OPERATING SYSTEMS

(CECSC09 -I)



Submitted By:- **Ashish Kumar 2019UCO1518**

**INDEX**

|  |  |
| --- | --- |
| **S.No** | **TOPIC** |
| 1 | [Practice Linux shell commands.](#p1) |
| 2 | [Write C programs using fork(), getpid(), getppid() and exec() system calls.](#p2) |
| 3 | [Write a C program to represent a family of processes as a tree.](#p3) |
| 4 | [CPU Scheduling Algorithms - FCFS](#p4) |
| 5 | [CPU Scheduling Algorithms - SFJ](#p5) |
| 6 | [CPU Scheduling Algorithms – Round Robin](#p6) |
| 7 | [CPU Scheduling Algorithms – preempititve priority scheduling](#p7) |
| 8 | [Multilevel Feedback Queue scheduling algorithm.](#p8) |
| 9 | [Program to stimulate deadlock detection](#p9) |
| 10 | [Program to stimulate deadlock avoidance](#p10) |
| 11 | [program to simulate best-fit contiguous memory allocation.](#p11) |
| 12 | [**Program to stimulate FIFO page replacement algorithm**](#p12) |
| 13 | [**Program to stimulate LRU page replacement algorithm**](#p13) |
| 14 | [program to simulate Second Chance page replacement algorithm.](#p14) |
| 15 | [program to simulate Enhanced Second Chance page replacement algorithm.](#p15) |
| 16 | [program to simulate LFU page replacement algorithm.](#p16) |
| 17 | [program to simulate FCFS disk scheduling algorithm.](#p17) |
| 18 | [program to simulate SSTF disk scheduling algorithm.](#p18) |
| 19 | [program to simulate C-SCAN disk scheduling algorithm](#p19) |
| 20 | [program to simulate LOOK disk scheduling algorithm.](#p20) |

# Program 1 : Linux Shell Commands

|  |
| --- |
| **mkdir**: Used to create a directory if not already exist. It accepts the directory name as an input parameter.  **rmdir**: It is used to delete a directory if it is empty.  **touch**: Used to create or update a file.  **cat**: It is generally used to concatenate the files. It gives the output on the standard output.  **more**: It is a filter for paging through text one screenful at a time.  **cd**: Used to change the directory.  **touch**: Used to create or update a file.  **ls**: To get the list of all the files or folders.  **rm**: Used to remove files or directories.  **wc**: Used to count the number of characters, words in a file.  **sort**: This command is used to sort the contents of files.  **grep**: This command is used to search for the specified text in a file.  **head**: Used to print the first N lines of a file. It accepts N as input and the default value of N is 10.  **tail**: The tail command displays the last part (10 lines by default) of one or more files or piped data. It can be also used to monitor the file changes in real time |

### 

### PROGRAM 2 : Write C programs using fork(), getpid(), getppid() and exec() system calls.

**CODE:**

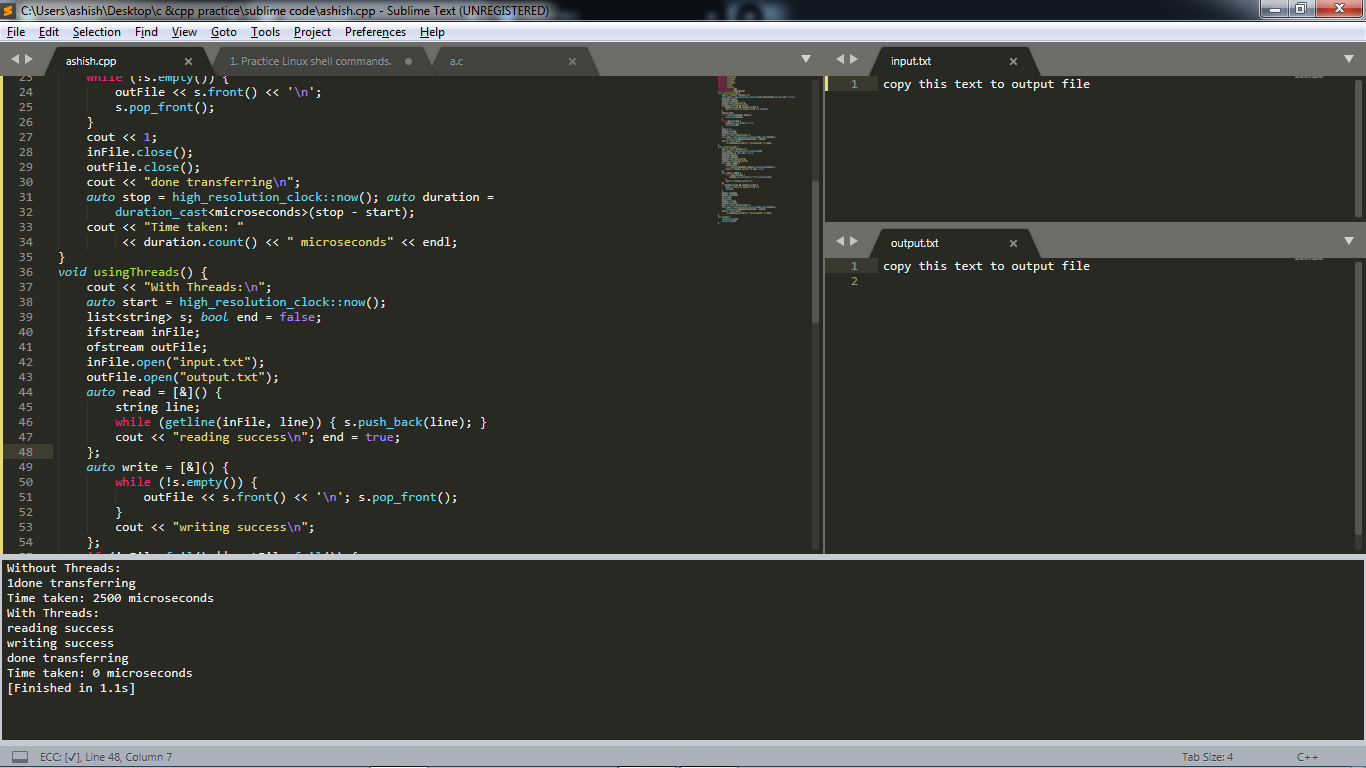
|  |
| --- |
| FILE 1 :  #include <iostream>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  using namespace std;  int main(int argc, char \*argv[])  {  cout << "Process Creation and Termination\n";  pid\_t id1, id2;  id1 = fork();  id2 = fork();  if (id1 == 0 && id2 > 0)  {  cout << "This is First child Process; pid= " << getpid() << "  Parent’s pid=" << getppid() << endl;  exit(0);  }  else if (id1 > 0 && id2 == 0)  {  cout << "This is Second child Process; pid= " << getpid() <  < "Parent’s pid=" << getppid() << endl;  exit(0);  }  else if (id1 == 0 && id2 == 0)  {  cout << "This is Third child Process; pid= " << getpid() << "  Parent’s pid=" << getppid() << endl;  exit(0);  }  else if (id1 > 0 && id2 > 0)  {  cout << "This is Parent Process; pid= " << getpid() << endl;  pid\_t Id = wait(NULL);  cout << "Parent Process returns from wait after process "  << Id << " terminates\n";  cout << "Calling exec() system call\n";  char \*args[] = {"a", "b", NULL};    execv("./prog", args);  }  else  cout << "Error\n";  return 0;  }  FILE2 :  #include <iostream>  #include <unistd.h>  #include <sys/wait.h>  #include <stdlib.h>  using namespace std;  int main(int argc, char \*argv[])  {  cout << "This is the replaced file\n";  cout << "Process Id= " << getpid() << endl;  return 0;  } |

**PROGRAM : 3 - Write a C program to represent a family of processes as a tree.**

**Code:**

|  |  |
| --- | --- |
| #include <iostream>  #include <fstream>  #include <thread>  #include <cstdlib>  #include <list>  #include <chrono>  using namespace std;  using namespace std::chrono;  void withoutThreads() {  cout << "Without Threads:\n";  auto start = high\_resolution\_clock::now(); list<string> s; bool end = false;  ifstream inFile;  ofstream outFile;  inFile.open("input.txt");  outFile.open("output.txt");  if (inFile.fail() || outFile.fail()) {  cout << "error in opening files\n"; exit(1);  }  string line;  while (getline(inFile, line)) {  s.push\_back(line);  }  while (!s.empty()) {  outFile << s.front() << '\n';  s.pop\_front();  }  cout << 1;  inFile.close();  outFile.close();  cout << "done transferring\n";  auto stop = high\_resolution\_clock::now(); auto duration =  duration\_cast<microseconds>(stop - start);  cout << "Time taken: "  << duration.count() << " microseconds" << endl;  }  void usingThreads() {  cout << "With Threads:\n";  auto start = high\_resolution\_clock::now();  list<string> s; bool end = false;  ifstream inFile;  ofstream outFile;  inFile.open("input.txt");  outFile.open("output.txt");  auto read = [&]() {  string line;  while (getline(inFile, line)) { s.push\_back(line); }  cout << "reading success\n"; end = true;  };  auto write = [&]() {  while (!s.empty()) {  outFile << s.front() << '\n'; s.pop\_front();  }  cout << "writing success\n";  }; | if (inFile.fail() || outFile.fail()) {  cout << "error in opening files\n";  exit(1);  }  thread t1(read);  thread t2(write);  t2.join();  t1.join();  inFile.close();  outFile.close();  cout << "done transferring\n";  auto stop = high\_resolution\_clock::now(); auto duration =  duration\_cast<microseconds>(stop - start);  cout << "Time taken: "  << duration.count() << " microseconds" << endl;  }  int main() {  withoutThreads();  usingThreads();  } |

