BEST FIT

```
#include <stdio.h>
#define MAX_BLOCKS 10
int main() {
    int blockSize[MAX_BLOCKS];
    int processSize;
    int numOfBlocks;
    int i, j;
    int bestFitIdx = -1;
printf("Enter the number of memory blocks: ");
    scanf("%d", &numOfBlocks);
// Input memory block sizes
    printf("Enter the sizes of memory blocks:\n");
    for (i = 0; i < numOfBlocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }
printf("Enter the size of the process: ");
    scanf("%d", &processSize);
// Find best fit
    for (i = 0; i < numOfBlocks; i++) {
        if (blockSize[i] >= processSize) {
            if (bestFitIdx == -1 || blockSize[i] < blockSize[bestFitIdx]) {
                bestFitIdx = i;
            }
        }
}
if (bestFitIdx != -1) {
        printf("Process allocated to Block %d\n", bestFitIdx + 1);
    } else {
        printf("Process cannot be allocated\n");
    }
return 0;
}
```

FCFS NON PREEMPTIVE

```
#include <stdio.h>

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burstTime[n], waitingTime[n], turnaroundTime[n];
    printf("Enter burst times for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &burstTime[i]);
    }
    waitingTime[0] = 0;
    for (int i = 1; i < n; i++) {
        waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
    }
    for (int i = 0; i < n; i++) {
        turnaroundTime[i] = waitingTime[i] + burstTime[i];
    }
    float avgWaitingTime = 0, avgTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        avgWaitingTime += waitingTime[i];
        avgTurnaroundTime += turnaroundTime[i];
    }
    avgWaitingTime /= n;
    avgTurnaroundTime /= n;
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", i + 1, burstTime[i], waitingTime[i], turnaroundTime[i]);
    }
    printf("Average Waiting Time: %.2f\n", avgWaitingTime);
    printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime);
    return 0;
}
```

FIRST FIT

```
#include <stdio.h>
define MAX_BLOCKS 10

int main() {
    int blockSize[MAX_BLOCKS];
    int processSize;
    int numOfBlocks;
    int i, j;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &numOfBlocks);

    // Input memory block sizes
    printf("Enter the sizes of memory blocks:\n");
    for (i = 0; i < numOfBlocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

    printf("Enter the size of the process: ");
    scanf("%d", &processSize);

    // Find first fit
    for (i = 0; i < numOfBlocks; i++) {
        if (blockSize[i] >= processSize) {
            printf("Process allocated to Block %d\n", i + 1);
            break;
        }
    }

    if (i == numOfBlocks) {
        printf("Process cannot be allocated\n");
    }

    return 0;
}
```

WORST FIT

```
#include <stdio.h>
#define MAX_BLOCKS 10
int main() {
    int blockSize[MAX_BLOCKS];
    int processSize;
    int numOfBlocks;
    int i, j;
    int worstFitIdx = -1;
printf("Enter the number of memory blocks: ");
    scanf("%d", &numOfBlocks);
// Input memory block sizes
    printf("Enter the sizes of memory blocks:\n");
    for (i = 0; i < numOfBlocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }
printf("Enter the size of the process: ");
    scanf("%d", &processSize);
// Find worst fit
    for (i = 0; i < numOfBlocks; i++) {
        if (blockSize[i] >= processSize) {
            if (worstFitIdx == -1 || blockSize[i] > blockSize[worstFitIdx]) {
                worstFitIdx = i;
            }
        }
    }
if (worstFitIdx != -1) {
        printf("Process allocated to Block %d\n", worstFitIdx + 1);
    } else {
        printf("Process cannot be allocated\n");
    }

    return 0;
}
```

FIFO PAGE REPLACEMENT

```
#include <stdio.h>
#define MAX_FRAMES 3
#define MAX_PAGES 10
int main() {
    int referenceString[MAX_PAGES];
    int frames[MAX_FRAMES];
    int pageFaults = 0;
    int numOfPages;
    int i, j;
    int nextFrameIndex = 0;
    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &numOfPages);
    // Input reference string
    printf("Enter the reference string:\n");
    for (i = 0; i < numOfPages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &referenceString[i]);
    }
    // Initialize frames as empty (-1 indicates an empty frame)
    for (i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1;
    }
    // Perform page replacement
    for (i = 0; i < numOfPages; i++) {
        int currentPage = referenceString[i];
        int isPageFault = 1; // Flag to indicate if it's a page fault
  // Check if the current page is already in a frame
        for (j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == currentPage) {
                isPageFault = 0;
                break;
            }
        }
// If it's a page fault, replace the oldest page in the frame
        if (isPageFault) {
            frames[nextFrameIndex] = currentPage;
            nextFrameIndex = (nextFrameIndex + 1) % MAX_FRAMES;
            pageFaults++;
        }
// Print the current state of frames
        printf("Frames: ");
        for (j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == -1) {
                printf("- ");
            } else {
                printf("%d ", frames[j]);
            }
        }
        printf("\n");
    }
printf("Total page faults: %d\n", pageFaults);
return 0;
}
```

