**NAME: ANKAN KUMAR MITRA**

**EXAM ROLL NO: T91/CSE/214032**

**REGISTRATION NO; D01-1111-0093-21**

**SEMESTER: 4**

**CLASS ROLL: 2**

**SUBJECT; OPERATING SYSTEMS PRACTICAL**

**PAPER CODE: CSCP-407**

**2: SJF Algorithm**

* **Problem Statement**

**Write a program to print a Gnatt chart for CPU scheduling using SJF algorithm.**

* + **Input example :**

**4 -> no. of processes**

**1 6 -> PID Burst\_time**

**2 8**

**3 7**

**4 3**

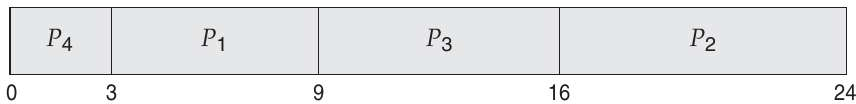
* + **Output example :**
  + **0 : Process 4 start**

**3 : Process 4 end Process 1 start**

**9 : Process 1 end Process 3 start**

**16 : Process 3 end Process 2 start**

**24 : Process 2 end**



* **Algorithm**

The SJF algorithm schedules the shortest job in the job queue to be executed first.This algorithm is non-preemptive so no job can preempt until the current executing job finishes.

The algorithm is like

At first a priority queue of the arrived jobs is developed with a minheap where priority is on the burst time i.e shorter burst time will have high priority.

Time counting is started from zero.

while priority queue is not empty{

Dequeue processes in ascending order of burst time.

Print process.id and the time when it finishes.

Time = Time+ process.burst\_time

}

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<math.h>

typedef struct node Queue;

struct node

{

int p\_id;

int burst\_time;

};

void swap(Queue \*x, Queue \*y)

{

Queue temp = \*x;

\*x = \*y;

\*y = temp;

}

void heapify(Queue \*arr,int n,int i)

{

int small = i;

int lchild = 2\*i + 1;

int rchild = 2\*i + 2;

if((lchild < n) && (arr[lchild].burst\_time < arr[small].burst\_time))

small = lchild;

if((rchild < n) && (arr[rchild].burst\_time < arr[small].burst\_time))

small = rchild;

if(small != i)

{

swap(&arr[i],&arr[small]);

heapify(arr,n,small);

}

}

void buildheap(Queue \*arr,int n)

{

int last\_internal\_node = (n/2) - 1;

int i;

for(i = last\_internal\_node ; i >= 0 ; i--)

heapify(arr,n,i);

}

Queue delheap(Queue \*queue,int n)

{

Queue element = queue[0];

swap(&queue[0],&queue[n-1]);

heapify(queue,n-1,0);

return element;

}

int main()

{

int num;

printf("Enter the number of process: ");

scanf("%d",&num);

Queue \*queue = (Queue \*)malloc(num\*sizeof(Queue));

Queue dequeued\_element;

Queue prev;

printf("Enter process no. and burst time:\n");

for(int i = 0 ; i < num ; i++)

{

scanf("%d %d",&queue[i].p\_id,&queue[i].burst\_time);

}

buildheap(queue,num);

dequeued\_element = delheap(queue,num);

int time = 0;

prev = dequeued\_element;

printf("%d : Process %d start\n",time,dequeued\_element.p\_id);

for(int i = num -1 ; i >= 0 ; i--)

{

time += prev.burst\_time;

if(i == 0)

printf("%d : Process %d end\n",time,prev.p\_id);

else

{

dequeued\_element = delheap(queue,i);

printf("%d : Process %d end Process %d start\n",time,prev.p\_id,dequeued\_element.p\_id);

}

prev = dequeued\_element;

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the number of process: 4

Enter process no. and burst time:

1 6

2 8

3 7

4 3

0 : Process 4 start

3 : Process 4 end Process 1 start

9 : Process 1 end Process 3 start

16 : Process 3 end Process 2 start

24 : Process 2 end

* **Discussion**

**The proposed algorithm has a runtime of O(nlogn) where n is the number of processes as the algorithm maintains a priority queue whose implementation takes nlogn time. Limitations : It is a non-preemptive algorithm and has biasness over shorter jobs,so we have to know the burst time of processes beforehand and it can be possible that shorter job has low priority and it causes high variance of waiting timees.For improving SJF algorithm we can implement shortest reamaining time algorithm (SRTF) which is the preemptive version of SJF otherwise we can apply HRN sheduling which considers waiting times also. Advantage: In this case throughput is more and has lower average waiting time than FIFO.**

**3: Priority Scheduling Algorithm**

* **Problem Statement**

**Write a program to print a Gnatt chart for CPU scheduling using priority algorithm.**

* + **Input example :**

**5 -> no. of processes**

**1 10 3 -> PID Burst\_time Priority**

**2 1 1**

**3 2 4**

**4 1 5**

**5 5 2**

* + **Output example : 0 : Process 2 start**

**1 : Process 2 end Process 5 start**

**6 : Process 5 end Process 1 start**

**16 : Process 1 end Process 3 start**

1. **: Process 3 end Process 4 start**
2. **: Process 4 end**

* **Algorithm**

The Priority scheduling algorithm schedules the highest priority job in the job queue to be executed first.This algorithm is non-preemptive so no job can preempt until the current executing job finishes.