**OUTPUT:**

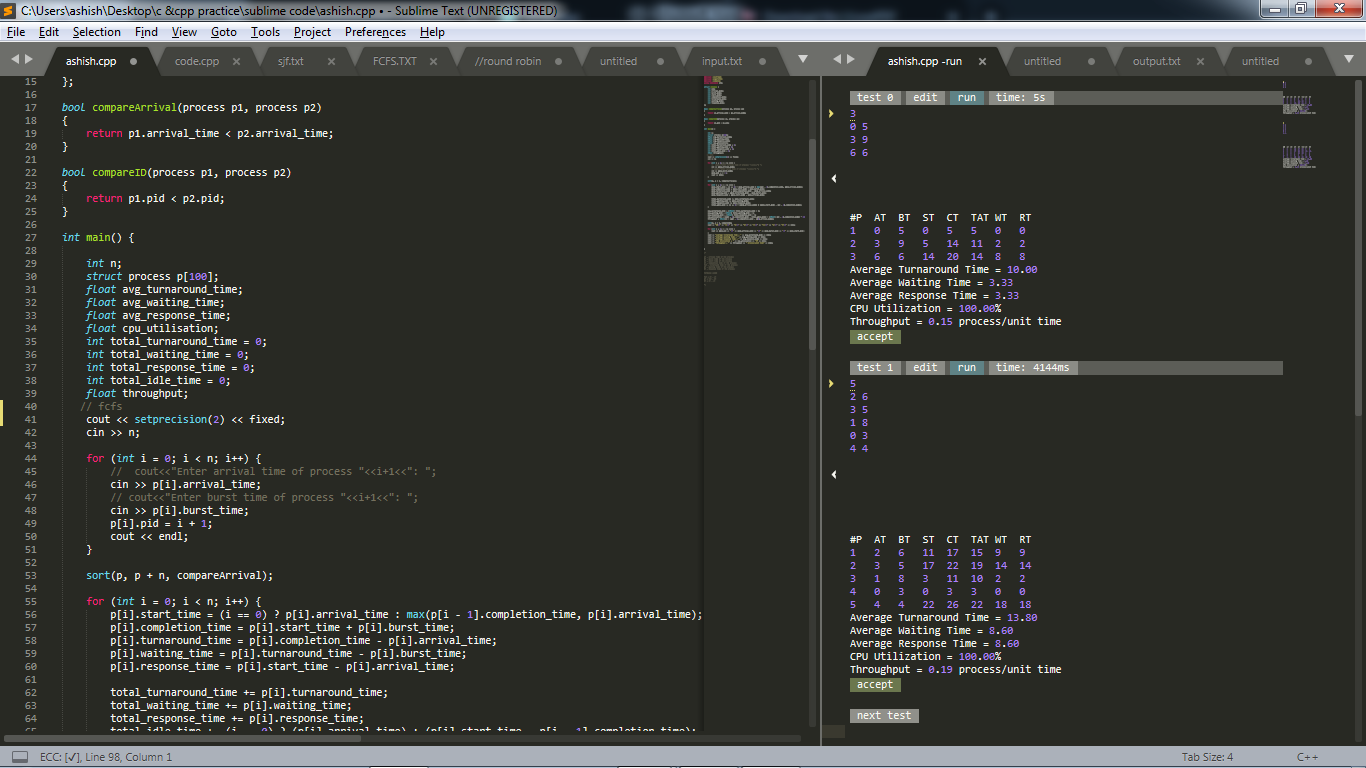
****

**PROGRAM 4 -  CPU SCHEDULING ALGORITHMS – FCFS**

**Code:**

|  |  |
| --- | --- |
| #include <iostream>  #include <algorithm>  #include <iomanip>  using namespace std;  struct process {  int pid;  int arrival\_time;  int burst\_time;  int start\_time;  int completion\_time;  int turnaround\_time;  int waiting\_time;  int response\_time;  };  bool compareArrival(process p1, process p2)  {  return p1.arrival\_time < p2.arrival\_time;  }  bool compareID(process p1, process p2)  {  return p1.pid < p2.pid;  }  int main() {  int n;  struct process p[100];  float avg\_turnaround\_time;  float avg\_waiting\_time;  float avg\_response\_time;  float cpu\_utilisation;  int total\_turnaround\_time = 0;  int total\_waiting\_time = 0;  int total\_response\_time = 0;  int total\_idle\_time = 0;  float throughput;  // fcfs  cout << setprecision(2) << fixed;  cin >> n;  for (int i = 0; i < n; i++) {  cin >> p[i].arrival\_time;  cin >> p[i].burst\_time;  p[i].pid = i + 1;  cout << endl;  }  sort(p, p + n, compareArrival);  for (int i = 0; i < n; i++) {  p[i].start\_time = (i == 0) ? p[i].arrival\_time : max(p[i - 1].completion\_time, p[i].arrival\_time);  p[i].completion\_time = p[i].start\_time + p[i].burst\_time;  p[i].turnaround\_time = p[i].completion\_time - p[i].arrival\_time;  p[i].waiting\_time = p[i].turnaround\_time - p[i].burst\_time;  p[i].response\_time = p[i].start\_time - p[i].arrival\_time;  total\_turnaround\_time += p[i].turnaround\_time;  total\_waiting\_time += p[i].waiting\_time;  total\_response\_time += p[i].response\_time;  total\_idle\_time += (i == 0) ? (p[i].arrival\_time) : (p[i].start\_time - p[i - 1].completion\_time);  }  avg\_turnaround\_time = (float) total\_turnaround\_time / n;  avg\_waiting\_time = (float) total\_waiting\_time / n;  avg\_response\_time = (float) total\_response\_time / n;  cpu\_utilisation = ((p[n - 1].completion\_time - total\_idle\_time) / (float) p[n - 1].completion\_time) \* 100;  throughput = float(n) / (p[n - 1].completion\_time - p[0].arrival\_time);  sort(p, p + n, compareID);  cout << "#P\t" << "AT\t" << "BT\t" << "ST\t" << "CT\t" << "TAT\t" << "WT\t" << "RT\t" << endl;  for (int i = 0; i < n; i++) {  cout << p[i].pid << "\t" << p[i].arrival\_time << "\t" << p[i].burst\_time << "\t" << p[i].start\_time << "\t" << p[i].completion\_time << "\t" << p[i].turnaround\_time << "\t" << p[i].waiting\_time << "\t" << p[i].response\_time << "\t" << endl;  }  cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;  cout << "Average Waiting Time = " << avg\_waiting\_time << endl;  cout << "Average Response Time = " << avg\_response\_time << endl;  cout << "CPU Utilization = " << cpu\_utilisation << "%" << endl;  cout << "Throughput = " << throughput << " process/unit time" << endl;  return 0;  }  **/\***  **AT - Arrival Time of the process**  **BT - Burst time of the process**  **ST - Start time of the process**  **CT - Completion time of the process**  **TAT - Turnaround time of the process**  **WT - Waiting time of the process**  **RT - Response time of the process**  **Formulas used:**  **TAT = CT - AT**  **WT = TAT - BT**  **RT = ST - AT**  **\*/** |  |

**Output:**

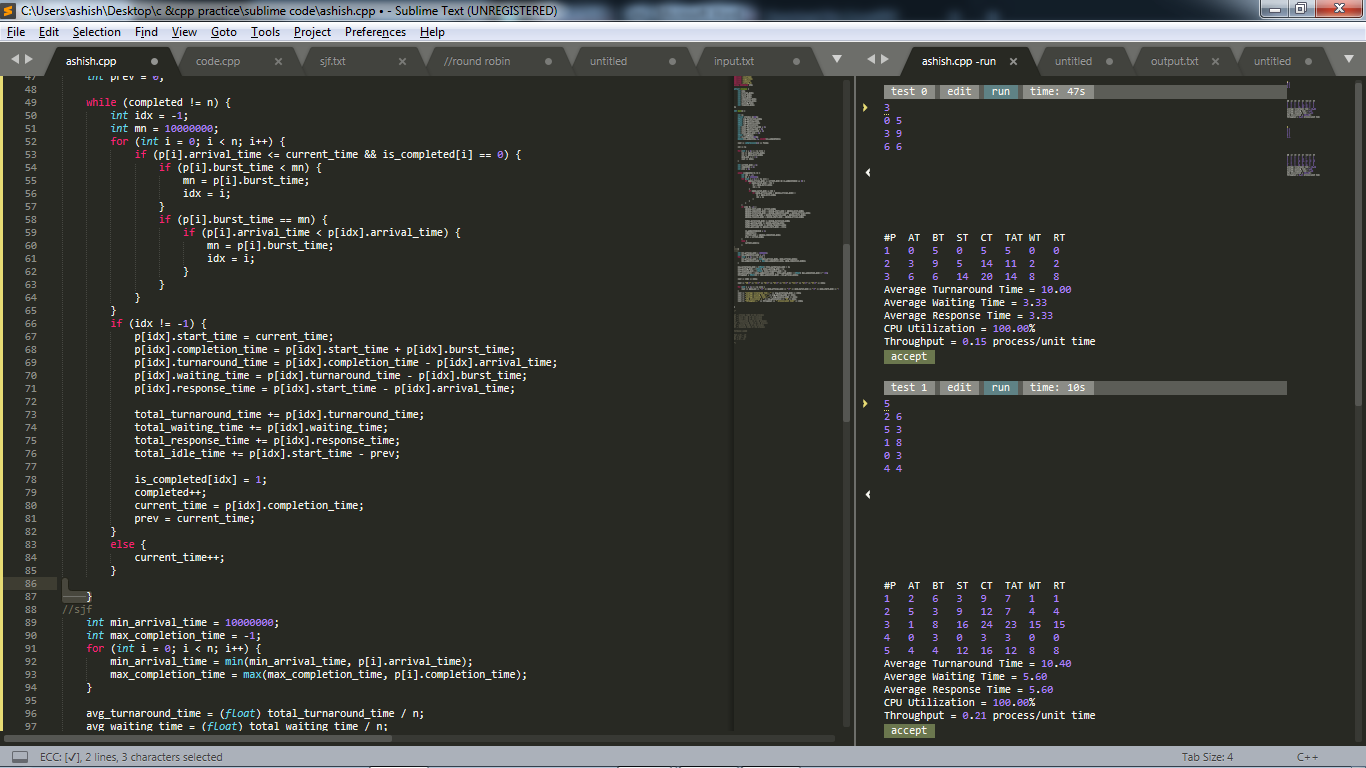
****

**Program 5 : CPU SCHEDULING ALGORITHMS – SJF**

**Code :**

|  |  |
| --- | --- |
| #include <iostream>  #include <algorithm>  #include <iomanip>  #include <string.h>  using namespace std;  struct process {  int pid;  int arrival\_time;  int burst\_time;  int start\_time;  int completion\_time;  int turnaround\_time;  int waiting\_time;  int response\_time;  };  int main() {  int n;  struct process p[100];  float avg\_turnaround\_time;  float avg\_waiting\_time;  float avg\_response\_time;  float cpu\_utilisation;  int total\_turnaround\_time = 0;  int total\_waiting\_time = 0;  int total\_response\_time = 0;  int total\_idle\_time = 0;  float throughput;  int is\_completed[100];  memset(is\_completed, 0, sizeof(is\_completed));  cout << setprecision(2) << fixed;  cin >> n;  for (int i = 0; i < n; i++) {  cin >> p[i].arrival\_time;  cin >> p[i].burst\_time;  p[i].pid = i + 1;  cout << endl;  }  int current\_time = 0;  int completed = 0;  int prev = 0;  while (completed != n) {  int idx = -1;  int mn = 10000000;  for (int i = 0; i < n; i++) {  if (p[i].arrival\_time <= current\_time && is\_completed[i] == 0) {  if (p[i].burst\_time < mn) {  mn = p[i].burst\_time;  idx = i;  }  if (p[i].burst\_time == mn) {  if (p[i].arrival\_time < p[idx].arrival\_time) {  mn = p[i].burst\_time;  idx = i;  }  }  }  }  if (idx != -1) {  p[idx].start\_time = current\_time;  p[idx].completion\_time = p[idx].start\_time + p[idx].burst\_time;  p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;  p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;  p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;  total\_turnaround\_time += p[idx].turnaround\_time;  total\_waiting\_time += p[idx].waiting\_time;  total\_response\_time += p[idx].response\_time;  total\_idle\_time += p[idx].start\_time - prev;  is\_completed[idx] = 1;  completed++;  current\_time = p[idx].completion\_time;  prev = current\_time;  }  else {  current\_time++;  }  }  int min\_arrival\_time = 10000000;  int max\_completion\_time = -1;  for (int i = 0; i < n; i++) {  min\_arrival\_time = min(min\_arrival\_time, p[i].arrival\_time);  max\_completion\_time = max(max\_completion\_time, p[i].completion\_time);  }  avg\_turnaround\_time = (float) total\_turnaround\_time / n;  avg\_waiting\_time = (float) total\_waiting\_time / n;  avg\_response\_time = (float) total\_response\_time / n;  cpu\_utilisation = ((max\_completion\_time - total\_idle\_time) / (float) max\_completion\_time ) \* 100;  throughput = float(n) / (max\_completion\_time - min\_arrival\_time);  cout << endl << endl;  cout << "#P\t" << "AT\t" << "BT\t" << "ST\t" << "CT\t" << "TAT\t" << "WT\t" << "RT\t" << endl;  for (int i = 0; i < n; i++) {  cout << p[i].pid << "\t" << p[i].arrival\_time << "\t" << p[i].burst\_time << "\t" << p[i].start\_time << "\t" << p[i].completion\_time << "\t" << p[i].turnaround\_time << "\t" << p[i].waiting\_time << "\t" << p[i].response\_time << "\t" << endl;  }  cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;  cout << "Average Waiting Time = " << avg\_waiting\_time << endl;  cout << "Average Response Time = " << avg\_response\_time << endl;  cout << "CPU Utilization = " << cpu\_utilisation << "%" << endl;  cout << "Throughput = " << throughput << " process/unit time" << endl;  }  **/\***  **AT - Arrival Time of the process**  **BT - Burst time of the process**  **ST - Start time of the process**  **CT - Completion time of the process**  **TAT - Turnaround time of the process**  **WT - Waiting time of the process**  **RT - Response time of the process**  **Formulas used:**  **TAT = CT - AT**  **WT = TAT - BT**  **RT = ST - AT**  **\*/** |  |

