1.BANKERS ALGORITHM:

```
#include <stdio.h>


int main() {

   int n, m, i, j, k;


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

scanf("%d", &n);


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

scanf("%d", &m);


   int alloc[n][m], max[n][m], avail[m];


printf("Enter the Allocation Matrix:\n");

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

      for (j = 0; j < m; j++) {

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

      }

   }


printf("Enter the MAX Matrix:\n");

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

      for (j = 0; j < m; j++) {

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

      }

   }


printf("Enter the Available Resources:\n");

   for (j = 0; j < m; j++) {

scanf("%d", &avail[j]);

   }


   int f[n], ans[n], ind = 0;
```

```
  for (k = 0; k < n; k++) {
    f[k] = 0;
  }
  int need[n][m];
  for (i = 0; i< n; i++) {
    for (j = 0; j < m; j++)
      need[i][j] = max[i][j] - alloc[i][j];
  }
  int y = 0;
  for (k = 0; k < n; k++) {
    for (i = 0; i< n; i++) {
      if (f[i] == 0) {
        int flag = 0;
        for (j = 0; j < m; j++) {
          if (need[i][j] > avail[j]) {
            flag = 1;
break;
          }
        }

        if (flag == 0) {
ans[ind++] = i;
          for (y = 0; y < m; y++)
            avail[y] += alloc[i][y];
          f[i] = 1;
        }
      }
    }
  }

  int flag = 1;
  for (i = 0; i< n; i++) {
    if (f[i] == 0) {
      flag = 0;
break;
```

```
      }
    }

    if (flag == 0) {
printf("The following system is not safe\n");
    } else {
printf("Following is the SAFE Sequence\n");
      for (i = 0; i< n - 1; i++)
printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);
    }

    return 0;
}
```

Output:

Enter the number of processes: 5

Enter the number of resources: 3

Enter the Allocation Matrix:

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Enter the MAX Matrix:

7 5 3

3 2 2

9 0 2

2 2 2

4 3 3

Enter the Available Resources:

3 3 2

Following is the SAFE Sequence

P1 -> P3 -> P4 -> P0 -> P2

2a.FIRST FIT:

```
#include <stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n) {
   int allocation[n];

   for (int i = 0; i< n; i++)
      allocation[i] = -1;

   for (int i = 0; i< n; i++) {
      for (int j = 0; j < m; j++) {
         if (blockSize[j] >= processSize[i]) {
            allocation[i] = j;
blockSize[j] -= processSize[i];
break;
         }
      }
   }

printf("\nFirst Fit Allocation:\n");
printf("Process No.\tProcess Size\tBlock No.\n");
   for (int i = 0; i< n; i++) {
printf(" %d \t\t %d \t\t", i+1, processSize[i]);
      if (allocation[i] != -1)
printf("%d\n", allocation[i] + 1);
      else
printf("Not Allocated\n");
   }
}

int main() {
   int m, n;
```

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

   int blockSize[m];
printf("Enter the size of each memory block:\n");
   for (int i = 0; i< m; i++) {
printf("Block %d: ", i+1);
scanf("%d", &blockSize[i]);
   }

printf("\nEnter the number of processes: ");
scanf("%d", &n);

   int processSize[n];
printf("Enter the size of each process:\n");
   for (int i = 0; i< n; i++) {
printf("Process %d: ", i+1);
scanf("%d", &processSize[i]);
   }

firstFit(blockSize, m, processSize, n);

   return 0;
}
```

Output:

```
Enter the number of memory blocks: 5
Enter the size of each memory block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter the number of processes: 4
```

Enter the size of each process:

Process 1: 212

Process 2: 417

Process 3: 112

Process 4: 426

First Fit Allocation:

Process No.	Process Size	Block No.
1	212	2
2	417	5
3	112	2
4	426	Not Allocated

2b.BEST FIT:

```
#include <stdio.h>

void bestFit(int blockSize[], int m, int processSize[], int n) {
   int allocation[n];

   for (int i = 0; i< n; i++)
      allocation[i] = -1;

   for (int i = 0; i< n; i++) {
      int bestIdx = -1;
      for (int j = 0; j < m; j++) {
         if (blockSize[j] >= processSize[i]) {
            if (bestIdx == -1 || blockSize[j] <blockSize[bestIdx])
bestIdx = j;
         }
      }

      if (bestIdx != -1) {
         allocation[i] = bestIdx;
blockSize[bestIdx] -= processSize[i];
      }
```