The algorithm is like

At first a priority queue of the arrived jobs is developed with a minheap sorted in ascending order of priority.

Time counting is started from zero.

while priority queue is not empty{

Dequeue processes in ascending order of priority.

Print process.id and the time when it finishes.

Time = Time+ process.burst\_time

}

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<math.h>

typedef struct node Queue;

struct node

{

int p\_id;

int burst\_time;

int priority;

};

void swap(Queue \*x, Queue \*y)

{

Queue temp = \*x;

\*x = \*y;

\*y = temp;

}

void heapify(Queue \*arr,int n,int i)

{

int small = i;

int lchild = 2\*i + 1;

int rchild = 2\*i + 2;

if((lchild < n) && (arr[lchild].priority < arr[small].priority))

small = lchild;

if((rchild < n) && (arr[rchild].priority < arr[small].priority))

small = rchild;

if(small != i)

{

swap(&arr[i],&arr[small]);

heapify(arr,n,small);

}

}

void buildheap(Queue \*arr,int n)

{

int last\_internal\_node = (n/2) - 1;

int i;

for(i = last\_internal\_node ; i >= 0 ; i--)

heapify(arr,n,i);

}

Queue delheap(Queue \*queue,int n)

{

Queue element = queue[0];

swap(&queue[0],&queue[n-1]);

heapify(queue,n-1,0);

return element;

}

int main()

{

int num;

printf("Enter the number of process: ");

scanf("%d",&num);

Queue \*queue = (Queue \*)malloc(num\*sizeof(Queue));

Queue dequeued\_element;

Queue prev;

printf("Enter process number, burst time and its priority:\n");

for(int i = 0 ; i < num ; i++)

{

scanf("%d %d %d",&queue[i].p\_id,&queue[i].burst\_time,&queue[i].priority);

}

buildheap(queue,num);

dequeued\_element = delheap(queue,num);

int time = 0;

prev = dequeued\_element;

printf("%d : Process %d start\n",time,dequeued\_element.p\_id);

for(int i = num -1 ; i >= 0 ; i--)

{

time += prev.burst\_time;

if(i == 0)

printf("%d : Process %d end\n",time,prev.p\_id);

else

{

dequeued\_element = delheap(queue,i);

printf("%d : Process %d end Process %d start\n",time,prev.p\_id,dequeued\_element.p\_id);

}

prev = dequeued\_element;

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the number of process: 5

Enter process number, burst time and its priority:

1 10 3

2 1 1

3 2 4

4 1 5

5 5 2

0 : Process 2 start

1 : Process 2 end Process 5 start

6 : Process 5 end Process 1 start

16 : Process 1 end Process 3 start

18 : Process 3 end Process 4 start

19 : Process 4 end

* **Discussion**

**The proposed algorithm has a runtime of O(nlogn) where n is the number of processes as the algorithm maintains a priority queue whose implementation takes nlogn time. Limitations : It is a non-preemptive algorithm.It maybe possible that longer jobs have more priority thus it may cause large variance and high average waiting time in those cases.For improving priority scheduling algorithm we can implement preemptive scheduling with a time quantum or increase the priority of waiting jobs accordingly which is called aging. Advantage: If priority is defined correctly to reduce overhead and maximize throughput then this scheduling is helpful.**

**5: Round Robin Algorithm**

* **Problem Statement**

**Write a program to print a Gnatt chart for CPU scheduling using round robin algorithm.**

* + **Input example :**

1. **-> no. of processes**
2. **-> time quantum**

**1 9 -> PID Burst\_time**

**2 3**

**3 3**

* + **Output example : 0 : Process 1 start**

**4 : Process 1 preempt Process 2 start**

**7 : Process 2 end Process 3 start**

**10 : Process 3 end Process 1 start**

1. **: Process 1 preempt Process 1 start**
2. **: Process 1 end**

**Algorithm**

The Round robin algorithm schedules the jobs in the job queue in FIFO but it is a preemptive scheduling so if a job does not finish withen a fixed time quantum then it will be pushed in the back of the queue and the job in the front of the queue preempts that job.

The algorithm is like

At first a queue is developed to hold the jobs and the jobs are pushed in the queue as they arrived.

Time counting is started from zero.

while queue is not empty{

The current process is preempted by the next job in the queue if execution is not completed withen the time quantum.

Print process.id and the time when it is preempted.

Append the preempted process at the back of queue and subtract time quantum from its burst time

Print process.id and the time when it finishes.

Time = Time+ process.burst\_time

}

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<math.h>

#define SIZE 100

static int front = -1;

static int rear = -1;

typedef struct node Queue;

struct node

{

int p\_id;

int burst\_time;

};

void enqueue(Queue \*queue,int no, int time)

{

if(rear == SIZE - 1)

printf("Queue is full\n");

else

{

queue[++rear].p\_id = no;

queue[rear].burst\_time = time;

}

}

Queue dequeue(Queue \*queue)

{

if(front == rear)

{

Queue item;

item.p\_id = -1;

return item;

}

else

{

return queue[++front];

}

}

int main()

{

int num,quantum,no,time;

printf("Enter the number of process: ");

scanf("%d",&num);

printf("Enter the time quantum: ");

scanf("%d",&quantum);

Queue \*queue = (Queue\*)malloc(SIZE\*sizeof(Queue));

int\* remain = (int\*)malloc(num\*sizeof(int));

