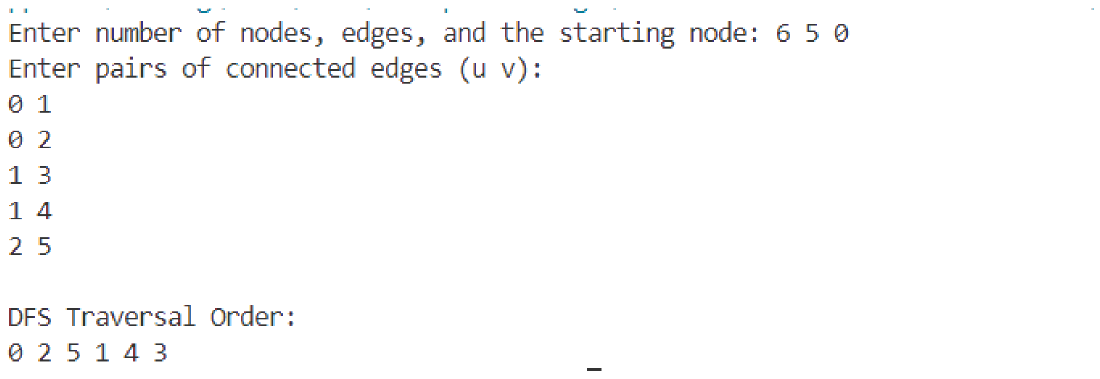
```
Name: Kshiteej Parkale
Roll No: 34        Div:B

Enter total number of elements: 5
Enter elements:
7
8
5
6
4
Sorted array is:
4 5 6 7 8
```

Practical 1B

```
#include <iostream>

#include <vector>

#include <stack>

#include <omp.h>

using namespace std

const int MAX = 100000;

vector<int> graph[MAX];

bool visited[MAX];

omp_lock_t lock[MAX];

void dfs(int start_node)

{

stack<int> s;

s.push(start_node)

while (!s.empty())

{

int curr_node = s.top();

s.pop();

// Lock for current node

omp_set_lock(&lock[curr_node]);

if (!visited[curr_node])

{

}

visited[curr_node] = true;

cout << curr_node << " ";

omp_unset_lock(&lock[curr_node]);

// Push adjacent nodes (no parallelization inside stack push)

for (int i = 0; i < graph[curr_node].size(); i++)

{

int adj_node = graph[curr_node][i];

omp_set_lock(&lock[adj_node]);
if (!visited[adj_node])
            {
                s.push(adj_node);
            }
            omp_unset_lock(&lock[adj_node]);
        }
    }
}
int main()
{
    int n, m, start_node;
    cout << "Enter number of nodes, edges, and the starting node: ";
    cin >> n >> m >> start_node;
    cout << "Enter pairs of connected edges (u v):\n";
    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u); // Assuming undirected graph
    }
    // Initialize visited and locks
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
        omp_init_lock(&lock[i]);
    }
    cout << "\nDFS Traversal Order:\n";
    dfs(start_node);
    cout << endl;
    // Destroy locks
```

```
 for (int i = 0; i < n; i++)
 {
omp_destroy_lock(&lock[i]);
}
return 0;
}
```

Output



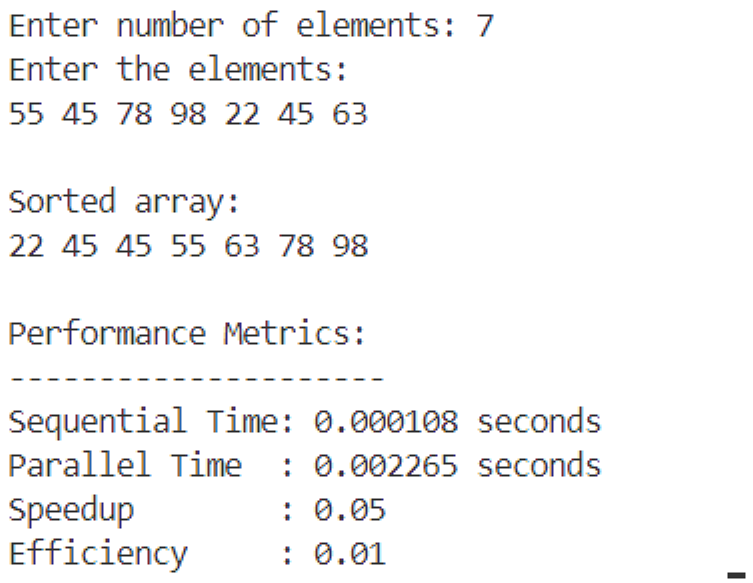
```
Enter number of nodes, edges, and the starting node: 6 5 0
Enter pairs of connected edges (u v):
0 1
0 2
1 3
1 4
2 5

DFS Traversal Order:
0 2 5 1 4 3
```