LRU

```
#include<stdio.h>
int findLRU(int time[], int n){
int i, minimum = time[0], pos
= 0;
for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}
int main()
{
    int no_of_frames,
no_of_pages, frames[10],
pages[30], counter = 0,
time[10], flag1, flag2, i, j, pos,
faults = 0;
printf("Enter number of
frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of
pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string:
");
    for(i = 0; i < no_of_pages;
++i){
     scanf("%d", &pages[i]);
    }
for(i = 0; i < no_of_frames;
++i){
     frames[i] = -1;
    }
    for(i = 0; i < no_of_pages;
++i){
     flag1 = flag2 = 0;
    for(j = 0; j < no_of_frames;
++j){
     if(frames[j] == pages[i]){
        counter++;
        time[j] = counter;
      flag1 = flag2 = 1;
      break;
      }
        }
if(flag1 == 0){
for(j = 0; j < no_of_frames;
++j){
        if(frames[j] == -1){
        counter++;
        faults++;
        frames[j] = pages[i];
        time[j] = counter;
        flag2 = 1;
        break;
        }
        }
        }
if(flag2 == 0){
        pos = findLRU(time,
no_of_frames);
        counter++;
        faults++;
        frames[pos] = pages[i];
        time[pos] = counter;
        }

        printf("\n");

        for(j = 0; j < no_of_frames;
++j){
       if(frames[j]==-1){
          frames[j]=0;
          }
        printf("%d\t", frames[j]);
        }
}
printf("\n\nTotal Page Faults
= %d", faults);
}
```

OPTIMAL

```
#include <stdio.h>

#define MAX_FRAMES 10
#define MAX_PAGES 100

int main() {
   int referenceString[MAX_PAGES];
   int frames[MAX_FRAMES];
   int pageFaults = 0;
   int numOfPages, numOfFrames;
   int i, j, k;
   int nextFrameIndex = 0;

   printf("Enter the number of pages in
the reference string: ");
   scanf("%d", &numOfPages);

   // Input reference string
   printf("Enter the reference
string:\n");
   for (i = 0; i < numOfPages; i++) {
      printf("Page %d: ", i + 1);
      scanf("%d", &referenceString[i]);
   }

   printf("Enter the number of frames:
");
   scanf("%d", &numOfFrames);

   // Initialize frames as empty (-1
indicates an empty frame)
   for (i = 0; i < numOfFrames; i++) {
      frames[i] = -1;
   }

   // Perform page replacement
   for (i = 0; i < numOfPages; i++) {
      int currentPage =
referenceString[i];
      int isPageFault = 1; // Flag to
indicate if it's a page fault

      // Check if the current page is
already in a frame
      for (j = 0; j < numOfFrames; j++) {
         if (frames[j] == currentPage) {
            isPageFault = 0;
            break;
         }
      }

      // If it's a page fault, replace the
page with the maximum distance in
future reference string
      if (isPageFault) {
         int maxDistance = 0;
         int replaceIndex = 0;

         for (j = 0; j < numOfFrames; j++) {
            if (frames[j] == -1) {
               replaceIndex = j;
               break;
            }

            int found = 0;
            for (k = i + 1; k < numOfPages;
k++) {
               if (frames[j] ==
referenceString[k]) {
                  found = 1;
                  if (k > maxDistance) {
                     maxDistance = k;
                     replaceIndex = j;
                  }
                  break;
               }
            }

            if (!found) {
               replaceIndex = j;
               break;
            }
         }

         frames[replaceIndex] =
currentPage;
         pageFaults++;
      }

      // Print the current state of frames
      printf("Frames: ");
      for (j = 0; j < numOfFrames; j++) {
         if (frames[j] == -1) {
            printf("- ");
         } else {
            printf("%d ", frames[j]);
         }
      }
      printf("\n");
   }
   printf("Total page faults: %d\n",
pageFaults);

   return 0;
}
```

