LABORATORY RECORD

OPERATING SYSTEM

Student Name- BEDANTA MAHANTY

Student Regn. No.- 23BAI11022

Interim Semester 2024-25

Slot: D11+D12+A14

Year- 2024-25

Faculty Name- Dr. Nilesh Kunhare

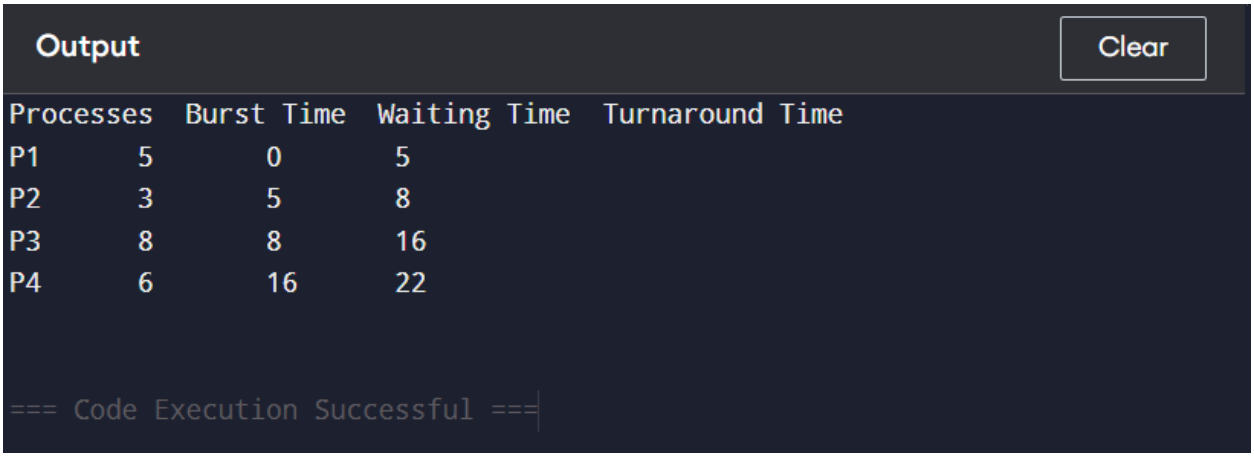
School of Computing Science & Engineering

VIT Bhopal University

Table of Contents:

1.First Come First Serve (FCFS) Scheduling

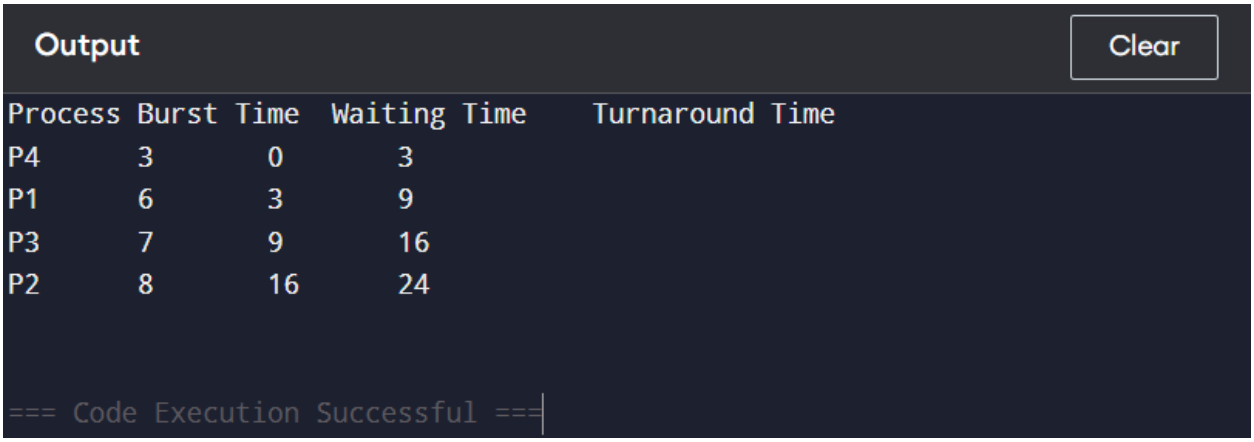
```
#include <iostream>
using namespace std;

void FCFS(int processes[], int n, int bt[], int wt[], int tat[]) {
   wt[0] = 0;
   for (int i = 1; i < n; i++)
      wt[i] = wt[i - 1] + bt[i - 1];
   for (int i = 0; i < n; i++)
      tat[i] = wt[i] + bt[i];
}

int main() {
   int processes[] = {1, 2, 3, 4};
   int n = 4;
   int burst_time[] = {5, 3, 8, 6};
   int waiting_time[n], turnaround_time[n];

   FCFS(processes, n, burst_time, waiting_time, turnaround_time);

   cout << "Processes  Burst Time  Waiting Time  Turnaround Time\n";
   for (int i = 0; i < n; i++) {
      cout << "P" << processes[i] << "\t\t" << burst_time[i] << "\t\t"
         << waiting_time[i] << "\t\t" << turnaround_time[i] << endl;
   }
   return 0;
}
```



