Ques:- Implementing First Come First Serve with arrival time

```
#include <bits/stdc++.h>
using namespace std;
main()
{
   int n;
   cout << "Enter the number of processes:- ";
   cin >> n;
   vector<vector<int>> sjfs(n, vector<int>(3));
   for (int i = 0; i < n; i++)
   {
      cout << "Enter arrival time and burst time for process " << i + 1 << ":- ";
      sjfs[i][0] = i + 1;
      cin >> sjfs[i][1] >> sjfs[i][2];
   }
   sort(
      sjfs.begin(), sjfs.end(), [](vector<int> &a, vector<int> &b)
      {
         if (a[1] == b[1])
         {
            return a[0] < b[0];
         }
         return a[1] < b[1]; });
   priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
   int i = 1;
   pq.push({sjfs[0][1], sjfs[0][0], sjfs[0][2]});
   int ct = 0;
   double tat = 0, wt = 0;
   vector<pair<int, pair<int, int>>> ans;
   while (!pq.empty())
```

```
    {
        int bt = pq.top()[2];
        int at = pq.top()[0];
        int process = pq.top()[1];
        ct += bt;
        tat += (ct - at);
        wt += ((ct - at) - bt);
        ans.push_back({process, {at, ct}});
        pq.pop();
        while (true)
        {
            if (i < n && sjfs[i][1] <= ct)
            {
                pq.push({sjfs[i][1], sjfs[i][0], sjfs[i][2]});
                i++;
            }
            else
                break;
        }
    }
    for (auto it : ans)
    {
        cout << it.first << " " << it.second.first << " " << it.second.second << endl;
    }
    cout << "Average Turn Around Time:- " << tat / n << endl;
    cout << "Average Waiting Time:- " << wt / n << endl;
}
```

OUTPUT

```
Enter the number of processes:- 4
Enter arrival time and burst time for process 1:- 1 2
Enter arrival time and burst time for process 2:- 2 3
Enter arrival time and burst time for process 3:- 3 4
Enter arrival time and burst time for process 4:- 4 5
1 1 2
2 2 5
3 3 9
4 4 14
Average Turn Around Time:- 5
Average Waiting Time:- 1.5
```

Ques:- Implementing SJFS scheduling algo without arrival time.

```
#include <bits/stdc++.h>
using namespace std;
main()
{
   int n;
   cout << "Enter the number of processes:- ";
   cin >> n;
   vector<vector<int>> sjfs(n, vector<int>(3));
   for (int i = 0; i < n; i++)
   {
      cout << "Enter burst time for the Process " << i + 1 << ":- ";
      sjfs[i][0] = i + 1;
      sjfs[i][1] = 0;
      cin >> sjfs[i][2];
   }
   sort(sjfs.begin(), sjfs.end(), [](vector<int> &a, vector<int> &b)
        {
      if(a[2] == b[2])
         return a[0] < b[0];
      return a[2]<b[2]; });
   int ct = 0;
   double tat = 0, wt = 0;
   for (auto it : sjfs)
   {
      int bt = it[2];
      ct += bt;
      tat += ct;
      wt += (ct - bt);
   }
   cout << "Average Turn Around Time:- " << tat / n << endl;
   cout << "Average Waiting Time:- " << wt / n << endl;
}
```

OUTPUT

```
Enter the number of processes:- 4
Enter burst time for the Process 1:- 2
Enter burst time for the Process 2:- 4
Enter burst time for the Process 3:- 5
Enter burst time for the Process 4:- 7
Average Turn Around Time:- 9.25
Average Waiting Time:- 4.75
PS C:\Users\harsh\OneDrive - Graphic Era University\Desktop\Programing\OS\output>
```