**Output:**

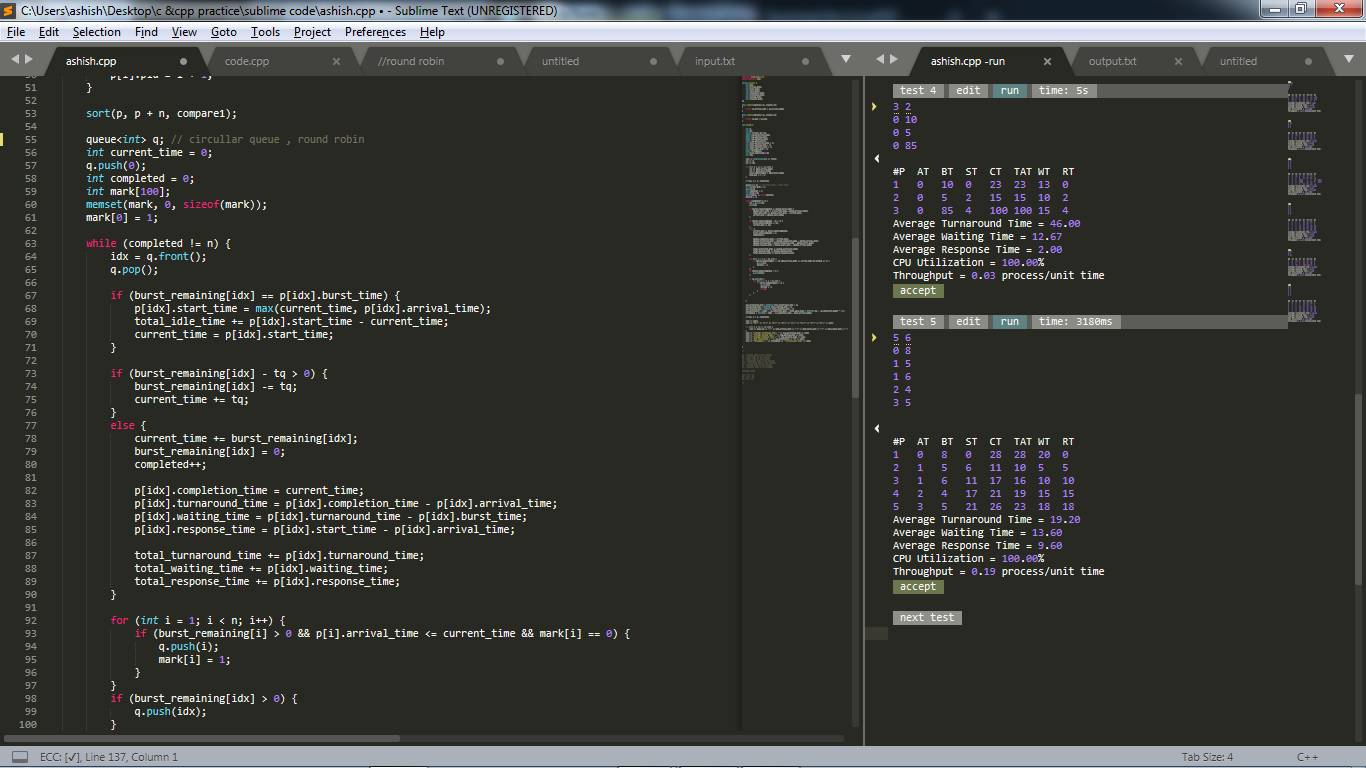


**PROGRAM 6 -  CPU SCHEDULING ALGORITHMS – ROUND ROBIN**

**Code:**

|  |  |
| --- | --- |
| #include <bits/stdc++.h>  using namespace std;  struct process {  int pid;  int arrival\_time;  int burst\_time;  int start\_time;  int completion\_time;  int turnaround\_time;  int waiting\_time;  int response\_time;  };  bool compare1(process p1, process p2)  {  return p1.arrival\_time < p2.arrival\_time;  }  bool compare2(process p1, process p2)  {  return p1.pid < p2.pid;  }  int main() {  int n;  int tq;  struct process p[100];  float avg\_turnaround\_time;  float avg\_waiting\_time;  float avg\_response\_time;  float cpu\_utilisation;  int total\_turnaround\_time = 0;  int total\_waiting\_time = 0;  int total\_response\_time = 0;  int total\_idle\_time = 0;  float throughput;  int burst\_remaining[100];  int idx;  cout << setprecision(2) << fixed;  cin >> n;  cin >> tq;  for (int i = 0; i < n; i++) {  cin >> p[i].arrival\_time;  cin >> p[i].burst\_time;  burst\_remaining[i] = p[i].burst\_time;  p[i].pid = i + 1;  }  sort(p, p + n, compare1);  queue<int> q; // circullar queue , round robin  int current\_time = 0;  q.push(0);  int completed = 0;  int mark[100];  memset(mark, 0, sizeof(mark));  mark[0] = 1;  while (completed != n) {  idx = q.front();  q.pop();  if (burst\_remaining[idx] == p[idx].burst\_time) {  p[idx].start\_time = max(current\_time, p[idx].arrival\_time);  total\_idle\_time += p[idx].start\_time - current\_time;  current\_time = p[idx].start\_time;  }  if (burst\_remaining[idx] - tq > 0) {  burst\_remaining[idx] -= tq;  current\_time += tq;  }  else {  current\_time += burst\_remaining[idx];  burst\_remaining[idx] = 0;  completed++;  p[idx].completion\_time = current\_time;  p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;  p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;  p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;  total\_turnaround\_time += p[idx].turnaround\_time;  total\_waiting\_time += p[idx].waiting\_time;  total\_response\_time += p[idx].response\_time;  }  for (int i = 1; i < n; i++) {  if (burst\_remaining[i] > 0 && p[i].arrival\_time <= current\_time && mark[i] == 0) {  q.push(i);  mark[i] = 1;  }  }  if (burst\_remaining[idx] > 0) {  q.push(idx);  }  if (q.empty()) {  for (int i = 1; i < n; i++) {  if (burst\_remaining[i] > 0) {  q.push(i);  mark[i] = 1;  break;  }  }  }  }  avg\_turnaround\_time = (float) total\_turnaround\_time / n;  avg\_waiting\_time = (float) total\_waiting\_time / n;  avg\_response\_time = (float) total\_response\_time / n;  cpu\_utilisation = ((p[n - 1].completion\_time - total\_idle\_time) / (float) p[n - 1].completion\_time) \* 100;  throughput = float(n) / (p[n - 1].completion\_time - p[0].arrival\_time);  sort(p, p + n, compare2);  cout << endl;  cout << "#P\t" << "AT\t" << "BT\t" << "ST\t" << "CT\t" << "TAT\t" << "WT\t" << "RT\t" << endl;  for (int i = 0; i < n; i++) {  cout << p[i].pid << "\t" << p[i].arrival\_time << "\t" << p[i].burst\_time << "\t" << p[i].start\_time << "\t" << p[i].completion\_time << "\t" << p[i].turnaround\_time << "\t" << p[i].waiting\_time << "\t" << p[i].response\_time << "\t" << endl;  }  cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;  cout << "Average Waiting Time = " << avg\_waiting\_time << endl;  cout << "Average Response Time = " << avg\_response\_time << endl;  cout << "CPU Utilization = " << cpu\_utilisation << "%" << endl;  cout << "Throughput = " << throughput << " process/unit time" << endl;  }  **/\***  **AT - Arrival Time of the process**  **BT - Burst time of the process**  **ST - Start time of the process**  **CT - Completion time of the process**  **TAT - Turnaround time of the process**  **WT - Waiting time of the process**  **RT - Response time of the process**  **Formulas used:**  **TAT = CT - AT**  **WT = TAT - BT**  **RT = ST – AT\*/** |  |

**Output:**

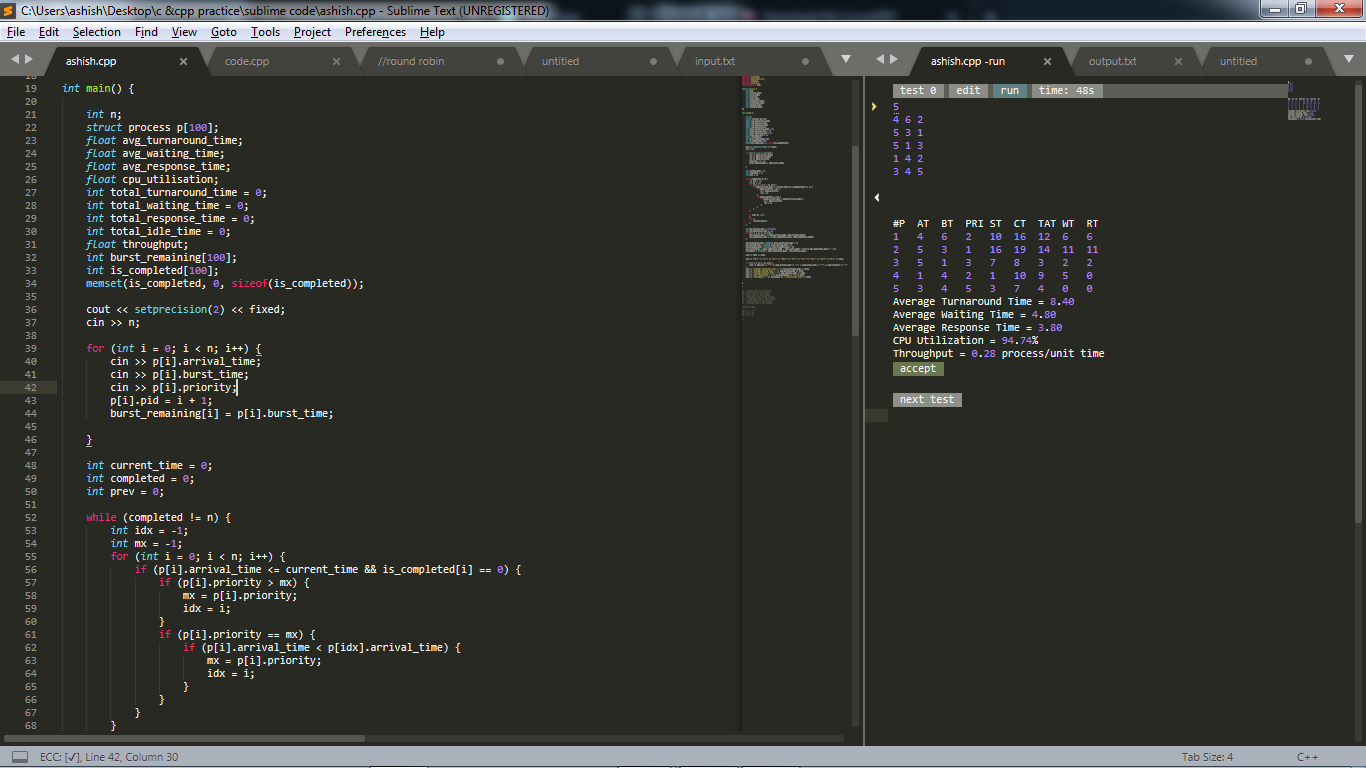
****

**PROGRAM 7 -  CPU SCHEDULING ALGORITHMS – PREEMPTITIVE PRIORITY SCHEDULING**

**Code:**

|  |  |
| --- | --- |
| #include <iostream>  #include <bits/stdc++.h>  #include <iomanip>  #include <string.h>  using namespace std;  struct process {  int pid;  int arrival\_time;  int burst\_time;  int priority;  int start\_time;  int completion\_time;  int turnaround\_time;  int waiting\_time;  int response\_time;  };  int main() {  int n;  struct process p[100];  float avg\_turnaround\_time;  float avg\_waiting\_time;  float avg\_response\_time;  float cpu\_utilisation;  int total\_turnaround\_time = 0;  int total\_waiting\_time = 0;  int total\_response\_time = 0;  int total\_idle\_time = 0;  float throughput;  int burst\_remaining[100];  int is\_completed[100];  memset(is\_completed, 0, sizeof(is\_completed));  cout << setprecision(2) << fixed;  cin >> n;  for (int i = 0; i < n; i++) {  cin >> p[i].arrival\_time;  cin >> p[i].burst\_time;  cin >> p[i].priority;  p[i].pid = i + 1;  burst\_remaining[i] = p[i].burst\_time;  }  int current\_time = 0;  int completed = 0;  int prev = 0;  while (completed != n) {  int idx = -1;  int mx = -1;  for (int i = 0; i < n; i++) {  if (p[i].arrival\_time <= current\_time && is\_completed[i] == 0) {  if (p[i].priority > mx) {  mx = p[i].priority;  idx = i;  }  if (p[i].priority == mx) {  if (p[i].arrival\_time < p[idx].arrival\_time) {  mx = p[i].priority;  idx = i;  }  }  }  }  if (idx != -1) {  if (burst\_remaining[idx] == p[idx].burst\_time) {  p[idx].start\_time = current\_time;  total\_idle\_time += p[idx].start\_time - prev;  }  burst\_remaining[idx] -= 1;  current\_time++;  prev = current\_time;  if (burst\_remaining[idx] == 0) {  p[idx].completion\_time = current\_time;  p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;  p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;  p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;    total\_turnaround\_time += p[idx].turnaround\_time;  total\_waiting\_time += p[idx].waiting\_time;  total\_response\_time += p[idx].response\_time;  is\_completed[idx] = 1;  completed++;  }  }  else {  current\_time++;  }  }  int min\_arrival\_time = 10000000;  int max\_completion\_time = -1;  for (int i = 0; i < n; i++) {  min\_arrival\_time = min(min\_arrival\_time, p[i].arrival\_time);  max\_completion\_time = max(max\_completion\_time, p[i].completion\_time);  }  avg\_turnaround\_time = (float) total\_turnaround\_time / n;  avg\_waiting\_time = (float) total\_waiting\_time / n;  avg\_response\_time = (float) total\_response\_time / n;  cpu\_utilisation = ((max\_completion\_time - total\_idle\_time) / (float) max\_completion\_time ) \* 100;  throughput = float(n) / (max\_completion\_time - min\_arrival\_time);  cout << endl << endl;  cout << "#P\t" << "AT\t" << "BT\t" << "PRI\t" << "ST\t" << "CT\t" << "TAT\t" << "WT\t" << "RT\t" << endl;  for (int i = 0; i < n; i++) {  cout << p[i].pid << "\t" << p[i].arrival\_time << "\t" << p[i].burst\_time << "\t" << p[i].priority << "\t" << p[i].start\_time << "\t" << p[i].completion\_time << "\t" << p[i].turnaround\_time << "\t" << p[i].waiting\_time << "\t" << p[i].response\_time << "\t" << endl;  }  cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;  cout << "Average Waiting Time = " << avg\_waiting\_time << endl;  cout << "Average Response Time = " << avg\_response\_time << endl;  cout << "CPU Utilization = " << cpu\_utilisation << "%" << endl;  cout << "Throughput = " << throughput << " process/unit time" << endl;  }  **/\***  **AT - Arrival Time of the process**  **BT - Burst time of the process**  **ST - Start time of the process**  **CT - Completion time of the process**  **TAT - Turnaround time of the process**  **WT - Waiting time of the process**  **RT - Response time of the process**  **Formulas used:**  **TAT = CT - AT**  **WT = TAT - BT**  **RT = ST - AT**  **\*/** |  |