Queue dequeued\_element;

Queue prev;

printf("Enter process number and burst time\n");

for(int i = 0 ; i < num ; i++)

{

scanf("%d %d",&no,&time);

enqueue(queue,no,time);

remain[i] = time;

}

int count = 0;

dequeued\_element = dequeue(queue);

prev = dequeued\_element;

time = 0;

printf("%d : Process %d start\n",time,dequeued\_element.p\_id);

while(1)

{

if(front != rear)

dequeued\_element = dequeue(queue);

if(prev.burst\_time > quantum)

{

time += quantum;

prev.burst\_time = prev.burst\_time - quantum;

printf("%d : Process %d preempt Process %d start\n",time,prev.p\_id,dequeued\_element.p\_id);

enqueue(queue,prev.p\_id,prev.burst\_time);

}

else if(prev.burst\_time <= quantum && prev.p\_id != dequeued\_element.p\_id)

{

time += prev.burst\_time;

printf("%d : Process %d end Process %d start\n",time,prev.p\_id,dequeued\_element.p\_id);

}

else

{

time += prev.burst\_time;

printf("%d : Process %d end\n",time,prev.p\_id);

break;

}

if(prev.p\_id != dequeued\_element.p\_id)

{

prev = dequeued\_element;

}

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the number of process: 3

Enter the time quantum: 4

Enter process number and burst time

1 9

2 3

3 3

0 : Process 1 start

4 : Process 1 preempt Process 2 start

7 : Process 2 end Process 3 start

10 : Process 3 end Process 1 start

14 : Process 1 preempt Process 1 start

15 : Process 1 end

* **Discussion**

**The proposed algorithm has an overall linear time complexity which mostly depend on time quantum, if time quantum is very less then algorithm cause more overhead. Advantage: It is a preemptive algorithm and it improves the FIFO algorithm and it is beneficial in time sharing environment having less average waiting time than FIFO. Limitations: It requires the system to maintain several processes in memory to minimize overhead. This algorithm can be improved by increasing the priority of the waiting processes in the job queue(aging).The improved version is selfish round robin which maintains two queues active and holding queue and increases priority of waiting jobs.**

**8: Optimal(OPT/MIN) Page Replacement Algorithm**

* **Problem Statement**

**Write a program for Optimal (OPT / MIN) page replacement (i.e. replace the page that will not be used for the longest period of time)**

* + **Input example :**

**3 -> no. of page frames**

**20 -> no. of page references**

**7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 -> page references**

* + **Output example :**

**7 : Miss 7 -1 -1 -> Reference : Hit/Miss <page frames, if miss>**

**0 : Miss 7 0 -1**

1. **: Miss 7 0 1**
2. **: Miss 2 0 1**

**0 : Hit**

**3 : Miss 2 4 3**

**0 : Hit**

**4 : Miss 4 0 3**

1. **: Hit**
2. **: Hit**

**0 : Miss 2 0 3**

**3 : Hit**

**2 : Hit**

1. **: Miss 2 0 1**
2. **: Hit**
3. **: Hit**
4. **: Hit**

**7 : Miss 7 0 1**

1. **: Hit**
2. **: Hit**

**Algorithm**

The optimal page replacement algorithm follows the principle as

If referenced page is in the page frame then increase hitcount

Else if page fault occurs then find if any page in the page frame is not referenced in future,

if such a page exists then replace the page with the currently referenced page

else find the page which is referenced farthest in future and rreplace it with the currently referenced page.

* **C Code**

#include<stdio.h>

#include<stdlib.h>

int Find(int\* arr,int low,int high,int key)

{

int i;

for(i = low ; i <= high ; i++)

{

if(arr[i] == key)

return i;

}

return -1;

}

int rear = -1;

void display(int\* arr,int n)

{

int i;

for(i = 0 ; i < n ; i++)

{

printf("%d ",arr[i]);

}

printf("\n");

}

int main()

{

int n,ref,search = 0;

printf("Enter the number of page frames\n");

scanf("%d",&n);

int\* page = (int\*)malloc(n\*sizeof(int));

int i;

int\* time = (int\*)malloc(n\*sizeof(int));

for(i = 0; i < n; i++)

{

page[i] = -1;

time[i] = 1;

}

printf("Enter the number of page references\n");

scanf("%d",&ref);

int\* page\_ref = (int\*)malloc(ref\*sizeof(int));

printf("Enter the page references\n");

for(i = 0; i < ref; i++)

{

scanf("%d",&page\_ref[i]);

}

for(i = 0; i < ref; i++)

{

search = Find(page,0,n-1,page\_ref[i]);

if(search != -1)

{

printf("%d: Hit\n",page\_ref[i]);

}

else

{

if(rear != n-1)

{

page[++rear] = page\_ref[i];

}

else

{

int refer = 0,index = 0,far\_refer = 0;

far\_refer = Find(page\_ref,i+1,ref-1,page[0]);

if(far\_refer != -1)

{

for(int k = 0 ; k < n ; k++)

{

refer = Find(page\_ref,i+1,ref-1,page[k]);

if(refer == -1)

{

index = k;

break;

}

else

{

if(far\_refer < refer)

{

far\_refer = refer;

index = k;

}

}

}

}

page[index] = page\_ref[i];

}

printf("%d: Miss ",page\_ref[i]);

display(page,n);

