SSTF

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

using namespace std;

int main() {
    // Requests in the disk queue
    vector<int> requests = {55, 58, 39, 18, 90, 160, 150, 38, 189};
    int head = 100;   // Initial head position

    vector<bool> visited(requests.size(), false);
    int totalSeek = 0;
    vector<int> order;

    for (int i = 0; i < requests.size(); i++) {
        int idx = -1;
        int minDist = INT_MAX;

        // Find the request with minimum seek time from current head
        for (int j = 0; j < requests.size(); j++) {
            if (!visited[j]) {
                int dist = abs(head - requests[j]);
                if (dist < minDist) {
                    minDist = dist;
                    idx = j;
                }
            }
        }

        // Move head to that request
        visited[idx] = true;
        totalSeek += minDist;
        head = requests[idx];
        order.push_back(head);
    }

    // Output
    cout << "Seek Sequence: ";
    for (int r : order) cout << r << " ";
    cout << "\nTotal Seek Time: " << totalSeek << endl;

    return 0;
```

```
}
```

C-look

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

int main() {
    vector<int> requests = {18, 38, 39, 55, 58, 90, 150, 160, 189};
    int head = 100;

    sort(requests.begin(), requests.end());

    vector<int> seq;
    for (int x : requests) {
        if (x >= head) seq.push_back(x);
    }
    for (int x : requests) {
        if (x < head) seq.push_back(x);
    }

    int seek = 0, cur = head;
    for (int x : seq) {
        seek += abs(cur - x);
        cur = x;
    }

    cout << "Seek Sequence: ";
    for (int x : seq) cout << x << " ";
    cout << "\nTotal Seek: " << seek << endl;

    return 0;
}
```

Best fit

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

int main()
{
    vector<int> memory = {150, 350, 200, 400, 100, 300};
    vector<int> process = {115, 500, 210, 100, 375};
    vector<int> alloc(process.size(), -1);
    vector<int> memCopy = memory;

    // Best Fit Allocation
    for (int i = 0; i < process.size(); i++)
    {
        int best = -1;
        for (int j = 0; j < memCopy.size(); j++)
        {
            if (memCopy[j] >= process[i])
            {
                if (best == -1 || memCopy[j] < memCopy[best])
                {
                    best = j;
                }
            }
        }
        if (best != -1)
        {
            alloc[i] = best;
            memCopy[best] -= process[i];
        }
    }

    // Internal Fragmentation Calculation
    int internalFrag = 0;
    for (int i = 0; i < process.size(); i++)
    {
        if (alloc[i] != -1)
            internalFrag += memCopy[alloc[i]];
    }
```

```
    // Total Free Space
    int totalFree = 0;
    int largestBlock = 0;
    for (int j = 0; j < memCopy.size(); j++)
    {
        totalFree += memCopy[j];
        largestBlock = max(largestBlock, memCopy[j]);
    }

    // Not Allocated Processes
    vector<int> notAllocatedList;
    for (int i = 0; i < process.size(); i++)
    {
        if (alloc[i] == -1)
            notAllocatedList.push_back(process[i]);
    }

    // External Fragmentation = min(smallest not allocated process, total free)
    int externalFrag = 0;
    if (!notAllocatedList.empty())
    {
        int minNotAllocated = *min_element(notAllocatedList.begin(), notAllocatedList.end());
        externalFrag = min(minNotAllocated, totalFree);
    }

    // Output
    cout << "Total Internal Fragmentation: " << internalFrag << " KB\n";
    cout << "Total Free Space: " << totalFree << " KB\n";
    if (!notAllocatedList.empty())
        cout << "External Fragmentation: " << externalFrag << " KB\n";

    // Process allocation result
    for (int i = 0; i < process.size(); i++)
    {
        if (alloc[i] != -1)
            cout << "Process " << i + 1 << " (" << process[i] << " KB) -> Block " << alloc[i] + 1 <<
"\n";
        else
            cout << "Process " << i + 1 << " (" << process[i] << " KB) -> Not Allocated\n";
    }