Practical 2B

```
#include <iostream>
#include <omp.h>
#include <vector>
using namespace std;
void merge(vector<int> &arr, int l, int m, int r)
{
}
int n1 = m - l + 1;
int n2 = r - m;
vector<int> L(n1), R(n2);
for (int i = 0; i < n1; i++)
L[i] = arr[l + i];
for (int j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
int i = 0, j = 0, k = l;
while (i < n1 && j < n2)
arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
while (i < n1)
arr[k++] = L[i++];
while (j < n2)
arr[k++] = R[j++];
void mergeSortSequential(vector<int> &arr, int l, int r)
{
if (l < r)
{
int m = l + (r - l) / 2;
mergeSortSequential(arr, l, m);
mergeSortSequential(arr, m + 1, r);
merge(arr, l, m, r);
}
```

```
}
void mergeSortParallel(vector<int> &arr, int l, int r, int depth = 0)
{
if (l < r)
{
int m = l + (r - l) / 2;
if (depth < 4)
{
#pragma omp parallel sections
{
#pragma omp section
mergeSortParallel(arr, l, m, depth + 1);
#pragma omp section
mergeSortParallel(arr, m + 1, r, depth + 1);
}  }
else
{
}
mergeSortSequential(arr, l, m);
mergeSortSequential(arr, m + 1, r);
merge(arr, l, m, r);
} }
int main()
{
int n;
cout << "Enter number of elements: ";
cin >> n;
vector<int> arr(n), arrSeq(n);
cout << "Enter the elements:\n";
for (int i = 0; i < n; i++) {
cin >> arr[i];
}

arrSeq = arr; // Copy input for sequential sort

double start = omp_get_wtime();

mergeSortSequential(arrSeq, 0, n - 1);

double end = omp_get_wtime();

double seqTime = end - start;

start = omp_get_wtime();

mergeSortParallel(arr, 0, n - 1);

end = omp_get_wtime();

double parTime = end - start;

cout << "\nSorted array:\n";

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

cout << "\n";

double speedup = seqTime / parTime;

int numThreads = omp_get_max_threads();

double efficiency = speedup / numThreads;

cout << "\nPerformance Metrics:";

cout << "\n---------------------";

cout << "\nSequential Time: " << seqTime << " seconds";

cout << "\nParallel Time  : " << parTime << " seconds";

cout << "\nSpeedup        : " << speedup;

cout << "\nEfficiency     : " << efficiency << endl;

return 0; }
```

Output



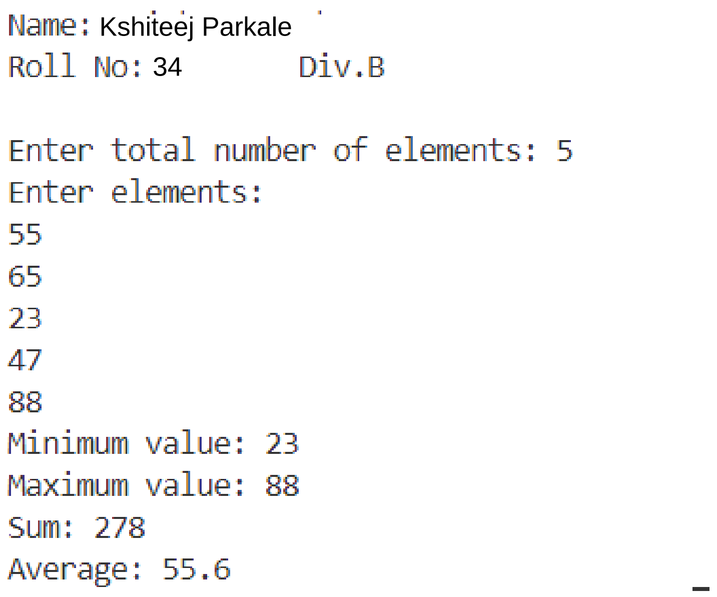
```
Enter number of elements: 7
Enter the elements:
55 45 78 98 22 45 63

Sorted array:
22 45 45 55 63 78 98

Performance Metrics:
---------------------
Sequential Time: 0.000108 seconds
Parallel Time  : 0.002265 seconds
Speedup        : 0.05
Efficiency     : 0.01
```