}

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the number of page frames

3

Enter the number of page references

20

Enter the page references

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7: Miss 7 -1 -1

0: Miss 7 0 -1

1: Miss 7 0 1

2: Miss 2 0 1

0: Hit

3: Miss 2 0 3

0: Hit

4: Miss 2 4 3

2: Hit

3: Hit

0: Miss 2 0 3

3: Hit

2: Hit

1: Miss 2 0 1

2: Hit

0: Hit

1: Hit

7: Miss 7 0 1

0: Hit

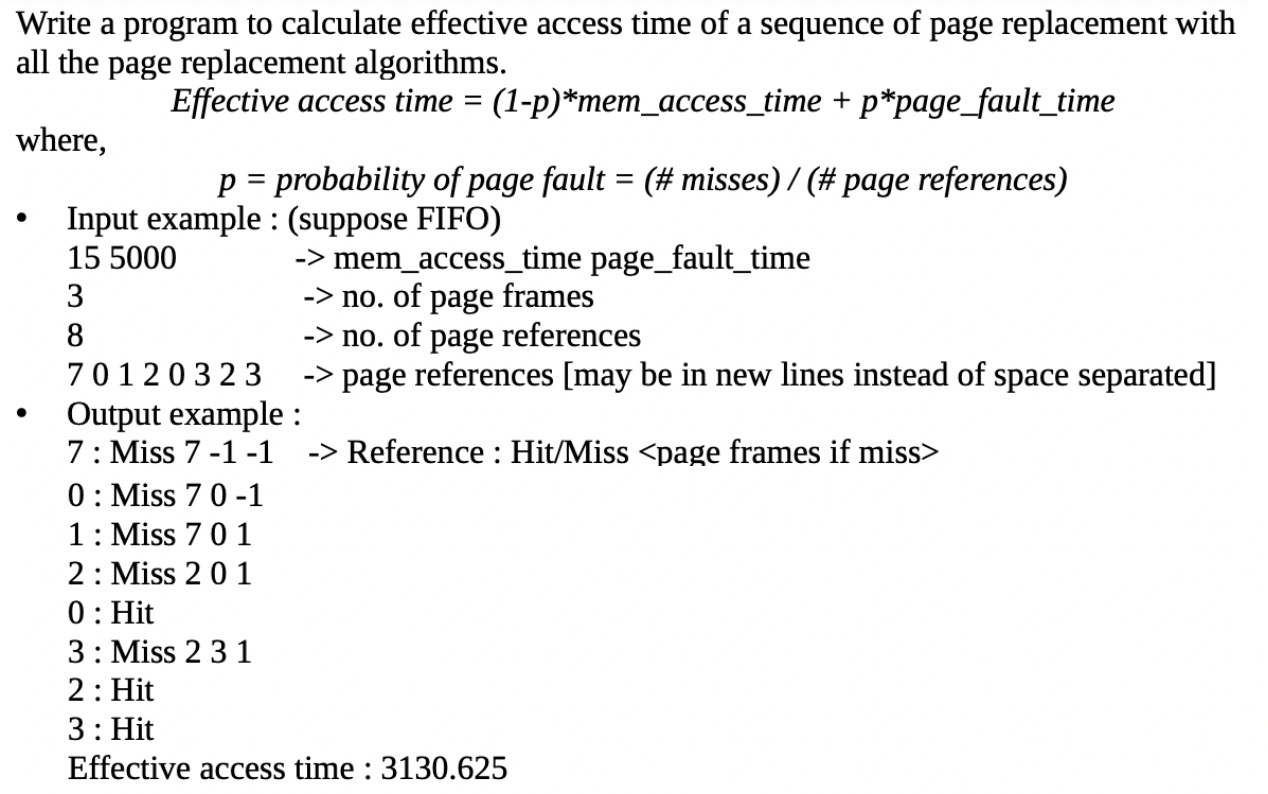
1: Hit

* **Discussion**

**The proposed algorithm has a runtime of O(n\*f) where n is the number of page references and f is the number of page frames. Limitations: Here we have to know the about sequeuence of page reference beforehand to determine which page to be replaced when page fault occurs.This algorithm can be optimized by implementing a hashmap of pages to minimize search time when page fault occurs but that incurs an extra space overhead. Advantage: In this case the hit ratio is more than fifo or lru page replacement as the pages which will not be referenced in future or or will be refereneced farthest in future are replaced with the incoming page.**

**9: Effective Memory Access Time for Page Replacement Algorithms**

* **Problem Statement**

****

**Algorithm**

This algorithm actually counts the number of hitcounts and misscounts of page references for different page replacement algorithms

For every page replacement algorithm,

If referenced page is in the page frame then increase hitcount

Else if page fault occurs then find the appropriate page in the page frame to be replaced with the currently referenced page and increase the misscount

Calculate p = probability of page fault = (misscount) / (number of page references) Calculate effectiveaccess time = (1-p)\*mem\_access\_time + p\*page\_fault\_time

* **C Code**

#include<stdio.h>

#include<stdlib.h>

int Find(int\* arr,int n,int key)

{

int i;

for(i = 0 ; i < n ; i++)

{

if(arr[i] == key)

return 1;

}

return 0;

}

int front = -1;/\* front,rear are set as -1 \*/

int rear = -1;

void QueueFull()

{

printf("Queue is Full\n");