DINNING

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
sem_t room;
sem_t chopstick[5];
void * philosopher(void *);
void eat(int);
int main()
{
        int i,a[5];
        pthread_t tid[5];
        sem_init(&room,0,4);
        for(i=0;i<5;i++)
        sem_init(&chopstick[i],0,1);
        for(i=0;i<5;i++)
        {
                a[i]=i;

        pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
        }
        for(i=0;i<5;i++)
        pthread_join(tid[i],NULL);
}
void * philosopher(void * num)
{
        int phil=*(int *)num;
        sem_wait(&room);
        printf("\n philosopher %d has
entered room",phil);
        sem_wait(&chopstick[phil]);
        sem_wait(&chopstick[(phil+1)%
5]);
        eat(phil);
        sleep(2);
        printf("\n philosopher %d has
finished eating",phil);
        sem_post(&chopstick[(phil+1)%
5]);
        sem_post(&chopstick[phil]);
        sem_post(&room);
}
void eat(int phil)
{
        printf("\n philosopher %d is
eating",phil);
}
```

SSTF

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int
i,n,k,req[50],mov=0,cp,index[50],min,
a[50],j=0,mini,cp1;
   printf("enter the current
position\n");
   scanf("%d",&cp);
   printf("enter the number of
requests\n");
   scanf("%d",&n);
   cp1=cp;
   printf("enter the request order\n");
   for(i=0;i<n;i++)
   {
      scanf("%d",&req[i]);
   }
   for(k=0;k<n;k++)
   {
   for(i=0;i<n;i++)
   {
      index[i]=abs(cp-req[i]);
   }
   min=index[0];
   mini=0;
   for(i=1;i<n;i++)
   {
      if(min>index[i])
      {
         min=index[i];
         mini=i;
      }
   }
   a[j]=req[mini];
   j++;
   cp=req[mini];
   req[mini]=999;
   }
   printf("Sequence is : ");
   printf("%d",cp1);
   mov=mov+abs(cp1-a[0]);
   printf(" -> %d",a[0]);
   for(i=1;i<n;i++)
   {
      mov=mov+abs(a[i]-a[i-1]);
      printf(" -> %d",a[i]);
   }
   printf("\n");
   printf("total head movement =
%d\n",mov); }
```

PRODUCER CONSUMER

```
#include<stdio.h>
#include<stdlib.h>

int
mutex=1,full=0,empty=3,x=0;

int main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.Producer\n2.Consumer\n3.Exit");
while(1)

{
printf("\nEnter your choice:");
scanf("%d",&n);
switch(n)
{
case 1:
if((mutex==1)&&(empty!=0))
producer();
else
printf("Buffer is full!!");
break;
case 2:
if((mutex==1)&&(full!=0))
consumer();
else
printf("Buffer is empty!!");
break;
case 3:
exit(0);
break;
}
}
return 0;
}

int wait(int s)
{
return (--s);
}

int signal(int s)
{
return(++s);
}

void producer()
{
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\nProducer produces
the item %d",x);
mutex=signal(mutex);
}

void consumer()
{
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\nConsumer
consumes item %d",x);
x--;
mutex=signal(mutex);
}
```

Look disc shed

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int
RQ[100],i,j,n,TotalHeadMoment=
0,initial,size,move;
   printf("Enter the number of
Requests\n");
   scanf("%d",&n);
   printf("Enter the Requests
sequence\n");
   for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
   printf("Enter initial head
position\n");
   scanf("%d",&initial);
   printf("Enter total disk size\n");
   scanf("%d",&size);
   printf("Enter the head
movement direction for high 1
and for low 0\n");
   scanf("%d",&move);
   for(i=0;i<n;i++)
   {
      for(j=0;j<n-i-1;j++)
      {
         if(RQ[j]>RQ[j+1])
         {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
         }

      }
   }

   int index;
   for(i=0;i<n;i++)
   {
      if(initial<RQ[i])
      {
         index=i;
         break;
      }
   }
   if(move==1)
   {
      for(i=index;i<n;i++)
      {
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];
      }

      for(i=index-1;i>=0;i--)
      {
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
          initial=RQ[i];

      }
   }
   else
   {
      for(i=index-1;i>=0;i--)
      {
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];
      }

      for(i=index;i<n;i++)
      {
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];