**Output:**

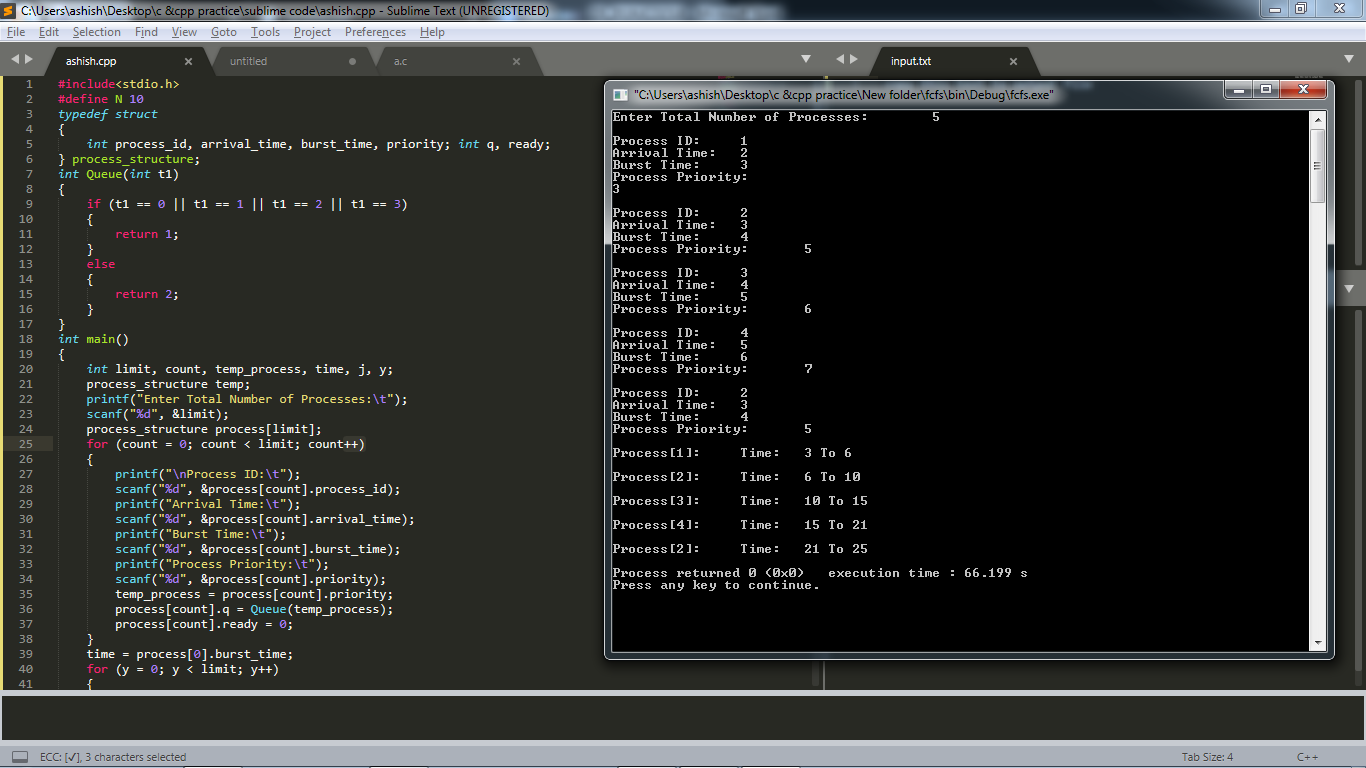
****

**PROGRAM: 8 simulate Multilevel Feedback Queue scheduling algorithm.**

**Code :**

|  |
| --- |
| #include<stdio.h>  #define N 10  typedef struct  {  int process\_id, arrival\_time, burst\_time, priority; int q, ready;  } process\_structure;  int Queue(int t1)  {  if (t1 == 0 || t1 == 1 || t1 == 2 || t1 == 3)  {  return 1;  }  else  {  return 2;  }  }  int main()  {  int limit, count, temp\_process, time, j, y;  process\_structure temp;  printf("Enter Total Number of Processes:\t");  scanf("%d", &limit);  process\_structure process[limit];  for (count = 0; count < limit; count++)  {  printf("\nProcess ID:\t");  scanf("%d", &process[count].process\_id);  printf("Arrival Time:\t");  scanf("%d", &process[count].arrival\_time);  printf("Burst Time:\t");  scanf("%d", &process[count].burst\_time);  printf("Process Priority:\t");  scanf("%d", &process[count].priority);  temp\_process = process[count].priority;  process[count].q = Queue(temp\_process);  process[count].ready = 0;  }  time = process[0].burst\_time;  for (y = 0; y < limit; y++)  {  for (count = y; count < limit; count++)  {  if (process[count].arrival\_time < time)  {  process[count].ready = 1;  }  }  for (count = y; count < limit - 1; count++)  {  for (j = count + 1; j < limit; j++)  {  if (process[count].ready == 1 && process[j].ready == 1)  {  if (process[count].q == 2 && process[j].q == 1)  {  temp = process[count];  process[count] = process[j];  process[j] = temp;  }  }  }  }  for (count = y; count < limit - 1; count++)  {  for (j = count + 1; j < limit; j++)  {  if (process[count].ready == 1 && process[j].ready == 1)  {  if (process[count].q == 1 && process[j].q == 1)  {  if (process[count].burst\_time >  process[j].burst\_time)  {  temp = process[count];  process[count] = process[j];  process[j] = temp;  }  else  {  break;  }  }  }  }  }  printf("\nProcess[%d]:\tTime:\t%d To %d\n", process[y].process\_id, time, time +  process[y].burst\_time);  time = time + process[y].burst\_time;  for (count = y; count < limit; count++)  {  if (process[count].ready == 1)  {  process[count].ready = 0;  }  }  }  return 0;  } |

**OUTPUT :**

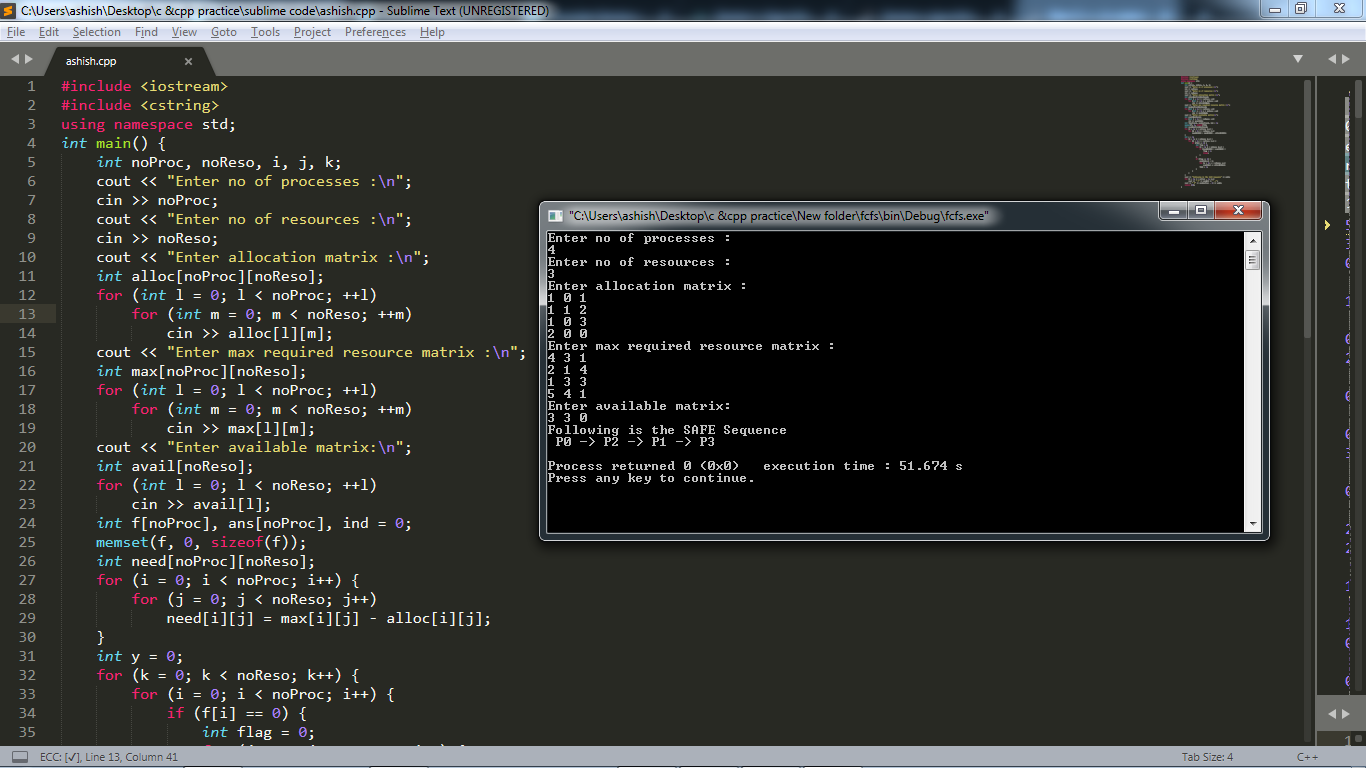
****

**PROGRAM 9 -  Program to stimulate deadlock detection**

**Code:**

|  |  |
| --- | --- |
| #include <iostream>  #include <cstring>  using namespace std;  int main() {  int noProc, noReso, i, j, k;  cout << "Enter no of processes :\n";  cin >> noProc;  cout << "Enter no of resources :\n";  cin >> noReso;  cout << "Enter allocation matrix :\n";  int alloc[noProc][noReso];  for (int l = 0; l < noProc; ++l)  for (int m = 0; m < noReso; ++m)  cin >> alloc[l][m];  cout << "Enter max required resource matrix :\n";  int max[noProc][noReso];  for (int l = 0; l < noProc; ++l)  for (int m = 0; m < noReso; ++m)  cin >> max[l][m];  cout << "Enter available matrix:\n";  int avail[noReso];  for (int l = 0; l < noReso; ++l)  cin >> avail[l];  int f[noProc], ans[noProc], ind = 0;  memset(f, 0, sizeof(f));  int need[noProc][noReso];  for (i = 0; i < noProc; i++) {  for (j = 0; j < noReso; j++)  need[i][j] = max[i][j] - alloc[i][j];  }  int y = 0;  for (k = 0; k < noReso; k++) {  for (i = 0; i < noProc; i++) {  if (f[i] == 0) {  int flag = 0;  for (j = 0; j < noReso; j++) {  if (need[i][j] > avail[j]) {  flag = 1;  break;  }  }  if (flag == 0) {  ans[ind++] = i;  for (y = 0; y < noReso; y++)  avail[y] += alloc[i][y];  f[i] = 1;  }  }  }  }  cout << "Following is the SAFE Sequence" << endl;  for (i = 0; i < noProc - 1; i++)  cout << " P" << ans[i] << " ->";  cout << " P" << ans[noProc - 1] << endl;  return (0);  } |  |