}

void QueueEmpty()

{

printf("Queue is empty\n");

}

/\* function to enqueue \*/

void enQueue(int\* queue,int id,int num)

{

rear = (rear+1)%num;

queue[rear] = id;

}

void display(int\* arr,int n)

{

int i;

for(i = 0 ; i < n ; i++)

{

printf("%d ",arr[i]);

}

printf("\n");

}

int main()

{

int n,ref,misscount = 0;

int mem\_access,page\_fault;

printf("Enter memory access time and page fault time\n");

scanf("%d %d",&mem\_access,&page\_fault);

printf("Enter the number of page frames\n");

scanf("%d",&n);

int\* page = (int\*)malloc(n\*sizeof(int));

int i;

for(i = 0; i < n; i++)

{

page[i] = -1;

}

printf("Enter the number of page references\n");

scanf("%d",&ref);

int\* page\_ref = (int\*)malloc(ref\*sizeof(int));

printf("Enter the page references\n");

for(i = 0; i < ref; i++)

{

scanf("%d",&page\_ref[i]);

}

for(i = 0; i < ref; i++)

{

if(Find(page,n,page\_ref[i]))

{

printf("%d: Hit\n",page\_ref[i]);

}

else

{

printf("%d: Miss ",page\_ref[i]);

enQueue(page,page\_ref[i],n);

display(page,n);

misscount++;

}

}

double effect\_access = (double)((1-((double)misscount/ref))\*mem\_access + ((double)misscount/ref)\*page\_fault);

printf("Effective memory access time: %f",effect\_access);

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter memory access time and page fault time

15 5000

Enter the number of page frames

3

Enter the number of page references

8

Enter the page references

7 0 1 2 0 3 2 3

7: Miss 7 -1 -1

0: Miss 7 0 -1

1: Miss 7 0 1

2: Miss 2 0 1

0: Hit

3: Miss 2 3 1

2: Hit

3: Hit

Effective memory access time: 3130.625000

* **Discussion**

**The proposed algorithm has a runtime of O(n) for traversing every page reference and calculating hitcounts and misscounts where n is the number of page references but most of the page replacement algorithm takes O(n\*f) time where f is the number of page frames. The effective memory access time helps to compare different page replacement algorithms for different inputs of page refereneces.**

**11: SSTF Disk Scheduling Algorithm**

* **Problem Statement**

Write a program for Shortest Seek Time First (SSTF) disk scheduling algorithm

* + Input example :

53 -> head start position

8 -> no. of disk references

98 183 37 122 14 124 65 67 -> references [may be in new lines instead of space separated]

* + Output example :

65 : 12 moves -> [Explanation : 65 has shortest seek time from 53]

67 : 2 moves -> [Explanation : 67 has shortest seek time from 65]

... to continue for all references in order of their access

**Algorithm**

The Shortest seek time first(sstf) algorithm follows the principle as

Declare an array M to hold the disk references

H = initial head position

While( M is not fully traversed){

Calculate absolute difference (M[i]-H) for all i and find the minimum

Assign H to that M[i] for which the difference is minimum

Print the the minimum difference and current position after movement

}

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

int getMinSeekTime(int\* arr,int n,int pivot,int\* visited)

{

int min = INT\_MAX,index = 0;

int seektime;

for(int i = 0 ; i < n ; i++)

{

if(!visited[i])

{

seektime = abs(arr[i]-pivot);

if(min > seektime)

{

min = seektime;

index = i;

}

}

}

return index;

}

int main()

{

int head,ref,index = 0;

printf("Enter the head start position: ");

scanf("%d",&head);

printf("Enter the number of disk refernces: ");

scanf("%d",&ref);

int\* movement = (int\*)malloc(ref\*sizeof(int));

int\* visited = (int\*)calloc(ref,sizeof(int));

int seektime;

printf("Enter the disk refernces: ");

for(int i = 0 ; i < ref ; i++)

{

scanf("%d",&movement[i]);

}

int move = head;

for(int i = 0 ; i < ref ; i++)

{

index = getMinSeekTime(movement,ref,move,visited);

printf("%d: %d moves\n",movement[index],abs(movement[index]-move));

visited[index] = 1;

move = movement[index];

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the head start position: 53

Enter the number of disk refernces: 8

Enter the disk refernces: 98 183 37 122 14 124 65 67

65: 12 moves

67: 2 moves

37: 30 moves

14: 23 moves

98: 84 moves

122: 24 moves

124: 2 moves

183: 59 moves

* **Discussion**

**The proposed algorithm has a runtime of O(n^2) where n is the number of disk references. Limitations: Here is a possibility of indefinite postponment if references having less movement occurr and also high variance of response times. The algorithm can be improved if the search time is optimized,for that reason scan,look algorithms are implemented so that it is easier to move in a sorted array where search time is constant. Advantage: It has higher throughput and lower average response time than fifo.**

**12: SCAN Disk Scheduling Algorithm**

* **Problem Statement**

Write a program for SCAN disk scheduling algorithm

* + Input example :

0 199 -> lower\_limit upper\_limit

53 -> head start position

*l* -> initial head movement direction [left(*l)/*right(*r)*]

8 -> no. of disk references

98 183 37 122 14 124 65 67 -> references [may be in new lines instead of space separated]

* + Output example : o/p format same as other disk scheduling algorithms

**Algorithm**

The Scan algorithm follows the principle as

Declare an array M to hold the disk references

H = initial head position

D = initial direction for movement(left/right)

Partition M into two arrays L and R where L contains references less than or equal to H and R contains which are greater than H

Sort L and Sort R

If direction is left,then

for(traverese index i from reverse L){

Calculate absolute difference (M[i]-H) and print

Assign H to that M[i]

}

Move to the lower limit of or left end point of the references

Similarly traverse R but in forward direction

Else if direction is right,then just like left here traverse R in forward direction first and move to upper limit then traverse the left in reverse.