      }
   }

   printf("Total head movement is
%d",TotalHeadMoment);
   return 0;
}
```

SCAN

```
#include<stdio.h>
int absoluteValue(int);
int main()
{
   int
queue[25],n,headposition,i,j,k,seek=0, maxrange,

difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;
   float averageSeekTime;
   printf("Enter the maximum
range of Disk: ");
   scanf("%d",&maxrange);
   printf("Enter the number of
queue requests: ");
   scanf("%d",&n);
   printf("Enter the initial head
position: ");
   scanf("%d",&headposition);
   printf("Enter the disk positions
to be read(queue): ");
   for(i=1;i<=n;i++)   {
      scanf("%d",&temp);
      if(temp>headposition){
         queue1[temp1]=temp;
         temp1++;
      }
      else   {
         queue2[temp2]=temp;
         temp2++;
      }
   }
   for(i=0;i<temp1-1;i++){
      for(j=i+1;j<temp1;j++){
         if(queue1[i]>queue1[j]){
            temp=queue1[i];
            queue1[i]=queue1[j];
            queue1[j]=temp;
         }
      }
   }
   for(i=0;i<temp2-1;i++){
      for(j=i+1;j<temp2;j++){
         if(queue2[i]<queue2[j]){
            temp=queue2[i];
            queue2[i]=queue2[j];
            queue2[j]=temp;
         }
      }
   }
   for(i=1,j=0;j<temp1;i++,j++){
      queue[i]=queue1[j];
   }
   queue[i]=maxrange;

for(i=temp1+2,j=0;j<temp2;i++,j++){
      queue[i]=queue2[j];
   }
   queue[i]=0;
   queue[0]=headposition;

   for(j=0; j<=n; j++) {
      difference =
absoluteValue(queue[j+1]-
queue[j]);
      seek = seek + difference;
      printf("Disk head moves
from position %d to %d with
Seek %d \n", queue[j],
queue[j+1], difference);
   }
   printf("Total Head Movement
= %d\n", seek);
}
int absoluteValue(int x){
   if(x>0){
      return x;
   }
   else{
      return x*-1;
      }
}
```

RR

```
#include <stdio.h>

#define MAX_PROCESSES 10

typedef struct {
   int processID;
   int burstTime;
   int remainingTime;
   int arrivalTime;
   int turnaroundTime;
   int waitingTime;
} Process;

void roundRobin(Process processes[],
int numOfProcesses, int timeQuantum)
{
   int completedProcesses = 0;
   int currentTime = 0;
   int i;

   while (completedProcesses <
numOfProcesses) {
      for (i = 0; i < numOfProcesses; i++) {
         if (processes[i].remainingTime >
0) {
            if (processes[i].remainingTime
<= timeQuantum) {
               currentTime +=
processes[i].remainingTime;
               processes[i].remainingTime
= 0;
               completedProcesses++;
               processes[i].turnaroundTime
= currentTime -
processes[i].arrivalTime;
               processes[i].waitingTime =
processes[i].turnaroundTime -
processes[i].burstTime;
            } else {
               currentTime +=
timeQuantum;
               processes[i].remainingTime -
= timeQuantum;
            }
         }
      }
   }
}

int main() {
   Process processes[MAX_PROCESSES];
   int numOfProcesses, timeQuantum;
   int i;
   float avgTurnaroundTime = 0,
avgWaitingTime = 0;

   printf("Enter the number of
processes: ");
   scanf("%d", &numOfProcesses);

   printf("Enter the burst time and
arrival time for each process:\n");
   for (i = 0; i < numOfProcesses; i++) {
      printf("Process %d\n", i + 1);
      printf("Burst time: ");
      scanf("%d",
&processes[i].burstTime);
      printf("Arrival time: ");
      scanf("%d",
&processes[i].arrivalTime);
      processes[i].processID = i + 1;
      processes[i].remainingTime =
processes[i].burstTime;
   }

   printf("Enter the time quantum: ");
   scanf("%d", &timeQuantum);

   roundRobin(processes,
numOfProcesses, timeQuantum);

   printf("\nProcess\tBurst
Time\tArrival Time\tTurnaround
Time\tWaiting Time\n");
   for (i = 0; i < numOfProcesses; i++) {

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
processes[i].processID,
processes[i].burstTime,
processes[i].arrivalTime,
         processes[i].turnaroundTime,
processes[i].waitingTime);
      avgTurnaroundTime +=
processes[i].turnaroundTime;
      avgWaitingTime +=
processes[i].waitingTime;
   }

   avgTurnaroundTime /=
numOfProcesses;
   avgWaitingTime /= numOfProcesses;

   printf("\nAverage Turnaround Time:
%.2f\n", avgTurnaroundTime);
   printf("Average Waiting Time:
%.2f\n", avgWaitingTime);

   return 0;
}
```