Ques:- Implementing SJFS scheduling algo with arrival time.

```
#include <bits/stdc++.h>
using namespace std;
main()
{
   int n;
   cout << "Enter the number of processes:- ";
   cin >> n;
   vector<vector<int>> sjfs(n, vector<int>(3));
   for (int i = 0; i < n; i++)
   {
      cout << "Enter arrival time and burst time for process " << i + 1 << ":- ";
      sjfs[i][0] = i + 1;
      cin >> sjfs[i][1] >> sjfs[i][2];
   }
   sort(
      sjfs.begin(), sjfs.end(), [](vector<int> &a, vector<int> &b)
      {
         if (a[1] == b[1])
         {
            if (a[2] == b[2])
               return a[0] < b[0];
            else
               return a[2] < b[2];
         }

            return a[1] < b[1]; });
   priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;

   vector<pair<int, pair<int, pair<int, int>>>> ans;
```

```
    float tat = 0, wt = 0;
    pq.push({sjfs[0][2], sjfs[0][1], sjfs[0][0]});
    int i = 1;
    int ct = 0;
    while (!pq.empty())
    {
        int bt = pq.top()[0];
        int at = pq.top()[1];
        int process = pq.top()[2];
        pq.pop();
        ct += bt;
        tat += ct;
        wt += ((ct - at) - bt);
        ans.push_back({process, {at, {bt, ct}}});

        while (true)
        {
            if (i < n && sjfs[i][1] <= ct)
            {
                pq.push({sjfs[i][2], sjfs[i][1], sjfs[i][0]});
                i++;
            }
            else
                break;
        }
    }
    sort(ans.begin(), ans.end(), [](const pair<int, pair<int, pair<int, int>>> &a, const pair<int,
pair<int, pair<int, int>>> &b)
         { return a.first < b.first; });

    tat /= n;
```

```
    wt /= n;
    for (auto it : ans)
        cout << it.first << " " << it.second.first << " " << it.second.second.first << " " <<
it.second.second.second << endl;
    cout << "Average Turn Around Time:- " << tat << endl;
    cout << "Average Waiting Time:- " << wt << endl;
}
```

OUTPUT

```
Enter the number of processes:- 4
Enter arrival time and burst time for process 1:- 1 2
Enter arrival time and burst time for process 2:- 2 3
Enter arrival time and burst time for process 3:- 3 4
Enter arrival time and burst time for process 4:- 4 5
1 1 2 2
2 2 3 5
3 3 4 9
4 4 5 14
Average Turn Around Time:- 7.5
Average Waiting Time:- 1.5
PS C:\Users\harsh\OneDrive - Graphic Era University\Desktop\Programing\OS\output>
```

Ques:- Implementing Shortest Job First scheduling algorithm.

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    vector<int> arrivalTime(n);
    vector<int> burstTime(n);
    vector<int> remainingTime(n);
    vector<int> completionTime(n);
    vector<int> turnaroundTime(n);
    vector<int> waitingTime(n);

    for (int i = 0; i < n; i++)
    {
        cout << "Enter arrival time for process " << i + 1 << ": ";
        cin >> arrivalTime[i];
        cout << "Enter burst time for process " << i + 1 << ": ";
        cin >> burstTime[i];
        remainingTime[i] = burstTime[i];
    }

    int currentTime = 0;
    int completedProcesses = 0;
```

```
    while (completedProcesses < n)
    {
        int shortestJob = -1;
        int shortestTime = INT_MAX;

        for (int i = 0; i < n; i++)
        {
            if (arrivalTime[i] <= currentTime && remainingTime[i] < shortestTime &&
remainingTime[i] > 0)
            {
                shortestJob = i;
                shortestTime = remainingTime[i];
            }
        }

        if (shortestJob == -1)
        {
            currentTime++;
            continue;
        }

        remainingTime[shortestJob]--;
        currentTime++;

        if (remainingTime[shortestJob] == 0)
        {
            completionTime[shortestJob] = currentTime;
            turnaroundTime[shortestJob] = completionTime[shortestJob] -
arrivalTime[shortestJob];
            waitingTime[shortestJob] = turnaroundTime[shortestJob] - burstTime[shortestJob];
            completedProcesses++;
```

```
        }
    }

    double avgTurnaroundTime = 0;
    double avgWaitingTime = 0;

    for (int i = 0; i < n; i++)
    {
        avgTurnaroundTime += turnaroundTime[i];
        avgWaitingTime += waitingTime[i];
    }

    avgTurnaroundTime /= n;
    avgWaitingTime /= n;

    cout << "Average Turnaround Time: " << avgTurnaroundTime << endl;
    cout << "Average Waiting Time: " << avgWaitingTime << endl;

    return 0;
}
```

OUTPUT

```
Enter the number of processes: 4
Enter arrival time for process 1: 1
Enter burst time for process 1: 2
Enter arrival time for process 2: 2
Enter burst time for process 2: 3
Enter arrival time for process 3: 3
Enter burst time for process 3: 4
Enter arrival time for process 4: 4
Enter burst time for process 4: 5
Average Turnaround Time: 6
Average Waiting Time: 2.5
```

Ques:- Implementing Round Robin Scheduling algo.