2.SJF (Shortest Job First)

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

struct Process {
   int id, bt, wt, tat;
};

bool compare(Process a, Process b) {
   return a.bt < b.bt;
}

int main() {
   Process p[] = {{1, 6}, {2, 8}, {3, 7}, {4, 3}};
   int n = 4;

   sort(p, p + n, compare);
   p[0].wt = 0;
```

```
    for (int i = 1; i < n; i++)
        p[i].wt = p[i - 1].wt + p[i - 1].bt;

    for (int i = 0; i < n; i++)
        p[i].tat = p[i].wt + p[i].bt;

    cout << "Process\tBurst Time\tWaiting Time\tTurnaround Time\n";
    for (int i = 0; i < n; i++)
        cout << "P" << p[i].id << "\t\t" << p[i].bt << "\t\t" << p[i].wt << "\t\t" << p[i].tat << endl;

    return 0;
}
```



3. Priority Scheduling

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

struct Process {
    int id, bt, pri, wt, tat;
```

```
};

bool compare(Process a, Process b) {
    return a.pri > b.pri;  // Higher priority first
}

int main() {
    Process p[] = {{1, 10, 1}, {2, 1, 3}, {3, 2, 2}, {4, 1, 4}};
    int n = 4;

    sort(p, p + n, compare);

    p[0].wt = 0;
    for (int i = 1; i < n; i++)
        p[i].wt = p[i - 1].wt + p[i - 1].bt;

    for (int i = 0; i < n; i++)
        p[i].tat = p[i].wt + p[i].bt;

    cout << "Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n";
    for (int i = 0; i < n; i++)
        cout << "P" << p[i].id << "\t\t" << p[i].bt << "\t\t" << p[i].wt << "\t\t" << p[i].tat << endl;

    return 0;
}
```

Output

Clear

```
Process Burst Time  Priority    Waiting Time    Turnaround Time
P4      1       0       1
P2      1       1       2
P3      2       2       4
P1      10      4       14
```

4. Round Robin

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

struct Process {
   int id, bt, remaining_bt;
};

void roundRobin(Process processes[], int n, int quantum) {
   queue<int> q;
   int waiting_time[n] = {0}, turnaround_time[n];
   int time = 0;

   for (int i = 0; i < n; i++)
      q.push(i);

   while (!q.empty()) {
      int i = q.front();
      q.pop();