**Output:**

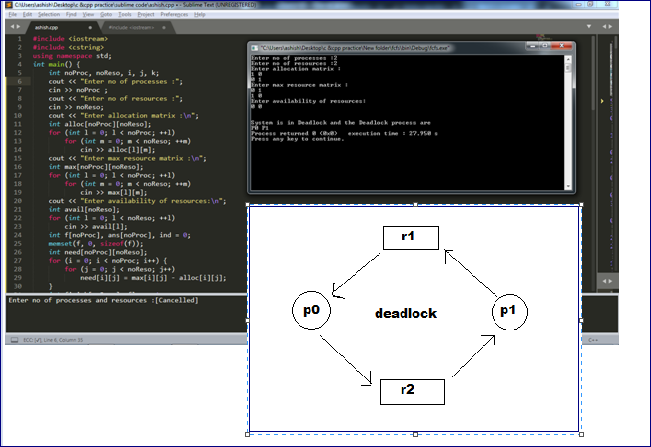
****

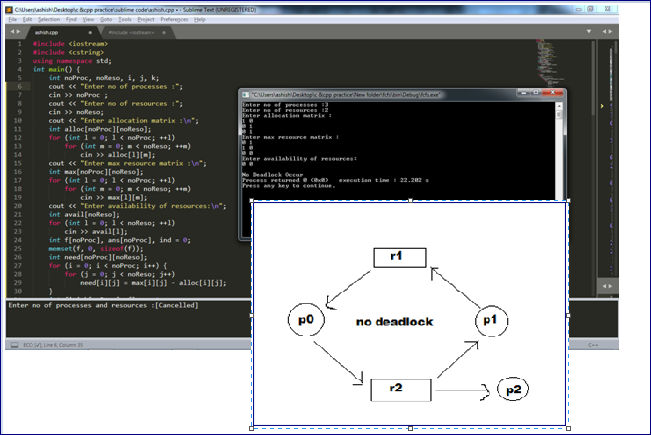
**PROGRAM 10 -  Program to stimulate deadlock avoidance**

**Code:**

|  |  |
| --- | --- |
| #include <iostream>  #include <cstring>  using namespace std;  int main() {  int noProc, noReso, i, j, k;  cout << "Enter no of processes :";  cin >> noProc ;  cout << "Enter no of resources :";  cin >> noReso;  cout << "Enter allocation matrix :\n";  int alloc[noProc][noReso];  for (int l = 0; l < noProc; ++l)  for (int m = 0; m < noReso; ++m)  cin >> alloc[l][m];  cout << "Enter max resource matrix :\n";  int max[noProc][noReso];  for (int l = 0; l < noProc; ++l)  for (int m = 0; m < noReso; ++m)  cin >> max[l][m];  cout << "Enter availability of resources:\n";  int avail[noReso];  for (int l = 0; l < noReso; ++l)  cin >> avail[l];  int f[noProc], ans[noProc], ind = 0;  memset(f, 0, sizeof(f));  int need[noProc][noReso];  for (i = 0; i < noProc; i++) {  for (j = 0; j < noReso; j++)  need[i][j] = max[i][j] - alloc[i][j];  }  int finish[noProc], flag = 1;  int deadlock[noProc];  memset(finish, 0, sizeof(finish));  //find need matrix  for (i = 0; i < noProc; i++) {  for (j = 0; j < noReso; j++) {  need[i][j] = max[i][j] - alloc[i][j];  }  }  while (flag) {  flag = 0;  for (i = 0; i < noProc; i++) {  int c = 0;  for (j = 0; j < noReso; j++)  if ((finish[i] == 0) && (need[i][j] <= avail[j])) {  c++;  if (c == noReso) {  for (k = 0; k < noReso; k++) {  avail[k] += alloc[i][j];  finish[i] = 1;  flag = 1;  }  if (finish[i] == 1)  i = noProc;  }  }  }  } | j = 0;  flag = 0;  for (i = 0; i < noProc; i++)  if (finish[i] == 0) {  deadlock[j++] = i;  flag = 1;  }  if (flag == 1) {  cout << "\n\nSystem is in Deadlock and the Deadlock process are\n";  for (i = 0; i < noProc; i++) {  cout << "P" << deadlock[i] << " ";  }  } else {  cout << "\nNo Deadlock Occur";  }  return (0);  } |

**Output:**

****

****

**PROGRAM 11 : best-fit contiguous memory allocation.**

**Code :**

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {    int fragment[20], b[20], p[20], i, j, nb, np, temp, lowest = 9999;  static int barray[20], parray[20];  cout << "\n\t\t\tMemory Management Scheme - Best Fit";  cout << "\nEnter the number of blocks:";  cin >> nb;  cout << "Enter the number of processes:";  cin >> np;  cout << "\nEnter the size of the blocks:-\n";  for (i = 1; i <= nb; i++)  {  cout << "Block no." << i << ":";  cin >> b[i];  }  cout << "\nEnter the size of the processes :-\n";  for (i = 1; i <= np; i++)  {  cout << "Process no. " << i << ":";  cin >> p[i];  }  for (i = 1; i <= np; i++)  {  for (j = 1; j <= nb; j++)  {  if (barray[j] != 1)  {  temp = b[j] - p[i];  if (temp >= 0)  if (lowest > temp)  {  parray[i] = j;  lowest = temp;  }  }  }  fragment[i] = lowest;  barray[parray[i]] = 1;  lowest = 10000;  }  cout << "\nProcess\_no\tProcess\_size\tBlock\_no\tBlock\_size\tFragment";  for (i = 1; i <= np && parray[i] != 0; i++)  cout << "\n" << i << "\t\t" << p[i] << "\t\t" << parray[i] << "\t\t" << b[parray[i]] << "\t\t" << fragment[i];  return 0;  } |

**OUTPUT:**

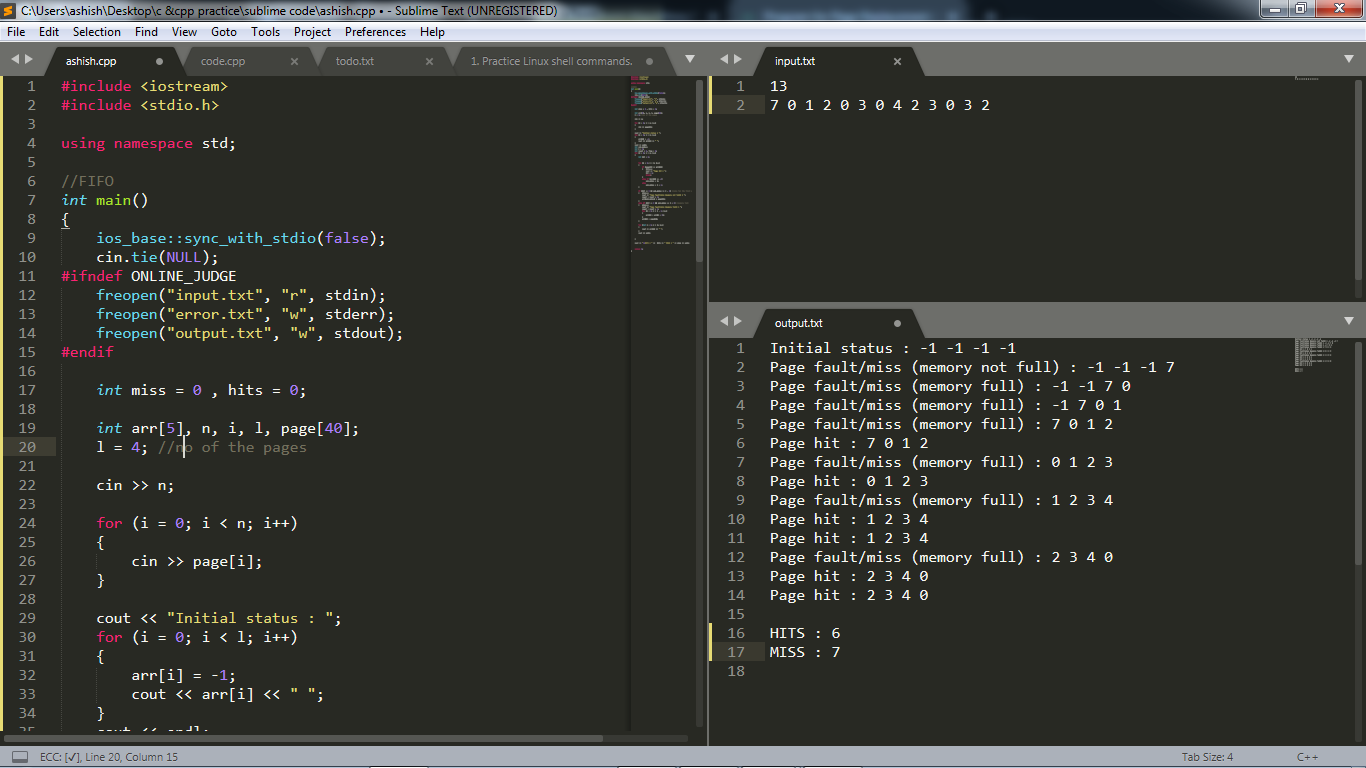


**PROGRAM 12 -  Program to stimulate FIFO page replacement algorithm**

**Code :**

|  |  |
| --- | --- |
| #include <iostream>  #include <stdio.h>  using namespace std;  //only give positive nos. as input  int main()  {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  int miss = 0 , hits = 0;  int arr[5], n, i, l, page[40];  l = 4; //no of the pages  cin >> n;  for (i = 0; i < n; i++)  {  cin >> page[i];  }  cout << "Initial status : ";  for (i = 0; i < l; i++)  {  arr[i] = -1;  cout << arr[i] << " ";  }  cout << endl;  int end\_index;  int j = 0;  int count = 0, flag = 0;  for (i = 0; i < n; i++)  {  int hit = 0;  for (j = 0; j < l; j++)  {  if (page[i] == arr[j])  { hits++;  cout << "Page hit : ";  hit = 1;  break;  }  else if (arr[j] == -1)  end\_index = j;  else  end\_index = l + 1;  }  if (hit == 0 && end\_index <= l - 1) //only for the first element  { miss++;  cout << "Page fault/miss (memory not full) : ";  count = count + 1;  arr[end\_index] = page[i];  }  else if (hit == 0 && end\_index == l + 1) //memory full  { miss++;  cout << "Page fault/miss (memory full) : ";  count = count + 1;  for (j = 0; j < l - 1; j++)  {  arr[j] = arr[j + 1];  }  arr[j] = page[i];  }  for (int k = 0; k < l; k++)  {  cout << arr[k] << " ";  }  cout << endl;  }  cout << "\nHITS : " << hits << " MISS : " << miss << endl;  return 0;  } |  |