```
#include <iostream>

using namespace std;

void queueUpdation(int queue[], int timer, int arrival[], int n, int maxProccessIndex)
{
    int zeroIndex;
    for (int i = 0; i < n; i++)
    {
        if (queue[i] == 0)
        {
            zeroIndex = i;
            break;
        }
    }
    queue[zeroIndex] = maxProccessIndex + 1;
}

void queueMaintainence(int queue[], int n)
{
    for (int i = 0; (i < n - 1) && (queue[i + 1] != 0); i++)
    {
        int temp = queue[i];
        queue[i] = queue[i + 1];
        queue[i + 1] = temp;
    }
}

void checkNewArrival(int timer, int arrival[], int n, int maxProccessIndex, int queue[])
```

```
{
    if (timer <= arrival[n - 1])
    {
        bool newArrival = false;
        for (int j = (maxProccessIndex + 1); j < n; j++)
        {
            if (arrival[j] <= timer)
            {
                if (maxProccessIndex < j)
                {
                    maxProccessIndex = j;
                    newArrival = true;
                }
            }
        }
        // adds the incoming process to the ready queue
        //(if any arrives)
        if (newArrival)
            queueUpdation(queue, timer, arrival, n, maxProccessIndex);
    }
}

// Driver Code
int main()
{
    int n, tq, timer = 0, maxProccessIndex = 0;
    float avgWait = 0, avgTT = 0;
    cout << "\nEnter the time quanta : ";
    cin >> tq;
    cout << "\nEnter the number of processes : ";
```

```
cin >> n;
int arrival[n], burst[n], wait[n], turn[n], queue[n], temp_burst[n];
bool complete[n];

cout << "\nEnter the arrival time of the processes : ";
for (int i = 0; i < n; i++)
   cin >> arrival[i];

cout << "\nEnter the burst time of the processes : ";
for (int i = 0; i < n; i++)
{
   cin >> burst[i];
   temp_burst[i] = burst[i];
}

for (int i = 0; i < n; i++)
{ // Initializing the queue and complete array
   complete[i] = false;
   queue[i] = 0;
}
while (timer < arrival[0]) // Incrementing Timer until the first process arrives
   timer++;
queue[0] = 1;

while (true)
{
   bool flag = true;
   for (int i = 0; i < n; i++)
   {
      if (temp_burst[i] != 0)
```

```
    {
        flag = false;
        break;
    }
}
if (flag)
    break;

for (int i = 0; (i < n) && (queue[i] != 0); i++)
{
    int ctr = 0;
    while ((ctr < tq) && (temp_burst[queue[0] - 1] > 0))
    {
        temp_burst[queue[0] - 1] -= 1;
        timer += 1;
        ctr++;

        // Checking and Updating the ready queue until all the processes arrive
        checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
    }
    // If a process is completed then store its exit time
    // and mark it as completed
    if ((temp_burst[queue[0] - 1] == 0) && (complete[queue[0] - 1] == false))
    {
        // turn array currently stores the completion time
        turn[queue[0] - 1] = timer;
        complete[queue[0] - 1] = true;
    }

    // checks whether or not CPU is idle
```

```
        bool idle = true;
        if (queue[n - 1] == 0)
        {
            for (int i = 0; i < n && queue[i] != 0; i++)
            {
                if (complete[queue[i] - 1] == false)
                {
                    idle = false;
                }
            }
        }
        else
            idle = false;

        if (idle)
        {
            timer++;
            checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
        }

        // Maintaining the entries of processes
        // after each premption in the ready Queue
        queueMaintainence(queue, n);
    }
}

for (int i = 0; i < n; i++)
{
    turn[i] = turn[i] - arrival[i];
    wait[i] = turn[i] - burst[i];
```

```
    }

    cout << "\nProgram No.\tArrival Time\tBurst Time\tWait Time\tTurnAround Time"
         << endl;
    for (int i = 0; i < n; i++)
    {
        cout << i + 1 << "\t\t" << arrival[i] << "\t\t"
             << burst[i] << "\t\t" << wait[i] << "\t\t" << turn[i] << endl;
    }
    for (int i = 0; i < n; i++)
    {
        avgWait += wait[i];
        avgTT += turn[i];
    }
    cout << "\nAverage wait time : " << (avgWait / n)
         << "\nAverage Turn Around Time : " << (avgTT / n);

    return 0;
}
```

OUTPUT

```
Enter the time quanta : 2

Enter the number of processes : 4

Enter the arrival time of the processes : 1
2
3
4

Enter the burst time of the processes : 2
3
4
5

Program No.     Arrival Time    Burst Time      Wait Time       TurnAround Time
1               1               2               0               2
2               2               3               5               8
3               3               4               5               9
4               4               5               6               11

Average wait time : 4
Average Turn Around Time : 7.5
```

Ques:- Implementing Priority Scheduling algo.

