Round Robin Scheduling Algorithm

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

// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
            int bt[], int wt[], int quantum)
{
    // Make a copy of burst times bt[] to store remaining
    // burst times.
    int rem_bt[n];
    for (int i = 0 ; i < n ; i++)
        rem_bt[i] = bt[i];

    int t = 0; // Current time

    // Keep traversing processes in round robin manner
    // until all of them are not done.
    while (1)
    {
        bool done = true;

        // Traverse all processes one by one repeatedly
        for (int i = 0 ; i < n; i++)
        {
            // If burst time of a process is greater than 0
            // then only need to process further
            if (rem_bt[i] > 0)
            {
```

```
    done = false; // There is a pending process

    if (rem_bt[i] > quantum)
    {
        // Increase the value of t i.e. shows
        // how much time a process has been processed
        t += quantum;

        // Decrease the burst_time of current process
        // by quantum
        rem_bt[i] -= quantum;
    }

    // If burst time is smaller than or equal to
    // quantum. Last cycle for this process
    else
    {
        // Increase the value of t i.e. shows
        // how much time a process has been processed
        t = t + rem_bt[i];

        // Waiting time is current time minus time
        // used by this process
        wt[i] = t - bt[i];

        // As the process gets fully executed
        // make its remaining burst time = 0
        rem_bt[i] = 0;
    }
}
```

```
        }

        // If all processes are done
        if (done == true)
        break;
    }
}

// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n,
                        int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

// Function to calculate average time
void findavgTime(int processes[], int n, int bt[],
                                     int quantum)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt, quantum);

    // Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);
    // Display processes along with all details
    cout << "PN\t "<< " \tBT "
        << "  WT " << " \tTAT\n";
     // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
            << wt[i] <<"\t\t " << tat[i] <<endl;
    }
     cout << "Average waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
}
 // Driver code
int main()
{
    // process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
     // Burst time of all processes
    int burst_time[] = {10, 5, 8};
     // Time quantum
    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```

Priority Scheduling Algorithm

```
#include<stdio.h>
int main()
{
   int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
   printf("Enter Total Number of Process:");
   scanf("%d",&n);
   printf("\nEnter Burst Time and Priority\n");
   for(i=0;i<n;i++)
   {
      printf("\nP[%d]\n",i+1);
      printf("Burst Time:");
      scanf("%d",&bt[i]);
      printf("Priority:");
      scanf("%d",&pr[i]);
      p[i]=i+1;           //contains process number
   }
   //sorting burst time, priority and process number in ascending order using selection sort
   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
    //calculate waiting time
    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;
    printf("\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];
        printf("\nP[%d]\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat=total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);
return 0;
}
```

FIFO page replacement policy

```
#include < stdio.h >
int main()
{
   int incomingStream[] = {4 , 1 , 2 , 4 , 5};
   int pageFaults = 0;
   int frames = 3;
   int m, n, s, pages;
   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
   printf(" Incoming \ t Frame 1 \ t Frame 2 \ t Frame 3 ");
   int temp[ frames ];
   for(m = 0; m < frames; m++)
   {
      temp[m] = -1;
   }
   for(m = 0; m < pages; m++)
   {
      s = 0;
      for(n = 0; n < frames; n++)
      {
         if(incomingStream[m] == temp[n])
         {
            s++;
            pageFaults--;
         }
      }
      pageFaults++;
      if((pageFaults <= frames) && (s == 0))
      {
         temp[m] = incomingStream[m];
```

```
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = incomingStream[m];
        }
        printf("\n");
        printf("%d\t\t\t",incomingStream[m]);
        for(n = 0; n < frames; n++)
        {
            if(temp[n] != -1)
                printf(" %d\t\t\t", temp[n]);
            else
                printf(" - \t\t\t");
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

MRU page replacement algorithm

```
#include <bits/stdc++.h>
using namespace std;
 // Function to update the array
// in most recently used fashion
void mostRecentlyUsedProcesses(int* arr, int N, int K)
{
    int app_index = 0;
     // Finding the end index after K presses
    app_index = (K % N);
     // Shifting elements by 1 towards the found index
    // on which the K press ends
    int x = app_index, app_id = arr[app_index];
    while (x > 0) {
        arr[x] = arr[--x];
    }
     // Update the current active process
    arr[0] = app_id;
}
 // Utility function to print
// the contents of the array
void printArray(int* arr, int N)
{
    for (int i = 0; i < N; i++)
        cout << arr[i] << " ";
}
 // Driver code
int main()
{
    int K = 3;
    int arr[] = { 3, 5, 2, 4, 1 };
    int N = sizeof(arr) / sizeof(arr[0]);
    mostRecentlyUsedProcess(arr, N, K);
    printArray(arr, N);
    return 0;
}
```

LRU page replacement algorithm