```
  }

printf("\nBest Fit Allocation:\n");
printf("Process No.\tProcess Size\tBlock No.\n");
  for (int i = 0; i< n; i++) {
printf(" %d \t\t %d \t\t", i+1, processSize[i]);
    if (allocation[i] != -1)
printf("%d\n", allocation[i] + 1);
    else
printf("Not Allocated\n");
  }
}

int main() {
  int m, n;

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

  int blockSize[m];
printf("Enter the size of each memory block:\n");
  for (int i = 0; i< m; i++) {
printf("Block %d: ", i+1);
scanf("%d", &blockSize[i]);
  }

printf("\nEnter the number of processes: ");
scanf("%d", &n);

  int processSize[n];
printf("Enter the size of each process:\n");
  for (int i = 0; i< n; i++) {
printf("Process %d: ", i+1);
scanf("%d", &processSize[i]);
  }
```

```
bestFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

```
Enter the number of memory blocks: 5
Enter the size of each memory block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Best Fit Allocation:
Process No.    Process Size    Block No.
 1          212          4
 2          417          2
 3          112          3
 4          426          5
```

2c.WORST FIT

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

```
void worstFit(int blockSize[], int m, int processSize[], int n) {
   int allocation[n];

   for (int i = 0; i< n; i++)
      allocation[i] = -1;

   for (int i = 0; i< n; i++) {
      int worstIdx = -1;
      for (int j = 0; j < m; j++) {
         if (blockSize[j] >= processSize[i]) {
            if (worstIdx == -1 || blockSize[j] >blockSize[worstIdx])
worstIdx = j;
         }
      }

      if (worstIdx != -1) {
         allocation[i] = worstIdx;
blockSize[worstIdx] -= processSize[i];
      }
   }

printf("\nWorst Fit Allocation:\n");
printf("Process No.\tProcess Size\tBlock No.\n");
   for (int i = 0; i< n; i++) {
printf(" %d \t\t %d \t\t", i+1, processSize[i]);
      if (allocation[i] != -1)
printf("%d\n", allocation[i] + 1);
      else
printf("Not Allocated\n");
   }
}

int main() {
   int m, n;
```

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

   int blockSize[m];
printf("Enter the size of each memory block:\n");
   for (int i = 0; i< m; i++) {
printf("Block %d: ", i+1);
scanf("%d", &blockSize[i]);
   }

printf("\nEnter the number of processes: ");
scanf("%d", &n);

   int processSize[n];
printf("Enter the size of each process:\n");
   for (int i = 0; i< n; i++) {
printf("Process %d: ", i+1);
scanf("%d", &processSize[i]);
   }

worstFit(blockSize, m, processSize, n);

   return 0;
}
```

```
Enter the number of memory blocks: 5
Enter the size of each memory block:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
```

Process 2: 417

Process 3: 112

Process 4: 426

Worst Fit Allocation:

Process No.	Process Size	Block No.
1	212	5
2	417	2
3	112	5
4	426	Not Allocated

3.Producer consumer problem

Filename ProducerConsumerExample.java

```
import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;


// Class representing the bounded buffer

class BoundedBuffer {

  private final int capacity;

  private final Queue<Integer>queue;


  public BoundedBuffer(int capacity) {

this.capacity = capacity;

this.queue = new LinkedList<>();

  }


  // Method to produce an item

  public synchronized void produce(int value) throws InterruptedException {

    while (queue.size() == capacity) {

System.out.println("Buffer is full. Producer is waiting...");

wait();//System.exit(0);

    }

queue.add(value);
```

```
System.out.println("Produced: " + value);
notifyAll();
   }

   // Method to consume an item
   public synchronized int consume() throws InterruptedException {
      while (queue.isEmpty()) {
System.out.println("Buffer is empty. Consumer is waiting...");
wait();//System.exit(0);

      }
      int value = queue.poll();
System.out.println("Consumed: " + value);
notifyAll();
      return value;
   }
}

// Class representing a producer
class Producer implements Runnable {
   private final BoundedBufferbuffer;
   private final int itemsToProduce;

   public Producer(BoundedBuffer buffer, int itemsToProduce) {
this.buffer = buffer;
this.itemsToProduce = itemsToProduce;
   }

   @Override
   public void run() {
      try {
         for (int value = 0; value <itemsToProduce; value++) {
buffer.produce(value);
Thread.sleep((int) (Math.random() * 1000)); // Simulate work
         }
```

```
      } catch (InterruptedException e) {
Thread.currentThread().interrupt();
      }
  }
}

