

**CODE:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <cmath>

using namespace std;

int FCFS(const vector<int>& *fcfs\_requests*, int *fcfs\_head\_position*) {

    int fcfs\_seek\_time = 0;

    for (int fcfs\_request : *fcfs\_requests*) {

        fcfs\_seek\_time += abs(fcfs\_request - *fcfs\_head\_position*);

*fcfs\_head\_position* = fcfs\_request;

    }

    return fcfs\_seek\_time;

}

int SSTF(vector<int> *sstf\_requests*, int *sstf\_head\_position*) {

    int sstf\_seek\_time = 0;

    while (!*sstf\_requests*.empty()) {

        auto closest\_request = min\_element(*sstf\_requests*.begin(), *sstf\_requests*.end(),

                                           [*sstf\_head\_position*](int *a*, int *b*) {

                                               return abs(*a* - *sstf\_head\_position*) < abs(*b* - *sstf\_head\_position*);

                                           });

        sstf\_seek\_time += abs(\*closest\_request - *sstf\_head\_position*);

*sstf\_head\_position* = \*closest\_request;

*sstf\_requests*.erase(closest\_request);

    }

    return sstf\_seek\_time;

}

int SCAN(vector<int> *scan\_requests*, int *scan\_head\_position*, int *scan\_disk\_size*) {

    int scan\_seek\_time = 0;

*scan\_requests*.push\_back(0);

*scan\_requests*.push\_back(*scan\_disk\_size* - 1);

    sort(*scan\_requests*.begin(), *scan\_requests*.end());

    auto scan\_position = lower\_bound(*scan\_requests*.begin(), *scan\_requests*.end(), *scan\_head\_position*);

    for (auto it = scan\_position; it != *scan\_requests*.end(); ++it) {

        scan\_seek\_time += abs(\*it - *scan\_head\_position*);

*scan\_head\_position* = \*it;

    }

    for (auto it = scan\_position - 1; it >= *scan\_requests*.begin(); --it) {

        scan\_seek\_time += abs(\*it - *scan\_head\_position*);

*scan\_head\_position* = \*it;

    }

    return scan\_seek\_time;

}

int C\_SCAN(vector<int> *cscan\_requests*, int *cscan\_head\_position*, int *cscan\_disk\_size*) {

    int cscan\_seek\_time = 0;

*cscan\_requests*.push\_back(0);

*cscan\_requests*.push\_back(*cscan\_disk\_size* - 1);

    sort(*cscan\_requests*.begin(), *cscan\_requests*.end());

    auto cscan\_position = lower\_bound(*cscan\_requests*.begin(), *cscan\_requests*.end(), *cscan\_head\_position*);

    for (auto it = cscan\_position; it != *cscan\_requests*.end(); ++it) {

        cscan\_seek\_time += abs(\*it - *cscan\_head\_position*);

*cscan\_head\_position* = \*it;

    }

    cscan\_seek\_time += abs(*cscan\_disk\_size* - 1 - *cscan\_head\_position*);

*cscan\_head\_position* = 0;

    for (auto it = *cscan\_requests*.begin(); it < cscan\_position; ++it) {

        cscan\_seek\_time += abs(\*it - *cscan\_head\_position*);

*cscan\_head\_position* = \*it;

    }

    return cscan\_seek\_time;

}

int LOOK(vector<int> *look\_requests*, int *look\_head\_position*) {

    int look\_seek\_time = 0;

    sort(*look\_requests*.begin(), *look\_requests*.end());

    auto look\_position = lower\_bound(*look\_requests*.begin(), *look\_requests*.end(), *look\_head\_position*);

    for (auto it = look\_position; it != *look\_requests*.end(); ++it) {

        look\_seek\_time += abs(\*it - *look\_head\_position*);

*look\_head\_position* = \*it;

    }

    for (auto it = look\_position - 1; it >= *look\_requests*.begin(); --it) {

        look\_seek\_time += abs(\*it - *look\_head\_position*);

*look\_head\_position* = \*it;

    }

    return look\_seek\_time;

}

int C\_LOOK(vector<int> *clook\_requests*, int *clook\_head\_position*) {

    int clook\_seek\_time = 0;

    sort(*clook\_requests*.begin(), *clook\_requests*.end());

    auto clook\_position = lower\_bound(*clook\_requests*.begin(), *clook\_requests*.end(), *clook\_head\_position*);

    for (auto it = clook\_position; it != *clook\_requests*.end(); ++it) {

        clook\_seek\_time += abs(\*it - *clook\_head\_position*);

*clook\_head\_position* = \*it;

    }

    for (auto it = *clook\_requests*.begin(); it < clook\_position; ++it) {

        clook\_seek\_time += abs(\*it - *clook\_head\_position*);

*clook\_head\_position* = \*it;

    }

    return clook\_seek\_time;