```
#include<stdio.h>

int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;

for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}

int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
   for(i = 0; i < no_of_pages; ++i){
    scanf("%d", &pages[i]);
   }

for(i = 0; i < no_of_frames; ++i){
    frames[i] = -1;
   }
```

```
  for(i = 0; i < no_of_pages; ++i){
   flag1 = flag2 = 0;

   for(j = 0; j < no_of_frames; ++j){
   if(frames[j] == pages[i]){
   counter++;
   time[j] = counter;
  flag1 = flag2 = 1;
  break;
  }
   }

   if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
   if(frames[j] == -1){
   counter++;
   faults++;
   frames[j] = pages[i];
   time[j] = counter;
   flag2 = 1;
   break;
   }
   }
   }

   if(flag2 == 0){
   pos = findLRU(time, no_of_frames);
   counter++;
   faults++;
```

```
     frames[pos] = pages[i];
     time[pos] = counter;
     }

     printf("\n");

     for(j = 0; j < no_of_frames; ++j){
     printf("%d\t", frames[j]);
     }
}
printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

Optimal page replacement algorithm

```
#include<stdio.h>
int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                   flag1 = flag2 = 1;
                   break;
               }
        }
```

```
if(flag1 == 0){
    for(j = 0; j < no_of_frames; ++j){
        if(frames[j] == -1){
            faults++;
            frames[j] = pages[i];
            flag2 = 1;
            break;
        }
    }
}

if(flag2 == 0){
 flag3 =0;

    for(j = 0; j < no_of_frames; ++j){
     temp[j] = -1;

     for(k = i + 1; k < no_of_pages; ++k){
     if(frames[j] == pages[k]){
     temp[j] = k;
     break;
     }
     }
    }

    for(j = 0; j < no_of_frames; ++j){
     if(temp[j] == -1){
     pos = j;
     flag3 = 1;
```

```
             break;
             }
            }

            if(flag3 ==0){
             max = temp[0];
             pos = 0;

             for(j = 1; j < no_of_frames; ++j){
             if(temp[j] > max){
             max = temp[j];
             pos = j;
             }
             }
            }
frames[pos] = pages[i];
faults++;
        }

        printf("\n");

        for(j = 0; j < no_of_frames; ++j){
            printf("%d\t", frames[j]);
        }
    }

    printf("\n\nTotal Page Faults = %d", faults);
    return 0;
}
```

FCFS disk scheduling

```
#include <stdio.h>
#include <math.h>

int size = 8;

void FCFS(int arr[],int head)
{
   int seek_count = 0;
    int cur_track, distance;

   for(int i=0;i<size;i++)
     {
     cur_track = arr[i];

       // calculate absolute distance
     distance = fabs(head - cur_track);

       // increase the total count
     seek_count += distance;

       // accessed track is now new head
     head = cur_track;
   }

   printf("Total number of seek operations: %d\n",seek_count);

    // Seek sequence would be the same
   // as request array sequence
   printf("Seek Sequence is\n");
    for (int i = 0; i < size; i++) {
     printf("%d\n",arr[i]);
   }
}

//Driver code
int main()
{
    // request array
   int arr[8] = { 176, 79, 34, 60, 92, 11, 41, 114 };
    int head = 50;

   FCFS(arr,head);

   return 0;
}
```

SSTF disk scheduling

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

// Calculates difference of each
// track number with the head position
void calculatedifference(int request[], int head,
                         int diff[][2], int n)
{
    for(int i = 0; i < n; i++)
    {
        diff[i][0] = abs(head - request[i]);
    }
}

// Find unaccessed track which is
// at minimum distance from head
int findMIN(int diff[][2], int n)
{
    int index = -1;
    int minimum = 1e9;

    for(int i = 0; i < n; i++)
    {
        if (!diff[i][1] && minimum > diff[i][0])
        {
            minimum = diff[i][0];
            index = i;
        }
    }
    return index;
}

void shortestSeekTimeFirst(int request[],
                           int head, int n)
{
    if (n == 0)
    {
        return;
    }

    // Create array of objects of class node
    int diff[n][2] = { { 0, 0 } };

    // Count total number of seek operation
    int seekcount = 0;

    // Stores sequence in which disk access is done
    int seeksequence[n + 1] = {0};

    for(int i = 0; i < n; i++)
    {
        seeksequence[i] = head;
        calculatedifference(request, head, diff, n);
        int index = findMIN(diff, n);
        diff[index][1] = 1;

        // Increase the total count
        seekcount += diff[index][0];
        // Accessed track is now new head
        head = request[index];
    }
    seeksequence[n] = head;

    cout << "Total number of seek operations = "
         << seekcount << endl;
    cout << "Seek sequence is : " << "\n";

    // Print the sequence
    for(int i = 0; i <= n; i++)
    {
        cout << seeksequence[i] << "\n";
    }
}

// Driver code
int main()
{
    int n = 8;
    int proc[n] = { 176, 79, 34, 60, 92, 11, 41, 114 };

    shortestSeekTimeFirst(proc, 50, n);

    return 0;
}
```