// Class representing a consumer
class Consumer implements Runnable {
  private final BoundedBufferbuffer;
  private final int itemsToConsume;

  public Consumer(BoundedBuffer buffer, int itemsToConsume) {
this.buffer = buffer;
this.itemsToConsume = itemsToConsume;
  }

  @Override
  public void run() {
      try {
          for (int i = 0; i<itemsToConsume; i++) {
buffer.consume();
Thread.sleep((int) (Math.random() * 1000)); // Simulate work
          }
      } catch (InterruptedException e) {
Thread.currentThread().interrupt();
      }
  }
}

// Main class to run the producer-consumer example
public class ProducerConsumerExample {
  public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      // Get buffer capacity
```

```
System.out.print("Enter buffer capacity: ");
    int bufferCapacity = scanner.nextInt();
BoundedBuffer buffer = new BoundedBuffer(bufferCapacity);

    while (true) {
      // Display menu
System.out.println("\nMenu:");
System.out.println("1. Produce or Consume");
System.out.println("3. Exit");
System.out.print("Enter your choice: ");
      int choice = scanner.nextInt();

      switch (choice) {
        case 1:
System.out.print("Enter 1 for Producer or 2 for Consumer: ");
          int subChoice = scanner.nextInt();
          if (subChoice == 1) {
            // Producer operation
System.out.print("Enter number of items to produce: ");
            int itemsToProduce = scanner.nextInt();
            Thread producer = new Thread(new Producer(buffer, itemsToProduce));
producer.start();
            try {
producer.join();
            } catch (InterruptedException e) {
Thread.currentThread().interrupt();
            }
          } else if (subChoice == 2) {
            // Consumer operation
System.out.print("Enter number of items to consume: ");
            int itemsToConsume = scanner.nextInt();
            Thread consumer = new Thread(new Consumer(buffer, itemsToConsume));
consumer.start();
            try {
consumer.join();
```

```
            } catch (InterruptedException e) {
Thread.currentThread().interrupt();
            }
          } else {
System.out.println("Invalid choice. Please enter 1 for Producer or 2 for Consumer.");
          }
break;

        case 3:
          // Exit operation
System.out.println("Exiting...");
scanner.close();
System.exit(0);
break;

        default:
System.out.println("Invalid choice. Please enter 1 or 3.");
break;
      }
    }
  }
}
```

4.Page table:

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

int main() {
  int n = 10;
  int arr[10];
  int p;
  int d;
  int i;
```

```
   int physicaladd;

   // Accepting dynamic input for the array
printf("Enter 10 values for the array:\n");
for(i = 0; i< n; i++) {
printf("Enter value for arr[%d]: ", i);
scanf("%d", &arr[i]);
   }

while(1) {
printf("Enter 1 for PageNo and Displacement \nEnter 2 to exit program \n");
scanf("%d", &i);

      switch(i) {
         case 1:
printf("Enter pageno: ");
scanf("%d", &p);

if(p < 0 || p >= n) {
printf("Invalid pageno. Please enter a value between 0 and 9.\n");
break;
            }

printf("Enter displacement: ");
scanf("%d", &d);

physicaladd = arr[p] + d;
printf("The physical address is %d \n", physicaladd);
break;

         case 2:
printf("Exiting the program.\n");
exit(0);

         default:
```

```
printf("Invalid choice. Please enter 1 or 2.\n");
    }
  }

  return 0;
}
```

```
Enter 10 values for the array:
Enter value for arr[0]: 1000
Enter value for arr[1]: 2000
Enter value for arr[2]: 3000
Enter value for arr[3]: 4000
Enter value for arr[4]: 5000
Enter value for arr[5]: 6000
Enter value for arr[6]: 7000
Enter value for arr[7]: 8000
Enter value for arr[8]: 9000
Enter value for arr[9]: 10000
Enter 1 for PageNo and Displacement
Enter 2 to exit program
1
Enter pageno: 2
Enter displacement: 2
The physical address is 3002
Enter 1 for PageNo and Displacement
Enter 2 to exit program
2
Exiting the program.
```