}

int main() {

    int total\_requests, initial\_head\_position, disk\_capacity;

    cout << "Enter the number of requests: ";

    cin >> total\_requests;

    vector<int> input\_requests(total\_requests);

    cout << "Enter the requests: ";

    for (int i = 0; i < total\_requests; ++i) cin >> input\_requests[i];

    cout << "Enter the initial head position: ";

    cin >> initial\_head\_position;

    cout << "Enter the disk size: ";

    cin >> disk\_capacity;

    int fcfs\_seek = FCFS(input\_requests, initial\_head\_position);

    int sstf\_seek = SSTF(input\_requests, initial\_head\_position);

    int scan\_seek = SCAN(input\_requests, initial\_head\_position, disk\_capacity);

    int c\_scan\_seek = C\_SCAN(input\_requests, initial\_head\_position, disk\_capacity);

    int look\_seek = LOOK(input\_requests, initial\_head\_position);

    int c\_look\_seek = C\_LOOK(input\_requests, initial\_head\_position);

    cout << "FCFS Seek Time: " << fcfs\_seek << endl;

    cout << "SSTF Seek Time: " << sstf\_seek << endl;

    cout << "SCAN Seek Time: " << scan\_seek << endl;

    cout << "C-SCAN Seek Time: " << c\_scan\_seek << endl;

    cout << "LOOK Seek Time: " << look\_seek << endl;

    cout << "C-LOOK Seek Time: " << c\_look\_seek << endl;

    int min\_seek\_time = min({fcfs\_seek, sstf\_seek, scan\_seek, c\_scan\_seek, look\_seek, c\_look\_seek});

    cout << "Most efficient algorithm: ";

    if (min\_seek\_time == fcfs\_seek) cout << "FCFS";

    else if (min\_seek\_time == sstf\_seek) cout << "SSTF";

    else if (min\_seek\_time == scan\_seek) cout << "SCAN";

    else if (min\_seek\_time == c\_scan\_seek) cout << "C-SCAN";

    else if (min\_seek\_time == look\_seek) cout << "LOOK";

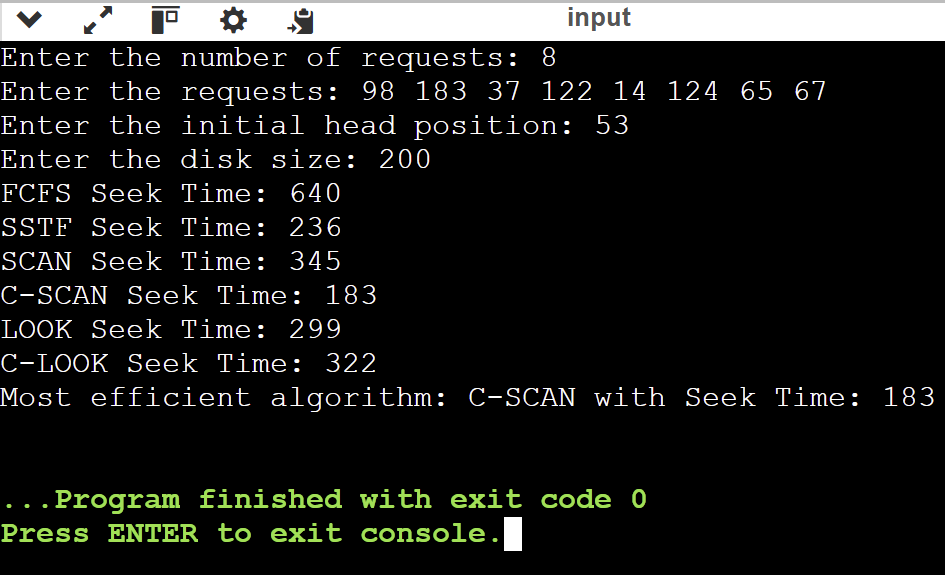
    else if (min\_seek\_time == c\_look\_seek) cout << "C-LOOK";