      if (processes[i].remaining_bt <= quantum) {
         time += processes[i].remaining_bt;
```

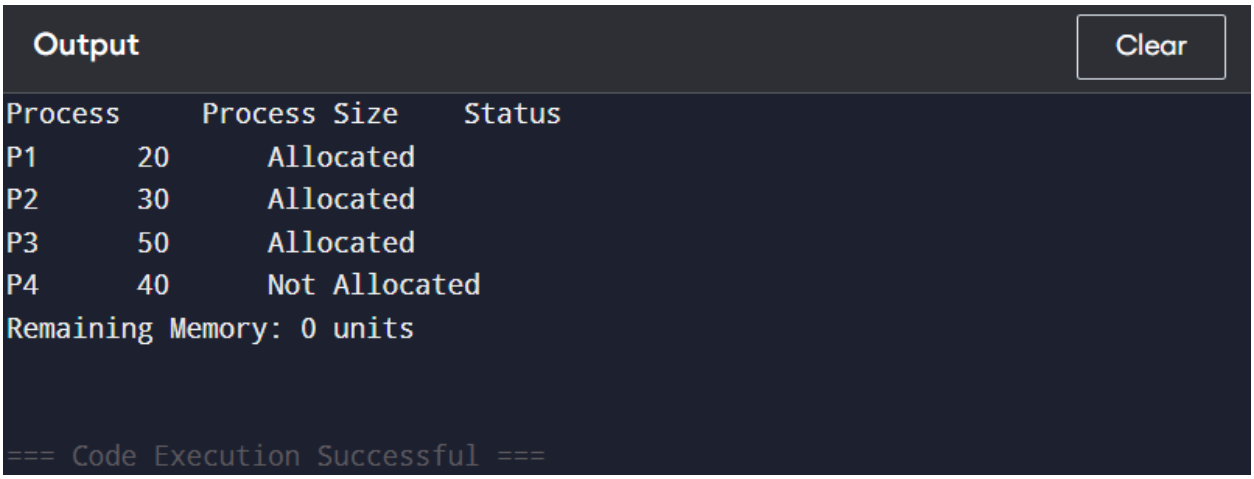
```
            turnaround_time[i] = time;
            processes[i].remaining_bt = 0;
        } else {
            time += quantum;
            processes[i].remaining_bt -= quantum;
            q.push(i);
        }
    }

    cout << "Process\tBurst Time\tTurnaround Time\n";
    for (int i = 0; i < n; i++)
        cout << "P" << processes[i].id << "\t\t" << processes[i].bt << "\t\t"
             << turnaround_time[i] << endl;
}

int main() {
    Process processes[] = {{1, 5, 5}, {2, 10, 10}, {3, 15, 15}};
    int n = 3, quantum = 5;

    roundRobin(processes, n, quantum);
    return 0;
}
```

Output Clear

```
Process Burst Time  Turnaround Time
P1      5       5
P2      10      20
P3      15      30
```

2.a) MVT (Multiple Variable Tasks)

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

void allocateMVT(int memorySize, int processSize[], int n) {
   int remainingMemory = memorySize;

   cout << "Process\t\tProcess Size\tStatus\n";
   for (int i = 0; i < n; i++) {
      if (processSize[i] <= remainingMemory) {
         remainingMemory -= processSize[i];
         cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tAllocated\n";
      } else {
         cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tNot Allocated\n";
      }
   }
   cout << "Remaining Memory: " << remainingMemory << " units\n";
}

int main() {
   int memorySize = 100;  // Total available memory
   int processSize[] = {20, 30, 50, 40};  // Sizes of processes
   int n = sizeof(processSize) / sizeof(processSize[0]);

   allocateMVT(memorySize, processSize, n);

   return 0;
}
```



2.b) MFT (Multiple Fixed Tasks)

```
#include <iostream>

using namespace std;


void allocateMFT(int blockSize, int memorySize, int processSize[], int n) {
   int numBlocks = memorySize / blockSize;
   int remainingBlocks = numBlocks;


   cout << "Process\t\tProcess Size\tStatus\t\tBlock Wasted\n";
   for (int i = 0; i < n; i++) {
      if (processSize[i] <= blockSize && remainingBlocks > 0) {
         remainingBlocks--;
         cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tAllocated\t" << blockSize - processSize[i]
<< endl;
      } else {
         cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tNot Allocated\t--\n";
      }
   }
   cout << "Remaining Blocks: " << remainingBlocks << endl;
}
```

```
int main() {
    int memorySize = 100;
    int blockSize = 25;
    int processSize[] = {20, 30, 15, 25};
    int n = sizeof(processSize) / sizeof(processSize[0]);

    allocateMFT(blockSize, memorySize, processSize, n);

    return 0;
}
```