5.FCFS:

```
#include<stdio.h>
#include<stdlib.h>
struct Process {
  int id;
  int arrival_time;
  int burst_time;
```

```
    int waiting_time;
    int turnaround_time;
};

int compareProcesses(const void* a, const void* b) {
    struct Process* process1 = (struct Process*)a;
    struct Process* process2 = (struct Process*)b;

    if (process1->arrival_time != process2->arrival_time)
        return process1->arrival_time - process2->arrival_time;
    else
        return process1->id - process2->id;
}

// Function to calculate waiting time, turn around time, and draw Gantt chart
void calculateAndDraw(int n, struct Process processes[]) {
    // Sort the processes based on arrival time and process ID
qsort(processes, n, sizeof(struct Process), compareProcesses);

    // Calculate waiting time and turn around time
    int completion_time[n], total_wt = 0, total_tat = 0;
    for (int i = 0; i< n; i++) {
        if (i == 0)
completion_time[i] = processes[i].burst_time;
        else
completion_time[i] = completion_time[i - 1] + processes[i].burst_time;

        // Calculate waiting time
        processes[i].waiting_time = completion_time[i] - processes[i].burst_time - processes[i].arrival_time;
        if (processes[i].waiting_time< 0)
            processes[i].waiting_time = 0;
total_wt += processes[i].waiting_time;

        // Calculate turn around time
        processes[i].turnaround_time = completion_time[i] - processes[i].arrival_time;
```

```
total_tat += processes[i].turnaround_time;
    }

    // Print Gantt chart
printf("\nGantt Chart:\n");
printf(" ____________________________________________________________________________\n");
printf("|");
    for (int i = 0; i< n; i++) {
printf("  P%d   |", processes[i].id);
    }
printf("\n");
printf("|");
    for (int i = 0; i< n; i++) {
printf("   %d   |", completion_time[i]);
    }
printf("\n");

    // Print WT and TAT for each process
printf("\nProcess   Burst time   Arrival time   Waiting time   Turnaround time\n");
    for (int i = 0; i< n; i++) {
printf("   %d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, processes[i].arrival_time,
processes[i].waiting_time, processes[i].turnaround_time);
    }

    // Print average waiting time and turn around time
    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);
}

int main() {
    int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
```

```
  struct Process processes[n];
printf("Enter burst time and arrival time for each process:\n");
  for (int i = 0; i< n; i++) {
printf("Process %d:\n", i + 1);
printf("Burst time: ");
scanf("%d", &processes[i].burst_time);
printf("Arrival time: ");
scanf("%d", &processes[i].arrival_time);
    processes[i].id = i + 1;
  }

calculateAndDraw(n, processes);
  return 0;
}
```

Burst time: 3

Arrival time: 2

Process 3:

Burst time: 2

Arrival time: 1

Process 4:

Burst time: 4

Arrival time: 1

Process 5:

Burst time: 2

Arrival time: 3

Gantt Chart:

| P1 | P3 | P4 | P2 | P5 |

| 5 | 7 | 11 | 14 | 16 |

Process	Burst time	Arrival time	Waiting time	Turnaround time
1	5	0	0	5
3	2	1	4	6
4	4	1	6	10
2	3	2	9	12
5	2	3	11	13

Average Waiting Time: 6.00

Average Turnaround Time: 9.20

6.SJF:

```
#include<stdio.h>

#include<stdlib.h>

#include <limits.h>

struct Process {

   int id;

   int arrival_time;

   int burst_time;

   int waiting_time;

   int turnaround_time;

};

int compareProcesses(const void* a, const void* b) {

   struct Process* process1 = (struct Process*)a;

   struct Process* process2 = (struct Process*)b;

   if (process1->arrival_time != process2->arrival_time)

      return process1->arrival_time - process2->arrival_time;

   else

      return process1->burst_time - process2->burst_time;

}

void calculateAndDraw(int n, struct Process processes[]) {

qsort(processes, n, sizeof(struct Process), compareProcesses);

   int remaining_time[n];
```

```
    for (int i = 0; i< n; i++) {
remaining_time[i] = processes[i].burst_time;
    }
    int completion_time[n];
    int time = 0;
    while (1) {
        int shortest_burst_index = -1;
        int shortest_burst = INT_MAX;
        for (int i = 0; i< n; i++) {
            if (processes[i].arrival_time<= time &&remaining_time[i] <shortest_burst&&remaining_time[i] > 0) {
shortest_burst = remaining_time[i];
shortest_burst_index = i;
            }
        }
        if (shortest_burst_index == -1)
break;
        time += remaining_time[shortest_burst_index];
completion_time[shortest_burst_index] = time;
remaining_time[shortest_burst_index] = 0;
        processes[shortest_burst_index].waiting_time = time - processes[shortest_burst_index].arrival_time - processes[shortest_burst_index].burst_time;
        if (processes[shortest_burst_index].waiting_time< 0)
            processes[shortest_burst_index].waiting_time = 0;
        processes[shortest_burst_index].turnaround_time = time - processes[shortest_burst_index].arrival_time;
    }
    int total_wt = 0, total_tat = 0;
    for (int i = 0; i< n; i++) {
total_wt += processes[i].waiting_time;
total_tat += processes[i].turnaround_time;
    }
printf("\nGantt Chart:\n");
printf(" _______________________________________________________________________\n");
printf("|");
    for (int i = 0; i< n; i++) {
printf("  P%d   |", processes[i].id);
```

```
    }
printf("\n");
printf("|");
    for (int i = 0; i< n; i++) {
printf("   %d   |", completion_time[i]);
    }
printf("\n");
printf("\nProcess   Burst time   Arrival time   Waiting time   Turnaround time\n");
    for (int i = 0; i< n; i++) {
printf("   %d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, processes[i].arrival_time,
processes[i].waiting_time, processes[i].turnaround_time);
    }
    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);
}
int main() {
    int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
    struct Process processes[n];
printf("Enter burst time and arrival time for each process:\n");
    for (int i = 0; i< n; i++) {
printf("Process %d:\n", i + 1);
printf("Burst time: ");
scanf("%d", &processes[i].burst_time);
printf("Arrival time: ");
scanf("%d", &processes[i].arrival_time);
        processes[i].id = i + 1;
    }
calculateAndDraw(n, processes);
    return 0;}
```