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

typedef struct node

{

int\* arr;

int lsize;

int rsize;

}Node;

void merging(int\* arr,int left,int mid,int right)

{

int i = left;

int j = mid + 1;

int k = 0;

int\* copy = (int\*)malloc((right - left + 1)\*sizeof(int));

while (( i <= mid ) && ( j <= right ))

{

if ( arr[i] < arr[j])

{

copy[k] = arr[i];

k++;

i++;

}

else

{

copy[k] = arr[j];

k++;

j++;

}

}

while ( i <= mid )

{

copy[k] = arr[i];

k++;

i++;

}

while ( j <= right )

{

copy[k] = arr[j];

k++;

j++;

}

for ( int i = 0 ; i <= (right - left) ; i++)

{

arr[left+i] = copy[i];

}

free(copy);/\* freeing the extra space \*/

}

/\* recursive merge sort function \*/

void merge\_sort(int\* arr,int left,int right)

{

int mid;

if ( left < right )

{

mid = (left + right)/2;

merge\_sort(arr,left,mid);

merge\_sort(arr,mid+1,right);

merging(arr,left,mid,right);

}

}

Node\* partition(int\* arr,int n,int pivot)

{

Node\* element = (Node\*)malloc(sizeof(Node));

element->arr = (int\*)malloc(n\*sizeof(int));

int j = 0,k = 0;

for(int i = 0 ; i < n ; i++)

{

if(arr[i] <= pivot )

{

element->arr[j++] = arr[i];

}

}

element->lsize = j;

element->rsize = n-j;

for(int i = 0 ; i < n ; i++)

{

if(arr[i] > pivot )

{

element->arr[j++] = arr[i];

}

}

merge\_sort(element->arr,0,element->lsize -1);

merge\_sort(element->arr,element->lsize,n-1);

return element;

}

void display(int\* arr,int n)

{

int i;

for(i = 0 ; i < n ; i++)

{

printf("%d ",arr[i]);

}

printf("\n");

}

int main()

{

int head,ref,index = 0;

char direction = 'l';

int lower\_lim,upper\_lim;

printf("Enter the lower and upper limits: ");

scanf("%d %d",&lower\_lim,&upper\_lim);

printf("Enter the head start position: ");

scanf("%d",&head);

fflush(stdin);

printf("Enter the direction(l for left/r for right): ");

scanf("%c",&direction);

printf("Enter the number of refernces: ");

scanf("%d",&ref);

printf("Enter the disk refernces: ");

int\* movement = (int\*)malloc(ref\*sizeof(int));

for(int i = 0 ; i < ref ; i++)

{

scanf("%d",&movement[i]);

}

Node\* partitioning = (Node\*)malloc(sizeof(Node));

partitioning = partition(movement,ref,head);

int\* arr = partitioning->arr;

int leftsize = partitioning->lsize;

int rightsize = partitioning->rsize;

int move = head;

if(direction == 'l')

{

for(int i = leftsize-1 ; i >= 0 ; i--)

{

printf("%d: %d moves\n",arr[i],(move-arr[i]));

move = arr[i];

}

if(arr[0] != lower\_lim)

{

printf("%d: %d moves\n",lower\_lim,move-lower\_lim);

}

for(int i = 0 ; i < rightsize ; i++)

{

printf("%d: %d moves\n",arr[i+leftsize],(arr[i+leftsize]-move));

move = arr[i+leftsize];

}

}

else

{

for(int i = 0 ; i < rightsize ; i++)

{

printf("%d: %d moves\n",arr[i+leftsize],(arr[i+leftsize]-move));

move = arr[i+leftsize];

}

if(arr[ref-1] != upper\_lim)

{

printf("%d: %d moves\n",upper\_lim,(upper\_lim-move));

}

for(int i = leftsize-1 ; i >= 0 ; i--)

{

printf("%d: %d moves\n",arr[i],(move-arr[i]));

move = arr[i];

}

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the lower and upper limits: 0 199

Enter the head start position: 53

Enter the direction(l for left/r for right): l

Enter the number of refernces: 8

Enter the disk refernces: 98 183 37 122 14 124 65 67

37: 16 moves

14: 23 moves

0: 14 moves

65: 51 moves

67: 2 moves

98: 31 moves

122: 24 moves

124: 2 moves

183: 59 moves

* **Discussion**

**The proposed algorithm has a runtime of O(nlogn) where n is the number of disk references. Advantage: It has higher throughput and low average response time.It has less variance of response times than sstf. Limitations: Here is a possibility of long waiting time for the reference requests just visited by the disk arm.It moves in fixed direction and changes direction once edge is reached,it may be possible that no reference near the edge,for that reason look algorithm is developed which moves withen the range of requests and C-scan be implemented to reduce variance of response time as it jumps to the opposite edge completing a sweep.**

**14: LOOK Disk Scheduling Algorithm**

* **Problem Statement**

Write a program for LOOK disk scheduling algorithm

* + Input example :

53 -> head start position

*l* -> initial head movement direction [left(*l)/*right(*r)*]

8 -> no. of disk references

98 183 37 122 14 124 65 67 -> references [may be in new lines instead of space separated]

* + Output example : o/p format same as other disk scheduling algorithms

**Algorithm**

The LOOK algorithm follows the principle of Scan algorithm but it traverses withen the range of the references.