Output Clear

```
Process     Process Size    Status      Block Wasted
P1      20      Allocated   5
P2      30      Not Allocated   --
P3      15      Allocated   10
P4      25      Allocated   0
Remaining Blocks: 1


=== Code Execution Successful ===
```

3.Simulate Bankers algorithm for Deadlock Avoidance

```
#include <iostream>
#include <vector>

using namespace std;
class BankersAlgorithm {
private:
    int numProcesses, numResources;
    vector<vector<int>> allocation, max, need;
    vector<int> available;
```

```
public:
    BankersAlgorithm(int p, int r) : numProcesses(p), numResources(r) {
        allocation.resize(p, vector<int>(r));
        max.resize(p, vector<int>(r));
        need.resize(p, vector<int>(r));
        available.resize(r);
    }
    void inputData() {
        allocation = {{0, 1, 0},
                      {2, 0, 0},
                      {3, 0, 2},
                      {2, 1, 1},
                      {0, 0, 2}};
        max = {{7, 5, 3},
               {3, 2, 2},
               {9, 0, 2},
               {2, 2, 2},
               {4, 3, 3}};
        available = {3, 3, 2};

        for (int i = 0; i < numProcesses; i++) {
            for (int j = 0; j < numResources; j++) {
                need[i][j] = max[i][j] - allocation[i][j];
            }
        }
    }
    bool isSafe() {
        vector<int> work = available;
        vector<bool> finish(numProcesses, false);
```

```
vector<int> safeSeq;


while (safeSeq.size() < numProcesses) {
   bool progressMade = false;
   for (int i = 0; i < numProcesses; i++) {
      if (!finish[i]) {
         bool canAllocate = true;
         for (int j = 0; j < numResources; j++) {
            if (need[i][j] > work[j]) {
               canAllocate = false;
               break;
            }
         }
         if (canAllocate) {
            for (int j = 0; j < numResources; j++) {
               work[j] += allocation[i][j];
            }
            finish[i] = true;
            safeSeq.push_back(i);
            progressMade = true;
         }
      }
   }
   if (!progressMade) {
      cout << "System is in an unsafe state.\n";
      return false;
   }
}
cout << "System is in a safe state. Safe Sequence: ";
```

```
        for (int i = 0; i < safeSeq.size(); i++) {
            cout << "P" << safeSeq[i] << " ";
        }
        cout << endl;

        return true;
    }
};
int main() {
    int numProcesses = 5, numResources = 3;

    BankersAlgorithm bankers(numProcesses, numResources);
    bankers.inputData();

    if (!bankers.isSafe()) {
        cout << "Deadlock detected. The system is unsafe.\n";
    }
    return 0;
}
```

Output Clear

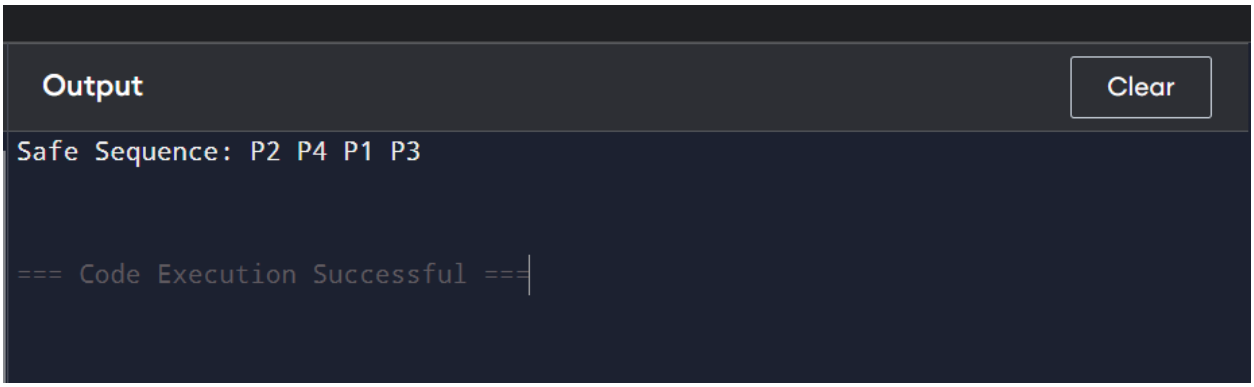
```
System is in a safe state. Safe Sequence: P1 P3 P4 P0 P2


=== Code Execution Successful ===
```