**Output:**

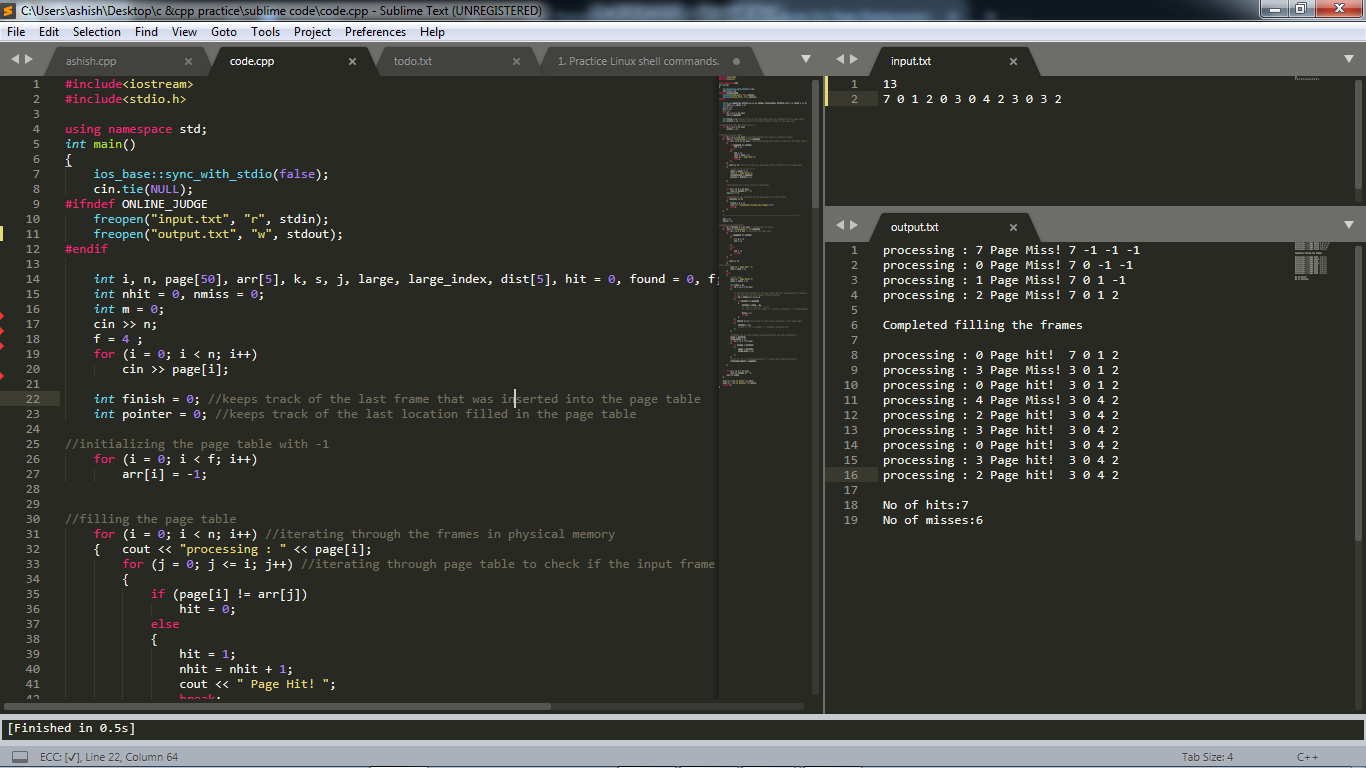


**PROGRAM 13 -  Program to stimulate LRU page replacement algorithm**

**Code:**

|  |  |
| --- | --- |
| #include<iostream>  #include<stdio.h>  using namespace std;  int main()  {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("output.txt", "w", stdout);  #endif  int i, n, page[50], arr[5], k, s, j, large, large\_index, dist[5], hit = 0, found = 0, f;  int nhit = 0, nmiss = 0;  int m = 0;  cin >> n;  f = 4 ;  for (i = 0; i < n; i++)  cin >> page[i];  int finish = 0; //keeps track of the last frame that was inserted into the page table  int pointer = 0; //keeps track of the last location filled in the page table  //initializing the page table with -1  for (i = 0; i < f; i++)  arr[i] = -1;  //filling the page table  for (i = 0; i < n; i++) //iterating through the frames in physical memory  { cout << "processing : " << page[i];  for (j = 0; j <= i; j++) //iterating through page table to check if the input frame is already present in the page table or not  {  if (page[i] != arr[j])  hit = 0;  else  {  hit = 1;  nhit = nhit + 1;  cout << " Page Hit! ";  break;  }  }  if (hit == 0) //if not presnt i.e. page miss ,then inserted into the page table  {  //cout << "\n j is:" << j;  nmiss = nmiss + 1;  cout << " Page Miss! ";  arr[pointer] = page[i];  pointer = pointer + 1;  }  //Displaying the current state of the array!  for (k = 0; k < f; k++)  cout << arr[k] << " ";  cout << "\n";  //checking if all locations of the page table have been filled  if (pointer == f)  {  finish = i + 1;  cout << "\nCompleted filling the frames\n\n";  break;  }  }  /\*-----------------------------------------------------------------------------\*/  hit = 0;  found = 0;  //optimal page replacement  for (i = finish; i < n; i++) // traversing the frames  { cout << "processing : " << page[i];  for (j = 0; j < f;) //traversing the page table  {  if (page[i] != arr[j])  {  j = j + 1;  hit = 0;  }  else  {  hit = 1;  break;  }  }  if (hit == 1)  {  //do nothing  cout << " Page hit! ";  nhit = nhit + 1;  }  else  {  //Page fault  cout << " Page Miss! ";  nmiss = nmiss + 1;  int index = i;  for (k = 0; k < f; k++)  {  // checking each element of the page table with the remaing array of frames, to check  // for the one with the longest forward distance  for (m = index; m > 0; m--)  {  if (arr[k] == page[m])  {  dist[k] = index - m;  // cout << "\n m is " << m;  // cout << "\n" << arr[k] << " alloted a distance " << int(dist[k]);  found = 1;  break;  }  }  if (found == 0) //not found in frame array ,assigned a very large value  {  dist[k] = 99;  //cout << "\n" << arr[k] << " alloted a distance 99";  }  }  //finding the one with largest backward distance and then replacing it  large = dist[0];  large\_index = 0;  for (s = 1; s < f; s++)  {  if (large < dist[s])  {  large = dist[s];  large\_index = s;  }  }  //cout << "\n" << arr[large\_index] << " stands with largest distance";  arr[large\_index] = page[i];  }  for (k = 0; k < f; k++)  cout << arr[k] << " ";  cout << endl;  }  cout << "\nNo of hits:" << nhit;  cout << "\nNo of misses:" << nmiss;  return 0;  } |  |

**Output:**

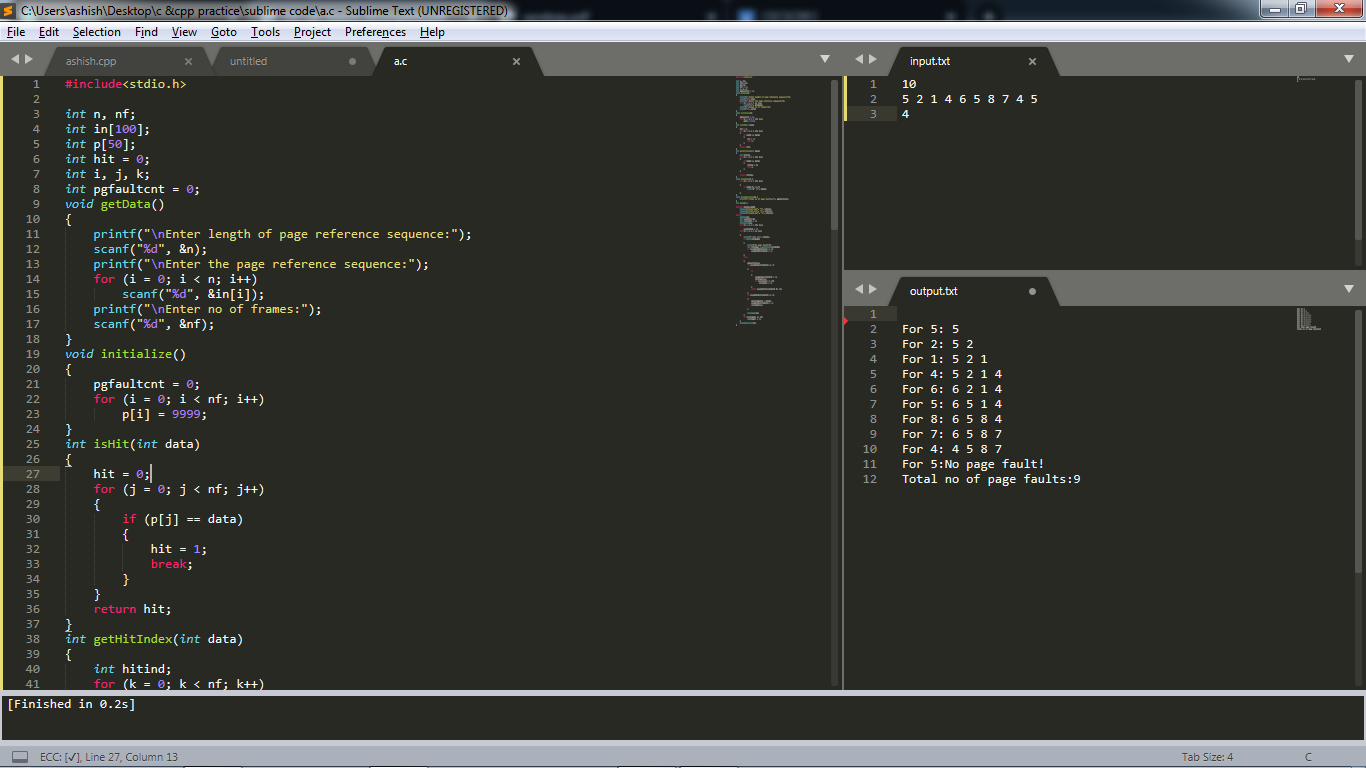


**PROGRAM 14 - Second Chance page replacement algorithm.**

**CODE :**

|  |
| --- |
| #include<stdio.h>  int n, nf;  int in[100];  int p[50];  int hit = 0;  int i, j, k;  int pgfaultcnt = 0;  void getData()  {  printf("\nEnter length of page reference sequence:");  scanf("%d", &n);  printf("\nEnter the page reference sequence:");  for (i = 0; i < n; i++)  scanf("%d", &in[i]);  printf("\nEnter no of frames:");  scanf("%d", &nf);  }  void initialize()  {  pgfaultcnt = 0;  for (i = 0; i < nf; i++)  p[i] = 9999;  }  int isHit(int data)  {  hit = 0;  for (j = 0; j < nf; j++)  {  if (p[j] == data)  {  hit = 1;  break;  }  }  return hit;  }  int getHitIndex(int data)  {  int hitind;  for (k = 0; k < nf; k++)  {  if (p[k] == data)  {  hitind = k;  break;  }  }  return hitind;  }  void dispPages() {  for (k = 0; k < nf; k++)  {  if (p[k] != 9999)  printf(" %d", p[k]);  }  }  void dispPgFaultCnt() {  printf("\nTotal no of page faults:%d", pgfaultcnt);  }  int main() {  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  getData();  int usedbit[50];  int victimptr = 0;  initialize();  for (i = 0; i < nf; i++)  usedbit[i] = 0;  for (i = 0; i < n; i++)  {  printf("\nFor %d:", in[i]);  if (isHit(in[i]))  {  printf("No page fault!");  int hitindex = getHitIndex(in[i]);  if (usedbit[hitindex] == 0)  usedbit[hitindex] = 1;  }  else  {  pgfaultcnt++;  if (usedbit[victimptr] == 1)  {  do  {  usedbit[victimptr] = 0;  victimptr++;  if (victimptr == nf)  victimptr = 0;  }  while (usedbit[victimptr] != 0);  }  if (usedbit[victimptr] == 0)  {  p[victimptr] = in[i];  usedbit[victimptr] = 1;  victimptr++;  }  dispPages();  }  if (victimptr == nf)  victimptr = 0;  }  dispPgFaultCnt();  } |