```
#include <iostream>
using namespace std;
int main()
{
    int bt[20], p[20], wt[20], tat[20], pr[20], i, j, n, total = 0, pos, temp, avg_wt, avg_tat;
    cout << "Enter Total Number of Process:";
    cin >> n;
    cout << "\nEnter Burst Time and Priority\n";
    for (i = 0; i < n; i++)
    {
        cout << "\nP[" << i + 1 << "]\n";
        cout << "Burst Time:";
        cin >> bt[i];
        cout << "Priority:";
        cin >> pr[i];
        p[i] = i + 1; // contains process number
    }
    for (i = 0; i < n; i++)
    {
        pos = i;
        for (j = i + 1; j < n; j++)
        {
            if (pr[j] < pr[pos])
                pos = j;
        }
        temp = pr[i];
        pr[i] = pr[pos];
        pr[pos] = temp;
        temp = bt[i];
```

```
            bt[i] = bt[pos];
            bt[pos] = temp;
            temp = p[i];
            p[i] = p[pos];
            p[pos] = temp;
        }
        wt[0] = 0; // waiting time for first process is zero
        for (i = 1; i < n; i++)
        {
            wt[i] = 0;
            for (j = 0; j < i; j++)
                wt[i] += bt[j];
            total += wt[i];
        }
        avg_wt = total / n; // average waiting time
        total = 0;
        cout << "\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time";
        for (i = 0; i < n; i++)
        {
            tat[i] = bt[i] + wt[i]; // calculate turnaround time
            total += tat[i];
            cout << "\nP[" << p[i] << "]\t\t  " << bt[i] << "\t\t    " << wt[i] << "\t\t\t" << tat[i];
        }
        avg_tat = total / n; // average turnaround time
        cout << "\n\nAverage Waiting Time=" << avg_wt;
        cout << "\nAverage Turnaround Time=" << avg_tat;
        return 0;
    }
```

OUTPUT

```
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:1
Priority:6

P[2]
Burst Time:2
Priority:9

P[3]
Burst Time:3
Priority:2

P[4]
Burst Time:4
Priority:8

Process     Burst Time          Waiting Time    Turnaround Time
P[3]              3                 0                 3
P[1]              1                 3                 4
P[4]              4                 4                 8
P[2]              2                 8                 10

Average Waiting Time=3
Average Turnaround Time=6
```

Ques:- Implementing Optimal Page replacement algorithm.

```
#include <bits/stdc++.h>
using namespace std;

bool search(int key, vector<int> &fr)
{
   for (int i = 0; i < fr.size(); i++)
      if (fr[i] == key)
         return true;
   return false;
}

int predict(int pg[], vector<int> &fr, int pn, int index)
{
   int res = -1, farthest = index;
   for (int i = 0; i < fr.size(); i++)
   {
      int j;
      for (j = index; j < pn; j++)
      {
         if (fr[i] == pg[j])
         {
            if (j > farthest)
            {
               farthest = j;
               res = i;
            }
            break;
         }
      }
```

```
        if (j == pn)
            return i;
    }
    return (res == -1) ? 0 : res;
}

void optimalPage(int pg[], int pn, int fn)
{
    vector<int> fr;
    int hit = 0;
    for (int i = 0; i < pn; i++)
    {
        if (search(pg[i], fr))
        {
            hit++;
            continue;
        }

        if (fr.size() < fn)
            fr.push_back(pg[i]);

        else
        {
            int j = predict(pg, fr, pn, i + 1);
            fr[j] = pg[i];
        }
    }
    cout << "No. of hits = " << hit << endl;
    cout << "No. of misses = " << pn - hit << endl;
}
```

```
int main()
{
    int pn, fn;
    cout << "Enter no of pages and no of frames:";
    cin >> pn >> fn;
    int pg[pn];
    cout << "Enter the sequence of page nos.:";
    for (int i = 0; i < pn; ++i)
        cin >> pg[i];
    optimalPage(pg, pn, fn);
    return 0;
}
```

OUTPUT

```
Enter no of pages and no of frames:10 16
Enter the sequence of page nos.:4 5 6 5 6 1 3 2 4 3
No. of hits = 4
No. of misses = 6
```

Ques:- Implementing LRU page replacement algorithm.