Declare an array M to hold the disk references

H = initial head position

D = initial direction for movement(left/right)

Partition M into two arrays L and R where L contains references less than or equal to H and R contains which are greater than H

Sort L and Sort R

If direction is left,then

for(traverese index i from reverse L){

Calculate absolute difference (M[i]-H) and print

Assign H to that M[i]

}

Similarly traverse R but in forward direction

Else if direction is right,then just like left here traverse R in forward direction first and move to upper limit then traverse the left in reverse.

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

typedef struct node

{

int\* arr;

int lsize;

int rsize;

}Node;

void merging(int\* arr,int left,int mid,int right)

{

int i = left;

int j = mid + 1;

int k = 0;

int\* copy = (int\*)malloc((right - left + 1)\*sizeof(int));

while (( i <= mid ) && ( j <= right ))

{

if ( arr[i] < arr[j])

{

copy[k] = arr[i];

k++;

i++;

}

else

{

copy[k] = arr[j];

k++;

j++;

}

}

while ( i <= mid )

{

copy[k] = arr[i];

k++;

i++;

}

while ( j <= right )

{

copy[k] = arr[j];

k++;

j++;

}

for ( int i = 0 ; i <= (right - left) ; i++)

{

arr[left+i] = copy[i];

}

free(copy);/\* freeing the extra space \*/

}

/\* recursive merge sort function \*/

void merge\_sort(int\* arr,int left,int right)

{

int mid;

if ( left < right )

{

mid = (left + right)/2;

merge\_sort(arr,left,mid);

merge\_sort(arr,mid+1,right);

merging(arr,left,mid,right);

}

}

Node\* partition(int\* arr,int n,int pivot)

{

Node\* element = (Node\*)malloc(sizeof(Node));

element->arr = (int\*)malloc(n\*sizeof(int));

int j = 0,k = 0;

for(int i = 0 ; i < n ; i++)

{

if(arr[i] <= pivot )

{

element->arr[j++] = arr[i];

}

}

element->lsize = j;

element->rsize = n-j;

for(int i = 0 ; i < n ; i++)

{

if(arr[i] > pivot )

{

element->arr[j++] = arr[i];

}

}

merge\_sort(element->arr,0,element->lsize -1);

merge\_sort(element->arr,element->lsize,n-1);

return element;

}

void display(int\* arr,int n)

{

int i;

for(i = 0 ; i < n ; i++)

{

printf("%d ",arr[i]);

}

printf("\n");

}

int main()

{

int head,ref,index = 0;

char direction = 'l';

printf("Enter the head start position: ");

scanf("%d",&head);

fflush(stdin);

printf("Enter the direction(l for left/r for right): ");

scanf("%c",&direction);

printf("Enter the number of refernces: ");

scanf("%d",&ref);

printf("Enter the disk refernces: ");

int\* movement = (int\*)malloc(ref\*sizeof(int));

for(int i = 0 ; i < ref ; i++)

{

scanf("%d",&movement[i]);

}

Node\* partitioning = (Node\*)malloc(sizeof(Node));

partitioning = partition(movement,ref,head);

int\* arr = partitioning->arr;

int leftsize = partitioning->lsize;

int rightsize = partitioning->rsize;

int move = head;

if(direction == 'l')

{

for(int i = leftsize-1 ; i >= 0 ; i--)

{

printf("%d: %d moves\n",arr[i],(move-arr[i]));

move = arr[i];

}

for(int i = 0 ; i < rightsize ; i++)

{

printf("%d: %d moves\n",arr[i+leftsize],(arr[i+leftsize]-move));

move = arr[i+leftsize];

}

}

else

{

for(int i = 0 ; i < rightsize ; i++)

{

printf("%d: %d moves\n",arr[i+leftsize],(arr[i+leftsize]-move));

move = arr[i+leftsize];

}

for(int i = leftsize-1 ; i >= 0 ; i--)

{

printf("%d: %d moves\n",arr[i],(move-arr[i]));

move = arr[i];

}

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the head start position: 53

Enter the direction(l for left/r for right): l

Enter the number of refernces: 8

Enter the disk refernces: 98 183 37 122 14 124 65 67

37: 16 moves

14: 23 moves

65: 51 moves

67: 2 moves

98: 31 moves

122: 24 moves

124: 2 moves

183: 59 moves

* **Discussion**

**The proposed algorithm has a runtime of O(nlogn) where n is the number of disk references. Advantage: It has higher throughput and low average response time.It has less variance of response times than sstf and scan. It moves in fixed direction and changes direction once the last reference is reached. Limitations: Here is a possibility of long waiting time for the reference requests just visited by the disk arm. C-look be implemented to reduce variance of response time and uniform average waiting time as it jumps to the opposite limit of reference completing a sweep.**

**15: C-LOOK Disk Scheduling Algorithm**

* **Problem Statement**

Write a program for C-LOOK disk scheduling algorithm

* + Input example :

53 -> head start position

*l* -> initial head movement direction [left(*l)/*right(*r)*]

8 -> no. of disk references

98 183 37 122 14 124 65 67 -> references [may be in new lines instead of space separated]

* + Output example : o/p format same as other disk scheduling algorithms

**Algorithm**

The C-LOOK algorithm follows the principle of Scan algorithm but it traverses withen the range of the references.