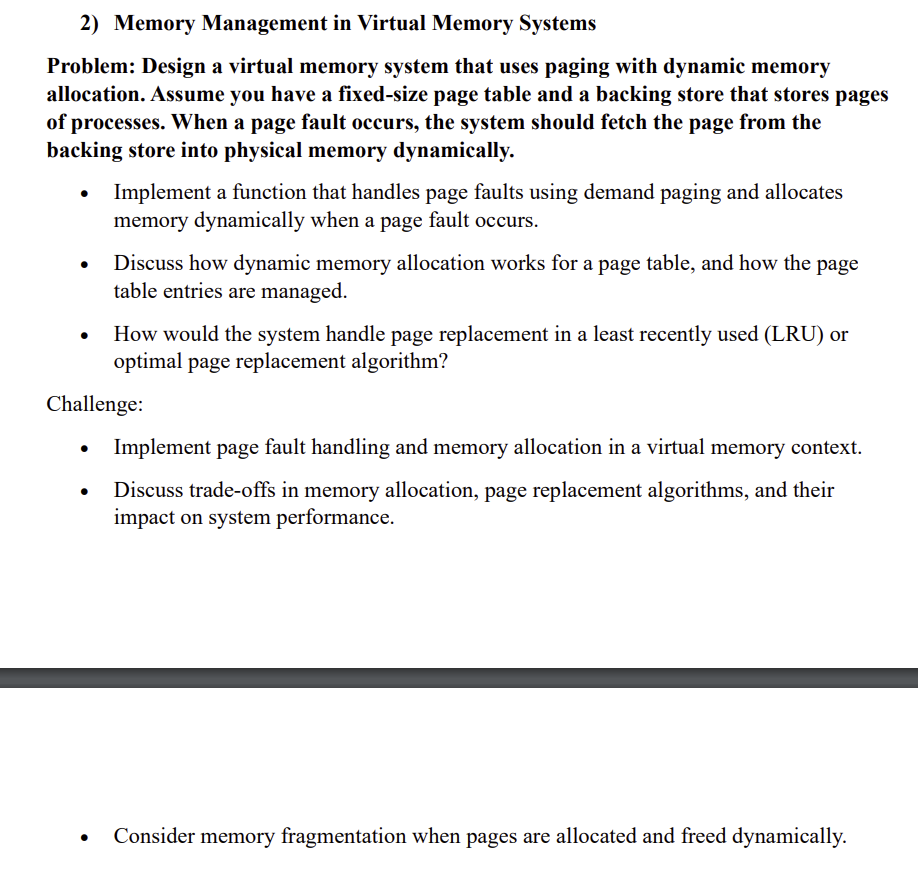
    cout << " with Seek Time: " << min\_seek\_time << endl;

    return 0;

}

**OUTPUT:**

****

****

**CODE:**

#include <iostream>

#include <unordered\_map>

#include <list>

#include <vector>

using namespace std;

class MemoryManager {

    int maxPageTableSize, frameSize, memoryCapacity;

    vector<int> secondaryStorage;

    unordered\_map<int, int> pageDirectory;

    list<int> recentlyUsedPages;

    unordered\_map<int, list<int>::iterator> pageLookup;

    vector<int> mainMemory;

public:

    MemoryManager(int *maxTableSize*, int *frameBytes*, int *totalMemory*, const *vector*<int>& *storage*)

        : maxPageTableSize(maxTableSize), frameSize(frameBytes), memoryCapacity(totalMemory), secondaryStorage(storage) {

        mainMemory.resize(memoryCapacity / frameSize, -1);

    }

    void loadPage(int *virtualAddress*) {

        int targetPage = virtualAddress / frameSize;

        if (pageDirectory.find(targetPage) == pageDirectory.end()) {

            cout << "Page fault for page: " << targetPage << endl;

            processPageFault(targetPage);

        } else {

            cout << "Page " << targetPage << " found in frame: " << pageDirectory[targetPage] << endl;

            refreshLRU(targetPage);

        }

    }

    void processPageFault(int *page*) {

        int freeFrame = findFreeMemoryFrame();

        if (freeFrame == -1) {

            freeFrame = performPageReplacement();

        }

        pageDirectory[page] = freeFrame;

        mainMemory[freeFrame] = secondaryStorage[page];

        cout << "Page " << page << " loaded into frame " << freeFrame << endl;

        refreshLRU(page);

    }

    int findFreeMemoryFrame() {

        for (int i = 0; i < mainMemory.size(); i++) {

            if (mainMemory[i] == -1) return i;

        }

        return -1;

    }

    int performPageReplacement() {

        int leastUsedPage = recentlyUsedPages.back();

        recentlyUsedPages.pop\_back();

        int replacedFrame = pageDirectory[leastUsedPage];

        pageDirectory.erase(leastUsedPage);

        cout << "Page " << leastUsedPage << " removed from frame " << replacedFrame << endl;

        return replacedFrame;

    }

    void refreshLRU(int *page*) {

        if (pageLookup.find(page) != pageLookup.end()) {

            recentlyUsedPages.erase(pageLookup[page]);

        }

        recentlyUsedPages.push\_front(page);

        pageLookup[page] = recentlyUsedPages.begin();

    }

};

int main() {

    int tableLimit, frameBytes, memorySize;

    cout << "Enter page table capacity: ";

    cin >> tableLimit;

    cout << "Enter size of each page frame: ";

    cin >> frameBytes;

    cout << "Enter total memory size: ";

    cin >> memorySize;

    int secondaryStorageSize;

    cout << "Enter size of secondary storage (backing store): ";

    cin >> secondaryStorageSize;

    vector<int> secondaryStorage(secondaryStorageSize);

    cout << "Enter data for each page in the secondary storage: ";

    for (int i = 0; i < secondaryStorageSize; i++) {

        cin >> secondaryStorage[i];

    }

    MemoryManager memSystem(tableLimit, frameBytes, memorySize, secondaryStorage);

    int addressCount;

    cout << "Enter number of virtual addresses to access: ";

    cin >> addressCount;

    cout << "Enter the virtual addresses: ";

    for (int i = 0; i < addressCount; i++) {

        int address;

        cin >> address;

        memSystem.loadPage(address);

    }

    return 0;

}

**OUTPUT:**