```
#include <bits/stdc++.h>
using namespace std;

int pageFaults(int pages[], int n, int capacity)
{
    unordered_set<int> s;
    unordered_map<int, int> indexes;
    int page_faults = 0;
    for (int i = 0; i < n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i]) == s.end())
            {
                s.insert(pages[i]);
                page_faults++;
            }
            indexes[pages[i]] = i;
        }
        else
        {
            if (s.find(pages[i]) == s.end())
            {
                int lru = INT_MAX, val;
                for (auto it = s.begin(); it != s.end(); it++)
                {
                    if (indexes[*it] < lru)
                    {
                        lru = indexes[*it];
```

```
                val = *it;
            }
        }
        s.erase(val);
        s.insert(pages[i]);
        page_faults++;
      }
      indexes[pages[i]] = i;
    }
  }

  return page_faults;
}
int main()
{

  int pn, fn;
  cout << "Enter no of pages and no of frames:";
  cin >> pn >> fn;
  int pg[pn];
  cout << "Enter the sequence of page nos.:";
  for (int i = 0; i < pn; ++i)
    cin >> pg[i];
  cout << pageFaults(pg, pn, fn);
}
```

OUTPUT

```
Enter no of pages and no of frames:10 16
Enter the sequence of page nos.:4 5 6 5 6 1 3 2 4 3
6
```

Ques:- Implementing FCFS disk scheduling algorithm

```
#include <bits/stdc++.h>
using namespace std;

void FCFS(vector<int> arr, int head)
{
    int seek_count = 0;
    int distance, cur_track;

    for (int i = 0; i < arr.size(); i++)
    {
        cur_track = arr[i];
        distance = abs(cur_track - head);
        seek_count += distance;
        head = cur_track;
    }

    cout << "Seek time = "
         << seek_count << endl;
}

int main()
{
    int n;
    cout << "Enter no of tracks to request:";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the sequence of tracks nos. requested:";
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
```

```
    cout << "Enter the initial position of head:";
    int head;
    cin >> head;
    FCFS(arr, head);
}
```

OUTPUT

```
Enter no of tracks to request:10
Enter the sequence of tracks nos. requested:4 5 6 5 6 1 3 2 4 3
Enter the initial position of head:0
Seek time = 19
```

Ques:- Implementing SCAN disk scheduling algorithm

```
#include <bits/stdc++.h>
using namespace std;

int size = 8;
int disk_size = 200;

void SCAN(int arr[], int head, string direction, int size, int disk_size)
{
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;
    if (direction == "left")
        left.push_back(0);
    else if (direction == "right")
        right.push_back(disk_size - 1);

    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
    std::sort(right.begin(), right.end());
    int run = 2;
    while (run--)
    {
```

```
    if (direction == "left")
    {
        for (int i = left.size() - 1; i >= 0; i--)
        {
            cur_track = left[i];
            seek_sequence.push_back(cur_track);
            distance = abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        direction = "right";
    }
    else if (direction == "right")
    {
        for (int i = 0; i < right.size(); i++)
        {
            cur_track = right[i];
            seek_sequence.push_back(cur_track);
            distance = abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        direction = "left";
    }
  }

  cout << "Seek time = "
       << seek_count << endl;
}
```

```
int main()
{

    int n, head, disk_size;
    cout << "Enter no of tracks to request:";
    cin >> n;
    int arr[n];
    cout << "Enter the sequence of tracks nos. requested:";
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
    cout << "Enter the initial position of head:";
    cin >> head;
    cout << "Enter initial direction of head movement(left or right):";
    string direction;
    cin >> direction;
    cout << "Enter the disk size:";
    cin >> disk_size;
    SCAN(arr, head, direction, n, disk_size);
}
```

OUTPUT

```
Enter no of tracks to request:10
Enter the sequence of tracks nos. requested:4 5 6 5 6 1 3 2 4 3
Enter the initial position of head:4
Enter initial direction of head movement(left or right):left
Enter the disk size:100
Seek time = 10
```

Ques:- Implementing CSCAN disk scheduling algorithm