Practical 3

```
#include <iostream>
#include <omp.h>
#include <climits>
using namespace std;
void min_reduction(int arr[], int n)
{
int min_value = INT_MAX;
#pragma omp parallel for reduction(min : min_value)
for (int i = 0; i < n; i++)
{
}
if (arr[i] < min_value)
{
}
min_value = arr[i];
cout << "Minimum value: " << min_value << endl;
}
void max_reduction(int arr[], int n)
{
int max_value = INT_MIN;
#pragma omp parallel for reduction(max : max_value)
for (int i = 0; i < n; i++)
{
}
if (arr[i] > max_value)
{
}
max_value = arr[i];
cout << "Maximum value: " << max_value << endl;
}
```

```
void sum_reduction(int arr[], int n)

{

int sum = 0;

#pragma omp parallel for reduction(+ : sum)

for (int i = 0; i < n; i++)

{

}

sum += arr[i];

cout << "Sum: " << sum << endl;

}

void average_reduction(int arr[], int n)

{

if (n <= 1)

{

}

cout << "Average: Cannot calculate (array size too small)" << endl;

return;

int sum = 0;

#pragma omp parallel for reduction(+ : sum)

for (int i = 0; i < n; i++)

{

}

sum += arr[i];

cout << "Average: " << static_cast<double>(sum) / n << endl;

}

int main()

{

cout << "\n\nName: Krishna S.Kabra\nRoll No: 23 \t Div.B\n\n";

int *arr, n;

cout << "\nEnter total number of elements: ";

cin >> n;
if (n <= 0)

{

cerr << "Error: Array size must be positive" << endl;

return 1;

}

arr = new int[n];

cout << "\nEnter elements:\n";

for (int i = 0; i < n; i++)

{

}

cin >> arr[i];

min_reduction(arr, n);

max_reduction(arr, n);

sum_reduction(arr, n);

average_reduction(arr, n);

delete[] arr;

return 0;

}
```

Output

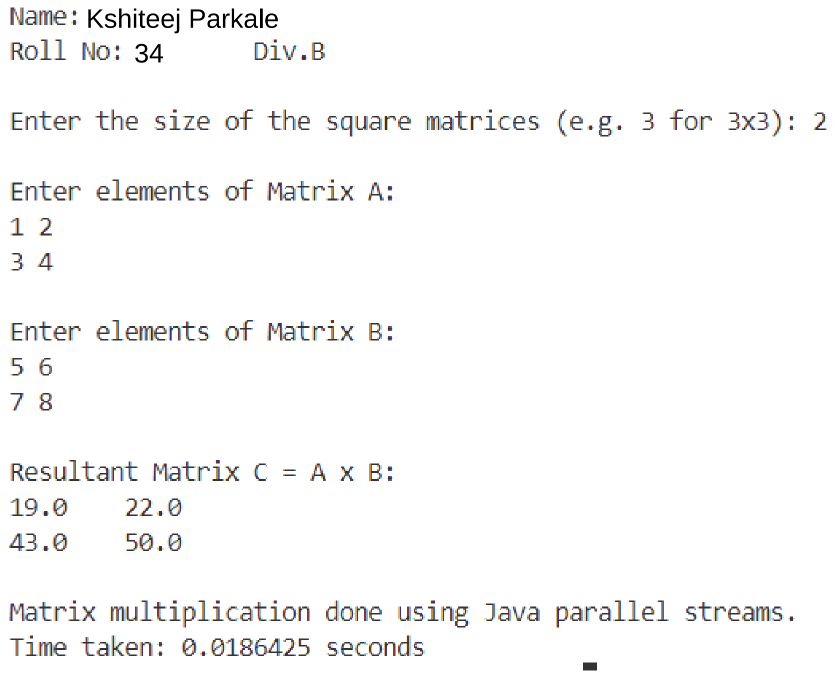


```
Name: Kshiteej Parkale
Roll No: 34        Div.B

Enter total number of elements: 5
Enter elements:
55
65
23
47
88
Minimum value: 23
Maximum value: 88
Sum: 278
Average: 55.6
```

Practical 4

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
int n;
cout << "\nName: Krishna S.Kabra\nRoll No: 23 \t Div.B\n";
cout << "\nEnter the size of the square matrices (e.g. 3 for 3x3): ";
cin >> n;
float A[n][n], B[n][n], C[n][n];
cout << "\nEnter elements of Matrix A:\n";
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
cin >> A[i][j];
cout << "\nEnter elements of Matrix B:\n";
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
cin >> B[i][j];
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = 0;
double start = omp_get_wtime();
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
for (int k = 0; k < n; k++)
C[i][j] += A[i][k] * B[k][j];
double end = omp_get_wtime();
cout << "\nResultant Matrix C = A x B:\n";
for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
      cout << C[i][j] << "\t";
    cout << endl;
  }
  cout << "\n Matrix multiplication done using OpenMP.";
  cout << "\n Time taken: " << end - start << " seconds\n";
  return 0;
}
```

Output



```
Name: Kshiteej Parkale
Roll No: 34       Div.B

Enter the size of the square matrices (e.g. 3 for 3x3): 2

Enter elements of Matrix A:
1 2
3 4

Enter elements of Matrix B:
5 6
7 8

Resultant Matrix C = A x B:
19.0    22.0
43.0    50.0

Matrix multiplication done using Java parallel streams.
Time taken: 0.0186425 seconds
```