4. Simulate Bankers Algorithm for deadlock Prevention

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

string bankers_algorithm(vector<string>& processes, vector<int>& available, vector<vector<int>>&
max_claims, vector<vector<int>>& allocations) {
    int num_processes = processes.size();
    int num_resources = available.size();
    vector<vector<int>> need(num_processes, vector<int>(num_resources));
    vector<int> work = available;
    vector<bool> finish(num_processes, false);
    vector<string> safe_sequence;

    for (int i = 0; i < num_processes; ++i)
        for (int j = 0; j < num_resources; ++j)
            need[i][j] = max_claims[i][j] - allocations[i][j];

    while (safe_sequence.size() < num_processes) {
        bool allocated = false;
        for (int i = 0; i < num_processes; ++i) {
            if (!finish[i]) {
                bool can_allocate = true;
                for (int j = 0; j < num_resources; ++j) {
                    if (need[i][j] > work[j]) {
                        can_allocate = false;
                        break;
```

```
                }
            }
            if (can_allocate) {
                for (int j = 0; j < num_resources; ++j)
                    work[j] += allocations[i][j];
                safe_sequence.push_back(processes[i]);
                finish[i] = true;
                allocated = true;
            }
        }
    }
    if (!allocated)
        return "Deadlock detected. No safe sequence.";
  }

  string result;
  for (const string& p : safe_sequence)
    result += p + " ";
  return result;
}

int main() {
  vector<string> processes = {"P1", "P2", "P3", "P4"};
  vector<int> available = {3, 3, 2};
  vector<vector<int>> max_claims = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}};
  vector<vector<int>> allocations = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}};

  string result = bankers_algorithm(processes, available, max_claims, allocations);
  cout << "Safe Sequence: " << result << endl;
```

```
    return 0;
}
```


Output
Clear
Safe Sequence: P2 P4 P1 P3
=== Code Execution Successful ===

5. Simulate all Page Replacement Algorithms

a) FIFO

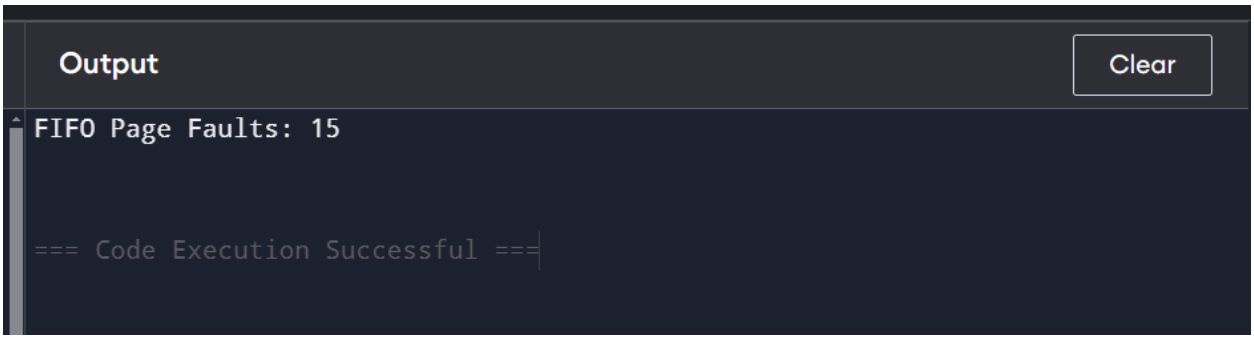
b) LRU

c) Optimal

A. FIFO

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

int fifo(vector<int>& pages, int capacity) {
   vector<int> memory;
   int faults = 0;
   for (int page : pages) {
      if (find(memory.begin(), memory.end(), page) == memory.end()) {
         if (memory.size() < capacity)
            memory.push_back(page);
         else {
            memory.erase(memory.begin());
            memory.push_back(page);
         }
         faults++;
      }
   }
   return faults;
}
```