**OUTPUT :**

****

**PROGRAM : 15 Enhanced Second Chance page replacement algorithm.**

**CODE :**

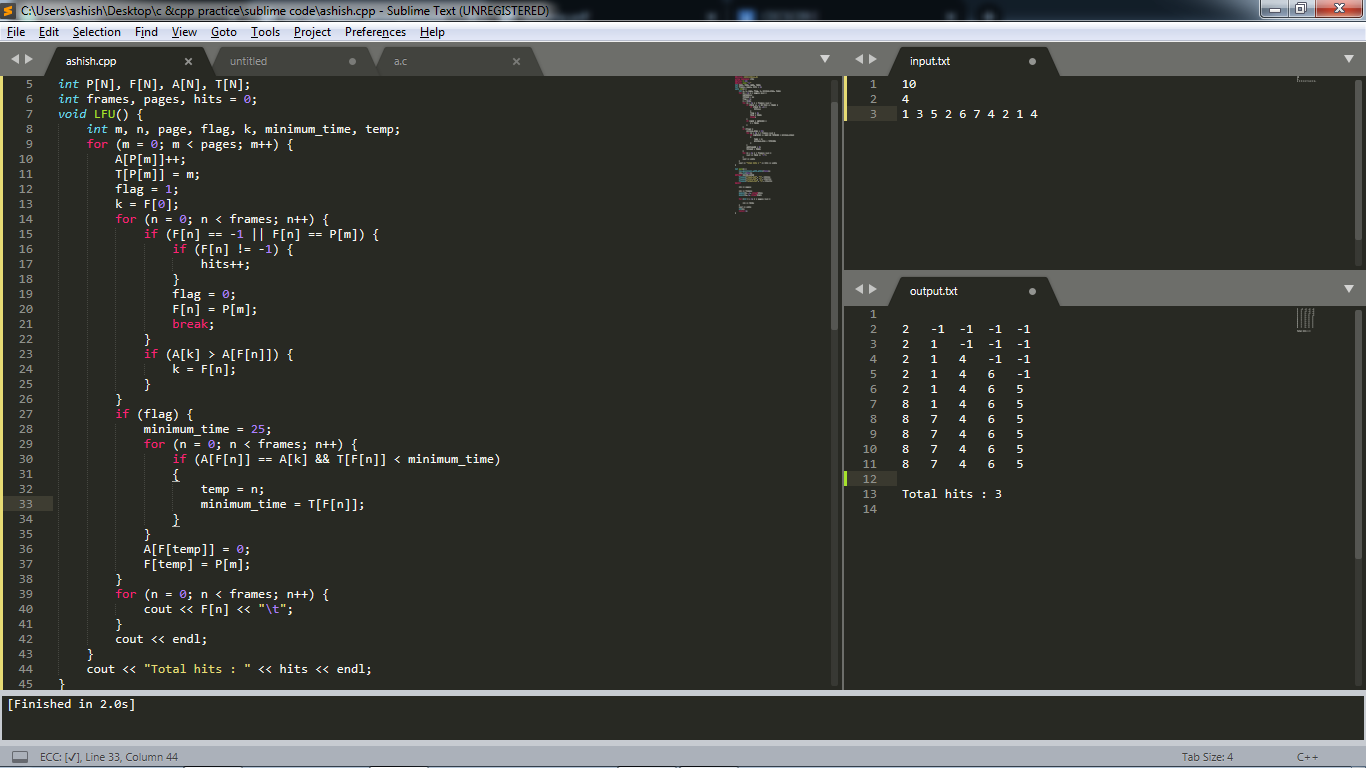
|  |
| --- |
| #include <iostream>  using namespace std;  void SecondChance\_page\_replacement(int n, int no\_of\_frames, int pages[], bool modified[])  {  int pointer = 0;  int page\_faults = 0;  int frames[no\_of\_frames];  bool ref[no\_of\_frames] = {0};  bool mod[no\_of\_frames] = {0};  for (int i = 0; i < no\_of\_frames; ++i)  frames[i] = -1;  for (int i = 0; i < n; i++)  {  int j;  for (j = 0; j < no\_of\_frames; j++)  {  if (frames[j] == -1)  {  page\_faults++;  frames[j] = pages[i];  mod[j] = modified[i];  cout << pages[i] << ": Page fault\t\t";  break;  }  else if (frames[j] == pages[i])  {  ref[j] = 1;  cout << pages[i] << ": Page hit\t\t";  break;  }  }  if (j == no\_of\_frames)  {  cout << "ENTERED FOR REPLACEMENT\n";  int k = 0;  while (true)  {  while (k < no\_of\_frames)  {  if (ref[pointer] == 0 && mod[pointer] == 0)  {  cout << "00 FOUND\n";  frames[pointer] = pages[i];  mod[pointer] = modified[i];  pointer = (pointer + 1) % no\_of\_frames;  cout << pages[i] << ": Page fault\t\t";  break;  }  pointer = (pointer + 1) % no\_of\_frames;  k++;  }  if (k == no\_of\_frames)  {  cout << "NO 00 FOUND\n";  while (k--)  {  if (ref[pointer] == 0 && mod[pointer] == 1)  {  cout << "01 FOUND\n";  frames[pointer] = pages[i];  mod[pointer] = modified[i];  pointer = (pointer + 1) % no\_of\_frames;  cout << pages[i] << ": Page fault\t\t";  page\_faults++;  break;  }  else  {  ref[pointer] = 0;  pointer = (pointer + 1) % no\_of\_frames;  }  }  if (k != 0)  break;  }  else  break;  }  }  }  } |

**PROGRAM : 16 : LFU page replacement algorithm.**

**CODE :**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  #define N 100  #define endl "\n"  int P[N], F[N], A[N], T[N];  int frames, pages, hits = 0;  void LFU() {  int m, n, page, flag, k, minimum\_time, temp;  for (m = 0; m < pages; m++) {  A[P[m]]++;  T[P[m]] = m;  flag = 1;  k = F[0];  for (n = 0; n < frames; n++) {  if (F[n] == -1 || F[n] == P[m]) {  if (F[n] != -1) {  hits++;  }  flag = 0;  F[n] = P[m];  break;  }  if (A[k] > A[F[n]]) {  k = F[n];  }  }  if (flag) {  minimum\_time = 25;  for (n = 0; n < frames; n++) {  if (A[F[n]] == A[k] && T[F[n]] < minimum\_time)  {  temp = n;  minimum\_time = T[F[n]];  }  }  A[F[temp]] = 0;  F[temp] = P[m];  }  for (n = 0; n < frames; n++) {  cout << F[n] << "\t";  }  cout << endl;  }  cout << "Total hits : " << hits << endl;  }  int main() {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  cin >> pages;  cin >> frames;  memset(F, -1, sizeof(F));  memset(A, 0, sizeof(A));  for (int i = 0; i < pages; i++) {  cin >> P[i];  }  cout << endl;  LFU();  return 0;  } |

**OUTPUT :**

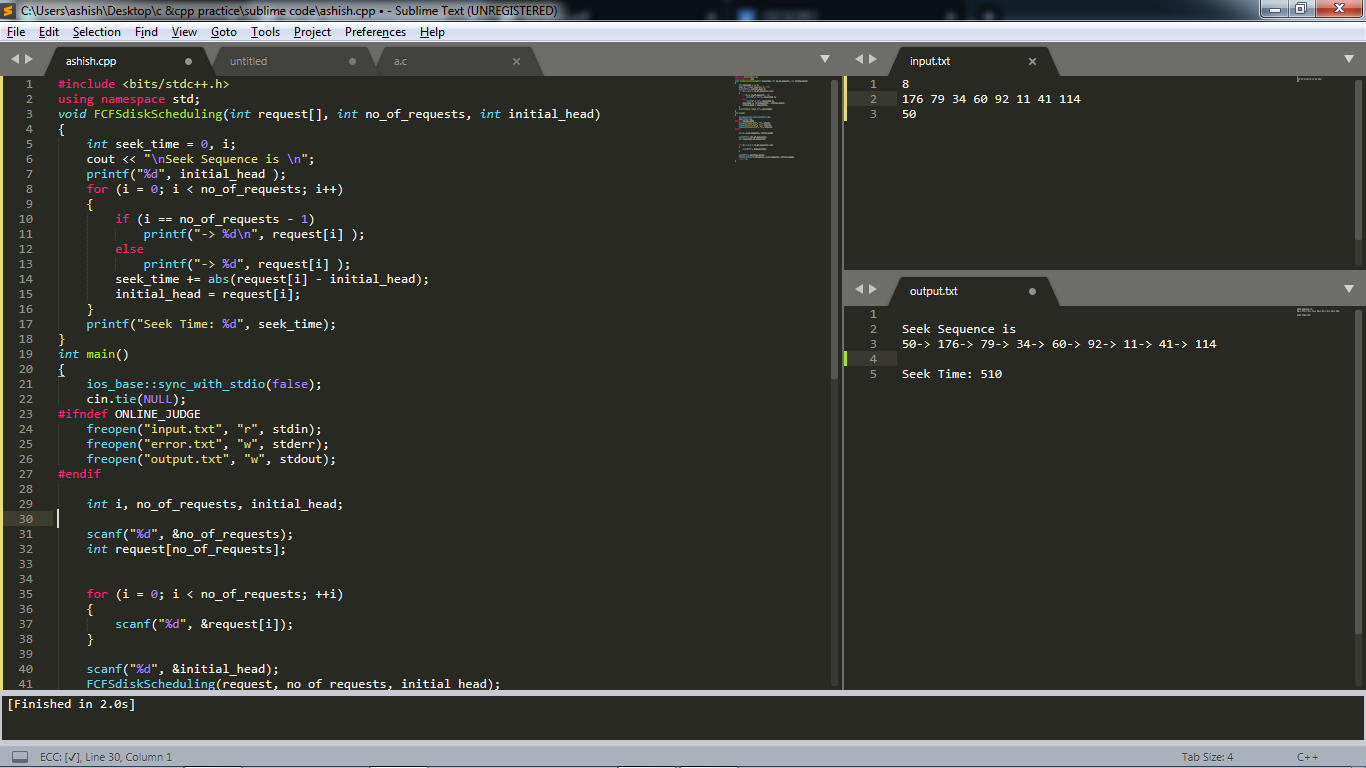
****

**PROGRAM 17 : FCFS disk scheduling algorithm.**

**CODE :**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  void FCFSdiskScheduling(int request[], int no\_of\_requests, int initial\_head)  {  int seek\_time = 0, i;  cout << "\nSeek Sequence is \n";  printf("%d", initial\_head );  for (i = 0; i < no\_of\_requests; i++)  {  if (i == no\_of\_requests - 1)  printf("-> %d\n", request[i] );  else  printf("-> %d", request[i] );  seek\_time += abs(request[i] - initial\_head);  initial\_head = request[i];  }  printf("Seek Time: %d", seek\_time);  }  int main()  {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  int i, no\_of\_requests, initial\_head;  scanf("%d", &no\_of\_requests);  int request[no\_of\_requests];  for (i = 0; i < no\_of\_requests; ++i)  {  scanf("%d", &request[i]);  }  scanf("%d", &initial\_head);  FCFSdiskScheduling(request, no\_of\_requests, initial\_head);  return 0;  } |

**OUTPUT :**

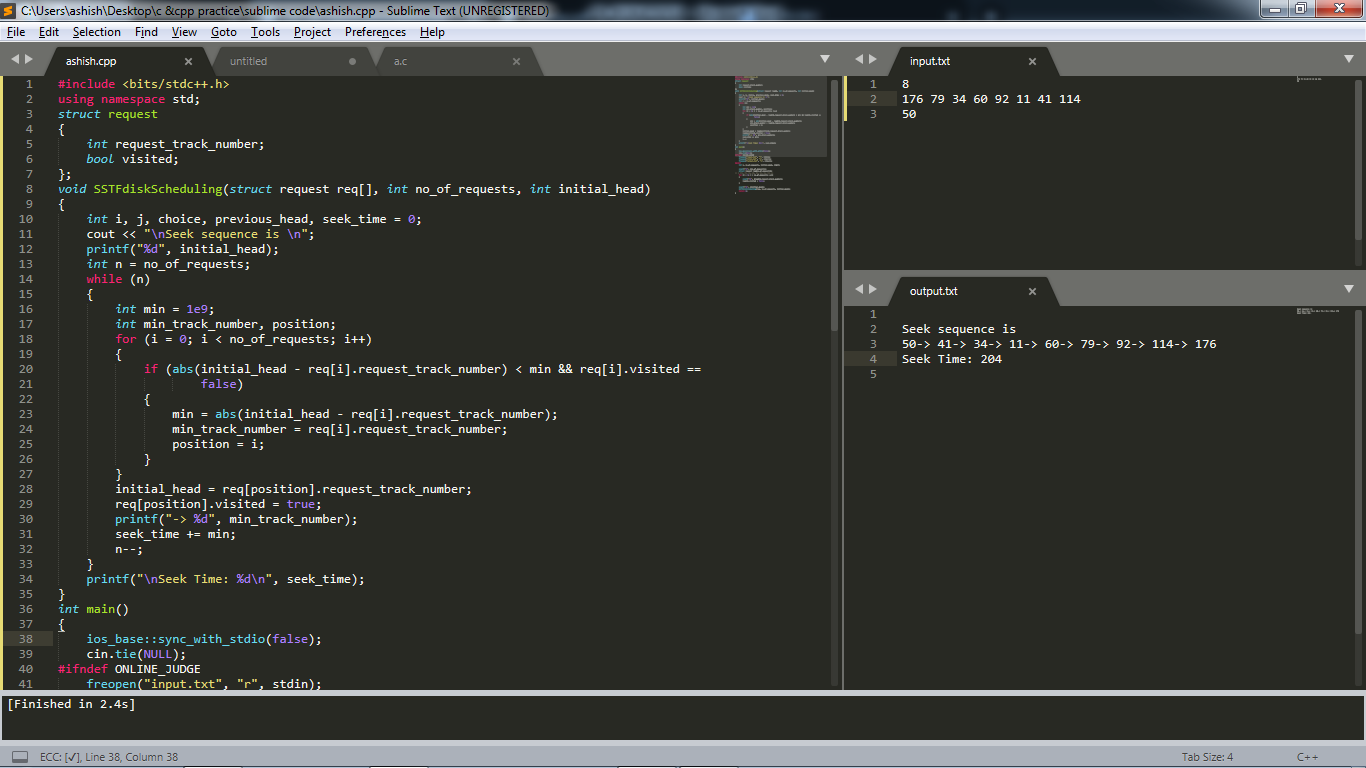
****

**PROGRAM 18 : SSTF disk scheduling algorithm.**

**CODE :**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  struct request  {  int request\_track\_number;  bool visited;  };  void SSTFdiskScheduling(struct request req[], int no\_of\_requests, int initial\_head)  {  int i, j, choice, previous\_head, seek\_time = 0;  cout << "\nSeek sequence is \n";  printf("%d", initial\_head);  int n = no\_of\_requests;  while (n)  {  int min = 1e9;  int min\_track\_number, position;  for (i = 0; i < no\_of\_requests; i++)  {  if (abs(initial\_head - req[i].request\_track\_number) < min && req[i].visited ==  false)  {  min = abs(initial\_head - req[i].request\_track\_number);  min\_track\_number = req[i].request\_track\_number;  position = i;  }  }  initial\_head = req[position].request\_track\_number;  req[position].visited = true;  printf("-> %d", min\_track\_number);  seek\_time += min;  n--;  }  printf("\nSeek Time: %d\n", seek\_time);  }  int main()  {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  int i, no\_of\_requests, initial\_head, limit;  scanf("%d", &no\_of\_requests);  struct request req[no\_of\_requests];  // { 176, 79, 34, 60, 92, 11, 41, 114 };  for (i = 0; i < no\_of\_requests; ++i)  {  scanf("%d", &req[i].request\_track\_number);  req[i].visited = false;  }  scanf("%d", &initial\_head);  SSTFdiskScheduling(req, no\_of\_requests, initial\_head);  return 0;  } |