Declare an array M to hold the disk references

H = initial head position

D = initial direction for movement(left/right)

Partition M into two arrays L and R where L contains references less than or equal to H and R contains which are greater than H

Sort L and Sort R

If direction is left,then

for(traverese index i from reverse L){

Calculate absolute difference (M[i]-H) and print

Assign H to that M[i]

}

Move to the last refernce in R

Similarly traverse R but in reverse direction

Else if direction is right,then just like left here traverse R in forward direction first and move to the lowest reference in L then traverse the L in forward.

* **C Code**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

typedef struct node

{

int\* arr;

int lsize;

int rsize;

}Node;

void merging(int\* arr,int left,int mid,int right)

{

int i = left;

int j = mid + 1;

int k = 0;

int\* copy = (int\*)malloc((right - left + 1)\*sizeof(int));

while (( i <= mid ) && ( j <= right ))

{

if ( arr[i] < arr[j])

{

copy[k] = arr[i];

k++;

i++;

}

else

{

copy[k] = arr[j];

k++;

j++;

}

}

while ( i <= mid )

{

copy[k] = arr[i];

k++;

i++;

}

while ( j <= right )

{

copy[k] = arr[j];

k++;

j++;

}

for ( int i = 0 ; i <= (right - left) ; i++)

{

arr[left+i] = copy[i];

}

free(copy);/\* freeing the extra space \*/

}

/\* recursive merge sort function \*/

void merge\_sort(int\* arr,int left,int right)

{

int mid;

if ( left < right )

{

mid = (left + right)/2;

merge\_sort(arr,left,mid);

merge\_sort(arr,mid+1,right);

merging(arr,left,mid,right);

}

}

Node\* partition(int\* arr,int n,int pivot)

{

Node\* element = (Node\*)malloc(sizeof(Node));

element->arr = (int\*)malloc(n\*sizeof(int));

int j = 0,k = 0;

for(int i = 0 ; i < n ; i++)

{

if(arr[i] <= pivot )

{

element->arr[j++] = arr[i];

}

}

element->lsize = j;

element->rsize = n-j;

for(int i = 0 ; i < n ; i++)

{

if(arr[i] > pivot )

{

element->arr[j++] = arr[i];

}

}

merge\_sort(element->arr,0,element->lsize -1);

merge\_sort(element->arr,element->lsize,n-1);

return element;

}

void display(int\* arr,int n)

{

int i;

for(i = 0 ; i < n ; i++)

{

printf("%d ",arr[i]);

}

printf("\n");

}

int main()

{

int head,ref,index = 0;

char direction = 'l';

printf("Enter the head start position: ");

scanf("%d",&head);

fflush(stdin);

printf("Enter the direction(l for left/r for right): ");

scanf("%c",&direction);

printf("Enter the number of refernces: ");

scanf("%d",&ref);

printf("Enter the disk refernces: ");

int\* movement = (int\*)malloc(ref\*sizeof(int));

for(int i = 0 ; i < ref ; i++)

{

scanf("%d",&movement[i]);

}

Node\* partitioning = (Node\*)malloc(sizeof(Node));

partitioning = partition(movement,ref,head);

int\* arr = partitioning->arr;

int leftsize = partitioning->lsize;

int rightsize = partitioning->rsize;

int move = head;

if(direction == 'l')

{

for(int i = leftsize-1 ; i >= 0 ; i--)

{

printf("%d: %d moves\n",arr[i],(move-arr[i]));

move = arr[i];

}

for(int i = rightsize-1 ; i >= 0 ; i--)

{

printf("%d: %d moves\n",arr[i+leftsize],abs(move-arr[i+leftsize]));

move = arr[i+leftsize];

}

}

else

{

for(int i = 0 ; i < rightsize ; i++)

{

printf("%d: %d moves\n",arr[i+leftsize],(arr[i+leftsize]-move));

move = arr[i+leftsize];

}

for(int i = 0 ; i < leftsize ; i++)

{

printf("%d: %d moves\n",arr[i],abs(arr[i]-move));

move = arr[i];

}

}

return 0;

}

**/\*------------------------------------------------------------------------------------------------------------------------- \*/**

**Input - Output example:**

Enter the head start position: 53

Enter the direction(l for left/r for right): l

Enter the number of refernces: 8

Enter the disk refernces: 98 183 37 122 14 124 65 67

37: 16 moves

14: 23 moves

183: 169 moves

124: 59 moves

122: 2 moves

98: 24 moves

67: 31 moves

65: 2 moves

* **Discussion**

**The proposed algorithm has a runtime of O(nlogn) where n is the number of disk references. Advantage: It has higher throughput and low average response time.It has less variance of response times than sstf and scan. It moves in fixed direction and jumps back to the opposite limit of reference causing uniform waiting time compared to Look. Limitations: For circularly traversing to the opposite reference limit througput decreases and average response time decrease though variance also decreases.**