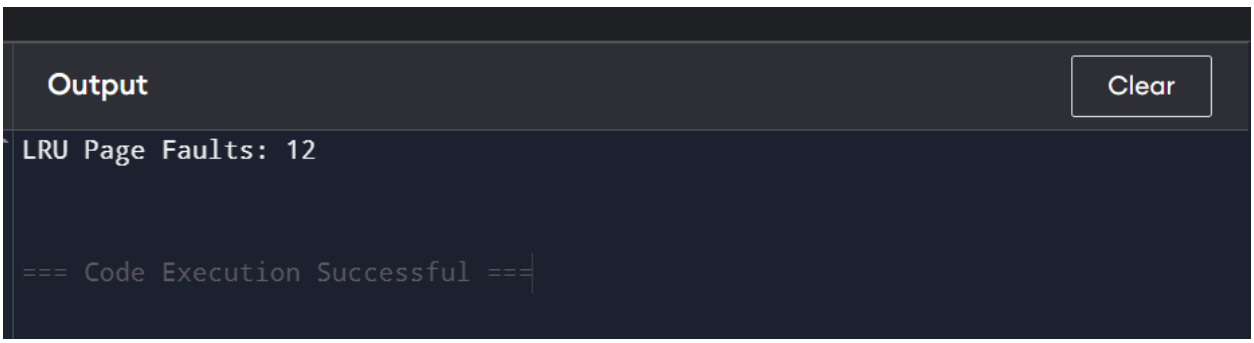
```
int main() {
    vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
    int capacity = 3;

    cout << "FIFO Page Faults: " << fifo(pages, capacity) << endl;
    return 0;
}
```



B. LRU

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

int lru(vector<int>& pages, int capacity) {
    vector<int> memory;
    unordered_map<int, int> recent;
    int faults = 0;
    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        if (find(memory.begin(), memory.end(), page) == memory.end()) {
            if (memory.size() < capacity)
```

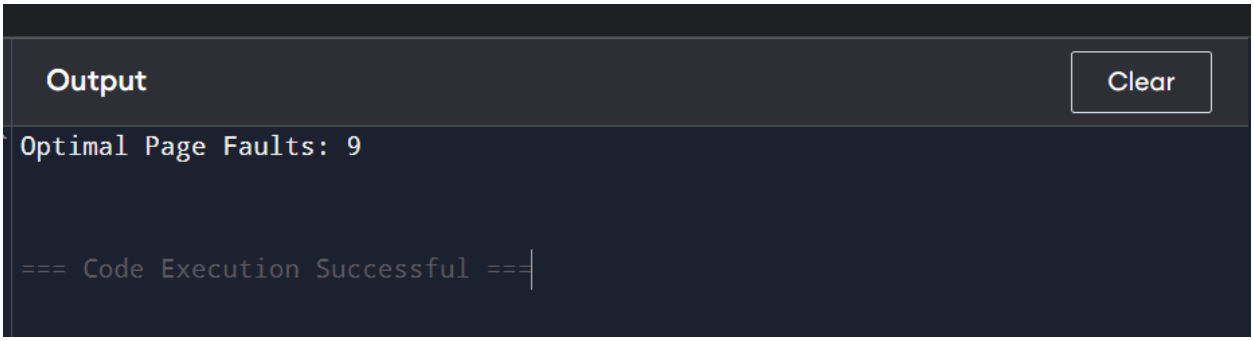
```
            memory.push_back(page);
        else {
            int lru_page = memory[0];
            for (int p : memory) {
                if (recent[p] < recent[lru_page])
                    lru_page = p;
            }
            memory.erase(find(memory.begin(), memory.end(), lru_page));
            memory.push_back(page);
        }
        faults++;
    }
    recent[page] = i;
  }
  return faults;
}