    return 0;
}
```

Worst fit

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

int main() {
    vector<int> memory = {150, 350, 200, 400, 100, 300};
    vector<int> process = {115, 500, 210, 100, 375};
    vector<int> alloc(process.size(), -1);
    vector<int> memCopy = memory;

    // Worst Fit Allocation
    for (int i = 0; i < process.size(); i++) {
        int worst = -1;
        for (int j = 0; j < memCopy.size(); j++) {
            if (memCopy[j] >= process[i]) {
                if (worst == -1 || memCopy[j] > memCopy[worst]) {
                    worst = j;
                }
            }
        }
        if (worst != -1) {
            alloc[i] = worst;
            memCopy[worst] -= process[i];
        }
    }

    // Internal Fragmentation Calculation
    int internalFrag = 0;
    for (int i = 0; i < process.size(); i++) {
        if (alloc[i] != -1)
            internalFrag += memCopy[alloc[i]];
    }

    // Total Free Space
    int totalFree = 0;
    for (int j = 0; j < memCopy.size(); j++)
```

```
        totalFree += memCopy[j];

    // Unused Free Space Calculation (never used blocks)
    int unusedFree = 0;
    for (int j = 0; j < memCopy.size(); j++) {
        if (memCopy[j] == memory[j])
            unusedFree += memCopy[j];
    }

    // Not Allocated Processes
    vector<int> notAllocatedList;
    for (int i = 0; i < process.size(); i++) {
        if (alloc[i] == -1)
            notAllocatedList.push_back(process[i]);
    }

    // Smallest Not Allocated Process
    int minNotAllocated = 0;
    if (!notAllocatedList.empty())
        minNotAllocated = *min_element(notAllocatedList.begin(), notAllocatedList.end());

    // External Fragmentation = min(smallest not allocated, total free)
    int externalFrag = 0;
    if (!notAllocatedList.empty())
        externalFrag = min(minNotAllocated, totalFree);

    // Output
    cout << "Total Internal Fragmentation: " << internalFrag << " KB\n";
    cout << "Unused Free Space (never used blocks): " << unusedFree << " KB\n";
    if (!notAllocatedList.empty())
        cout << "External Fragmentation: " << externalFrag << " KB\n";

    // Display process allocation
    for (int i = 0; i < process.size(); i++) {
        if (alloc[i] != -1)
            cout << "Process " << i + 1 << " (" << process[i] << " KB) -> Block " << alloc[i] + 1 <<
"\n";
        else
            cout << "Process " << i + 1 << " (" << process[i] << " KB) -> Not Allocated\n";
    }

    return 0;
}
```

FIFO

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

int main() {
    vector<int> ref = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};
    int n = ref.size();
    int fm = 3;  // number of frames

    vector<int> frames;
    int pagefault = 0;
    int pagehit = 0;

    for (int i = 0; i < n; i++) {
        int page = ref[i];
        bool found = false;

        // check if page is already in frame
       // for (int i = 0; i < frames.size(); i++) { int f = frames[i]; }
        for (int f : frames) {
            if (f == page) {
                found = true;
                break;
            }
        }

        if (!found) {
            pagefault++;
            if (frames.size() == fm) {
                frames.erase(frames.begin());  // FIFO remove first
            }
            frames.push_back(page);
        } else {
            pagehit++;
        }

        // Print current frame status
```

```
        cout << "Frames after accessing " << page << ": ";
        for (int f : frames)
            cout << f << " ";
        cout << endl;
    }

    cout << "\nTotal Pages: " << n << endl;
    cout << "Total Page Hits: " << pagehit << endl;
    cout << "Total Page Misses: " << pagefault << endl;

    cout << "Hit Ratio: " << double(pagehit) / n << endl;
    cout << "Miss Ratio: " << double(pagefault) / n << endl;

    return 0;
}
```

LIFO

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

int main() {
    vector<int> ref = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};
    int n = ref.size();
    int fm = 3; // Number of frames

    vector<int> frames;
    int pagehit = 0, pagefault = 0;

    cout << "LRU Page Replacement Simulation\n\n";

    for (int i = 0; i < n; i++) {
        int page = ref[i];
        bool found = false;

        // Check if page is already in frames
        for (int j = 0; j < frames.size(); j++) {
            if (frames[j] == page) {
                found = true;
                break;
            }
```

```
        }

        if (found) {
            pagehit++;
            // Move this page to the most recently used position
            frames.erase(remove(frames.begin(), frames.end(), page), frames.end());
            frames.push_back(page);
        } else {
            pagefault++;
            if (frames.size() == fm) {
                // Remove least recently used page (first element)
                frames.erase(frames.begin());
            }
            frames.push_back(page);
        }

        // Display current frames (traditional loop)
        cout << "Frames after accessing page " << page << ": ";
        for (int j = 0; j < frames.size(); j++) {
            cout << frames[j] << " ";
        }
        cout << endl;
    }

    // Output statistics
    cout << "\nTotal Pages: " << n << endl;
    cout << "Total Page Hits: " << pagehit << endl;
    cout << "Total Page Misses: " << pagefault << endl;
    cout << "Hit Ratio: " << double(pagehit) / n << endl;
    cout << "Miss Ratio: " << double(pagefault) / n << endl;

    return 0;
}
```