**OUTPUT :**

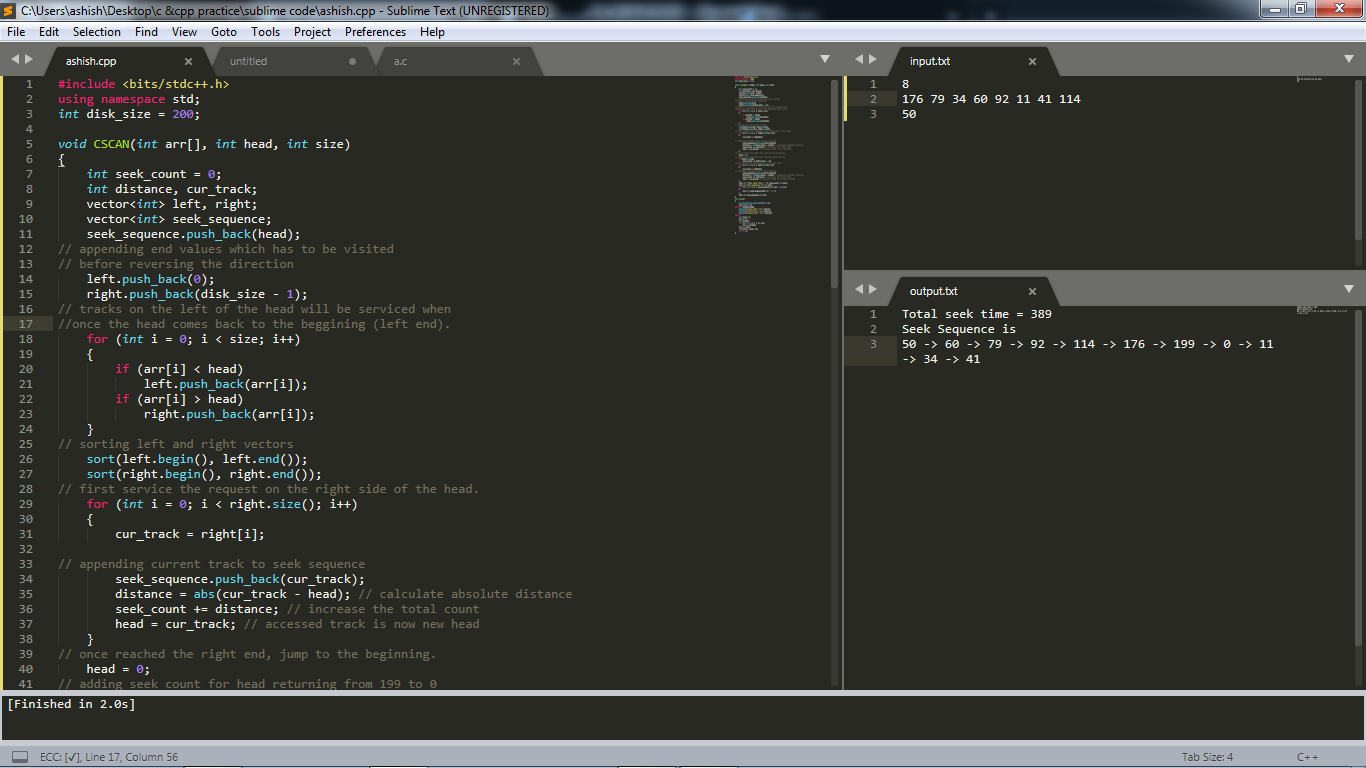
****

**PROGRAM 19 : C-SCAN disk scheduling algorithm.**

**CODE :**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int disk\_size = 200;  void CSCAN(int arr[], int head, int size)  {  int seek\_count = 0;  int distance, cur\_track;  vector<int> left, right;  vector<int> seek\_sequence;  seek\_sequence.push\_back(head);  // appending end values which has to be visited  // before reversing the direction  left.push\_back(0);  right.push\_back(disk\_size - 1);  // tracks on the left of the head will be serviced when  //once the head comes back to the beggining (left end).  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]);  }  // sorting left and right vectors  sort(left.begin(), left.end());  sort(right.begin(), right.end());  // first service the request on the right side of the head.  for (int i = 0; i < right.size(); i++)  {  cur\_track = right[i];  // appending current track to seek sequence  seek\_sequence.push\_back(cur\_track);  distance = abs(cur\_track - head); // calculate absolute distance  seek\_count += distance; // increase the total count  head = cur\_track; // accessed track is now new head  }  // once reached the right end, jump to the beginning.  head = 0;  // adding seek count for head returning from 199 to 0  if (left.size())  seek\_count += (disk\_size - 1);  // Now service the requests again which are left.  for (int i = 0; i < left.size(); i++)  {  cur\_track = left[i];  // appending current track to seek sequence  seek\_sequence.push\_back(cur\_track);  distance = abs(cur\_track - head); // calculate absolute distance  seek\_count += distance; // increase the total count  head = cur\_track; // accessed track is now the new head  }  cout << "Total seek time = " << seek\_count << endl;  cout << "Seek Sequence is" << endl;  for (int i = 0; i < seek\_sequence.size() - 1; i++)  {  cout << seek\_sequence[i] << " -> ";  }  cout << seek\_sequence.back();  }  int main()  {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  int head, n;  cin >> n;  int arr[n];  for (int i = 0; i < n; i++)  cin >> arr[i];  cin >> head;  CSCAN(arr, head, n);  return 0;  } |

**OUTPUT :**

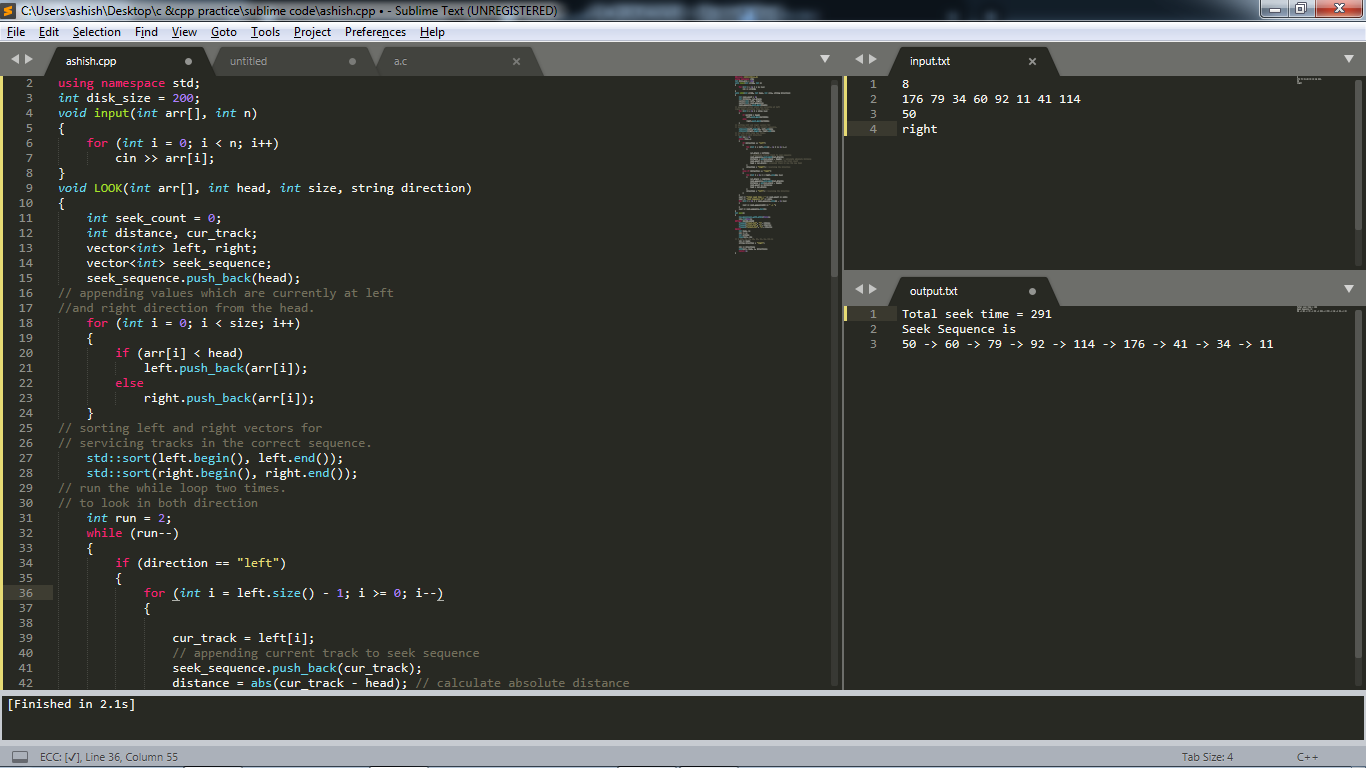
****

**PROGRAM : 20 LOOK disk scheduling algorithm.**

**CODE :**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int disk\_size = 200;  void input(int arr[], int n)  {  for (int i = 0; i < n; i++)  cin >> arr[i];  }  void LOOK(int arr[], int head, int size, string direction)  {  int seek\_count = 0;  int distance, cur\_track;  vector<int> left, right;  vector<int> seek\_sequence;  seek\_sequence.push\_back(head);  // appending values which are currently at left  //and right direction from the head.  for (int i = 0; i < size; i++)  {  if (arr[i] < head)  left.push\_back(arr[i]);  else  right.push\_back(arr[i]);  }  // sorting left and right vectors for  // servicing tracks in the correct sequence.  std::sort(left.begin(), left.end());  std::sort(right.begin(), right.end());  // run the while loop two times.  // to look in both direction  int run = 2;  while (run--)  {  if (direction == "left")  {  for (int i = left.size() - 1; i >= 0; i--)  {  cur\_track = left[i];  // appending current track to seek sequence  seek\_sequence.push\_back(cur\_track);  distance = abs(cur\_track - head); // calculate absolute distance  seek\_count += distance;// increase the total count  head = cur\_track;// accessed track is now the new head  }  direction = "right";// reversing the direction  }  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";// reversing the direction  }  }  cout << "Total seek time = " << seek\_count << endl;  cout << "Seek Sequence is" << endl;  for (int i = 0; i < seek\_sequence.size() - 1; i++)  {  cout << seek\_sequence[i] << " -> ";  }  cout << seek\_sequence.back();  }  int main()  {  ios\_base::sync\_with\_stdio(false);  cin.tie(NULL);  #ifndef ONLINE\_JUDGE  freopen("input.txt", "r", stdin);  freopen("error.txt", "w", stderr);  freopen("output.txt", "w", stdout);  #endif  int head, n;  cin >> n;  int arr[n];  input(arr, n);  // { 176, 79, 34, 60, 92, 11, 41, 114 };  cin >> head;  string direction = "right";  cin >> direction;  LOOK(arr, head, n, direction);  return 0;  } |

**OUTPUT :**

****