int main() {
  vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
  int capacity = 3;
  cout << "LRU Page Faults: " << lru(pages, capacity) << endl;
  return 0;
}
```



C. OPTIMAL

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

int optimal(vector<int>& pages, int capacity) {
    vector<int> memory;
    int faults = 0;
    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        if (find(memory.begin(), memory.end(), page) == memory.end()) {
            if (memory.size() < capacity)
                memory.push_back(page);
            else {
                int farthest = -1, to_remove = -1;
                for (int j = 0; j < memory.size(); j++) {
                    int next_use = find(pages.begin() + i + 1, pages.end(), memory[j]) - pages.begin();
                    if (next_use > farthest) {
                        farthest = next_use;
                        to_remove = j;
                    }
                }
                memory[to_remove] = page;
            }
            faults++;
        }
    }
```

```
    return faults;
}

int main() {
    vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
    int capacity = 3;

    cout << "Optimal Page Faults: " << optimal(pages, capacity) << endl;
    return 0;
}
```



6. Simulate Paging Technique of Memory Management

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

void pagingTechnique(int memorySize, int pageSize, unordered_map<int, vector<int>>& processes) {
    int numFrames = memorySize / pageSize;
    vector<int> memory(numFrames, -1); // Initialize all frames to -1 (empty)
    unordered_map<int, unordered_map<int, int>> pageTable;    cout << "Total Memory Frames: " <<
numFrames << endl;
```

```
for (auto& process : processes) {
    int pid = process.first;
    vector<int>& pages = process.second;

    cout << "Allocating pages for Process " << pid << ":\n";
    for (int page : pages) {
        auto emptyFrame = find(memory.begin(), memory.end(), -1);
        if (emptyFrame != memory.end()) {
            int frameIndex = emptyFrame - memory.begin();
            memory[frameIndex] = pid;
            pageTable[pid][page] = frameIndex;
            cout << "Page " << page << " of Process " << pid << " added to Frame " << frameIndex << endl;
        } else {
            cout << "No free frames available for Page " << page << " of Process " << pid << endl;
        }
    }
}

cout << "\nFinal Memory Allocation:\n";
for (int i = 0; i < memory.size(); i++) {
    if (memory[i] == -1) {
        cout << "Frame " << i << ": Empty\n";
    } else {
        cout << "Frame " << i << ": Process " << memory[i] << endl;
    }
}

cout << "\nPage Tables:\n";
```

```
    for (auto& process : pageTable) {
        int pid = process.first;
        cout << "Process " << pid << " Page Table:\n";
        for (auto& entry : process.second) {
            cout << "  Page " << entry.first << " -> Frame " << entry.second << endl;
        }
    }
}

int main() {
    int memorySize = 16; //
    int pageSize = 4;
    unordered_map<int, vector<int>> processes = {
        {1, {0, 1}},
        {2, {0, 2}},
        {3, {1, 3}}
    };
    pagingTechnique(memorySize, pageSize, processes);
    return 0;
}
```

Output

Clear

```
Total Memory Frames: 4
Allocating pages for Process 3:
Page 1 of Process 3 added to Frame 0
Page 3 of Process 3 added to Frame 1
Allocating pages for Process 2:
Page 0 of Process 2 added to Frame 2
Page 2 of Process 2 added to Frame 3
Allocating pages for Process 1:
No free frames available for Page 0 of Process 1
No free frames available for Page 1 of Process 1

Final Memory Allocation:
Frame 0: Process 3
Frame 1: Process 3
Frame 2: Process 2
Frame 3: Process 2

Page Tables:
Process 2 Page Table:
  Page 2 -> Frame 3
  Page 0 -> Frame 2
Process 3 Page Table:
  Page 3 -> Frame 1
  Page 1 -> Frame 0
```