```
#include <bits/stdc++.h>
using namespace std;

void CSCAN(int arr[], int head, int size, int disk_size)
{
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;
    left.push_back(0);
    right.push_back(disk_size - 1);
    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
    std::sort(right.begin(), right.end());
    for (int i = 0; i < right.size(); i++)
    {
        cur_track = right[i];
        seek_sequence.push_back(cur_track);
        distance = abs(cur_track - head);
        seek_count += distance;
        head = cur_track;
    }
    head = 0;
```

```
    seek_count += (disk_size - 1);
    for (int i = 0; i < left.size(); i++)
    {
        cur_track = left[i];
        seek_sequence.push_back(cur_track);
        distance = abs(cur_track - head);
        seek_count += distance;
        head = cur_track;
    }

    cout << "Seek time = "
         << seek_count << endl;
}
int main()
{
    int n, head, disk_size;
    cout << "Enter no of tracks to request:";
    cin >> n;
    int arr[n];
    cout << "Enter the sequence of tracks nos. requested:";
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
    cout << "Enter the initial position of head:";
    cin >> head;
    cout << "Enter the disk size:";
    cin >> disk_size;
    CSCAN(arr, head, n, disk_size);

    return 0;
}
```

OUTPUT

```
Enter no of tracks to request:10
Enter the sequence of tracks nos. requested:4 5 6 5 6 1 3 2 4 3
Enter the initial position of head:2
Enter the disk size:100
Seek time = 197
```

Ques:- Implementing LOOK disk scheduling algorithm

```
#include <bits/stdc++.h>
using namespace std;

void LOOK(int arr[], int head, string direction, int size, int disk_size)
{
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
    std::sort(right.begin(), right.end());
    int run = 2;
    while (run--)
    {
        if (direction == "left")
        {
            for (int i = left.size() - 1; i >= 0; i--)
            {
                cur_track = left[i];
                seek_sequence.push_back(cur_track);
                distance = abs(cur_track - head);
                seek_count += distance;
```

```
            head = cur_track;
        }
        direction = "right";
    }
    else if (direction == "right")
    {
        for (int i = 0; i < right.size(); i++)
        {
            cur_track = right[i];
            seek_sequence.push_back(cur_track);
            distance = abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        direction = "left";
    }
  }

  cout << "Seek time = "
       << seek_count << endl;
}

int main()
{
  int n, head, disk_size;
  cout << "Enter no of tracks to request:";
  cin >> n;
  int arr[n];
  cout << "Enter the sequence of tracks nos. requested:";
  for (int i = 0; i < n; ++i)
```

```
        cin >> arr[i];
    cout << "Enter the initial position of head:";
    cin >> head;
    cout << "Enter initial direction of head movement(left or right):";
    string direction;
    cin >> direction;
    cout << "Enter the disk size:";
    cin >> disk_size;
    LOOK(arr, head, direction, n, disk_size);
}
```

OUTPUT

```
Enter no of tracks to request:10
Enter the sequence of tracks nos. requested:4 5 6 5 6 1 3 2 4 3
Enter the initial position of head:3
Enter initial direction of head movement(left or right):left
Enter the disk size:100
Seek time = 7
```

Ques:- Implementing CLOOK disk scheduling algorithm

```
#include <bits/stdc++.h>
using namespace std;

void CLOOK(int arr[], int head, int size, int disk_size)
{
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
    std::sort(right.begin(), right.end());
    for (int i = 0; i < right.size(); i++)
    {
        cur_track = right[i];
        seek_sequence.push_back(cur_track);
        distance = abs(cur_track - head);
        seek_count += distance;
        head = cur_track;
    }
    seek_count += abs(head - left[0]);
    head = left[0];
    for (int i = 0; i < left.size(); i++)
```

```
    {
        cur_track = left[i];
        seek_sequence.push_back(cur_track);
        distance = abs(cur_track - head);
        seek_count += distance;
        head = cur_track;
    }
    cout << "Seek time = "
         << seek_count << endl;
}
int main()
{
    int n, head, disk_size;
    cout << "Enter no of tracks to request:";
    cin >> n;
    int arr[n];
    cout << "Enter the sequence of tracks nos. requested:";
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
    cout << "Enter the initial position of head:";
    cin >> head;
    cout << "Enter the disk size:";
    cin >> disk_size;
    CLOOK(arr, head, n, disk_size);

    return 0;
}
```

OUTPUT

```
Enter no of tracks to request:10
Enter the sequence of tracks nos. requested:4 5 6 5 6 1 3 2 4 3
Enter the initial position of head:4
Enter the disk size:100
Seek time = 9
```