Burst time: 3

Arrival time: 2

Process 3:

Burst time: 2

Arrival time: 1

Process 4:

Burst time: 4

Arrival time: 1

Process 5:

Burst time: 2

Arrival time: 3

Gantt Chart:

____

| P1 | P3 | P5 | P2 | P4 |

| 0 | 5 | 7 | 9 | 12 | 16 |

Process	Burst time	Arrival time	Waiting time	Turnaround time
1	5	0	0	5
3	2	1	4	6
4	4	1	11	15
2	3	2	7	10
5	2	3	4	6

Average Waiting Time: 5.20

Average Turnaround Time: 8.40

7.NON PREEMPTIVE:

```
#include <stdio.h>

int main() {
   int n; // Number of Processes
printf("Enter the number of processes: ");
```

```
scanf("%d", &n);

   int arrivaltime[n], bursttime[n], priority[n], waitingTime[n], turnaroundTime[n];
   int CPU = 0, allTime = 0;

printf("Enter arrival time, burst time, and priority for each process:\n");
   for (int i = 0; i< n; i++) {
printf("For Process %d:\n", i + 1);
printf("Arrival Time: ");
scanf("%d", &arrivaltime[i]);
printf("Burst Time: ");
scanf("%d", &bursttime[i]);
printf("Priority: ");
scanf("%d", &priority[i]);
   }

   int ATt[n], PPt[n];
   int NoP = n;
   int i = 0;

   for (i = 0; i< n; i++) {
PPt[i] = priority[i];
ATt[i] = arrivaltime[i];
   }

   int LAT = 0;
   for (i = 0; i< n; i++)
      if (arrivaltime[i] > LAT)
         LAT = arrivaltime[i];

   int MAX_P = 0;
   for (i = 0; i< n; i++)
      if (PPt[i] > MAX_P)
         MAX_P = PPt[i];
```

```
   int ATi = 0, P1 = PPt[0], P2 = PPt[0];
   int j = -1;
   while (NoP> 0 && CPU <= 1000) {
      for (i = 0; i< n; i++) {
         if ((ATt[i] <= CPU) && (ATt[i] != (LAT + 10))) {
            if (PPt[i] != (MAX_P + 1)) {
               P2 = PPt[i];
               j = 1;

               if (P2 < P1) {
                  j = 1;
ATi = i;
                  P1 = PPt[i];
                  P2 = PPt[i];
               }
            }
         }
      }

      if (j == -1) {
         CPU = CPU + 1;
continue;
      } else {
waitingTime[ATi] = CPU - ATt[ATi];
         CPU = CPU + bursttime[ATi];
turnaroundTime[ATi] = CPU - ATt[ATi];
ATt[ATi] = LAT + 10;
         j = -1;
PPt[ATi] = MAX_P + 1;
ATi = 0;
         P1 = MAX_P + 1;
         P2 = MAX_P + 1;
NoP = NoP - 1;
      }
   }
```

```
   printf("\nProcess_Number\tBurst_Time\tPriority\tArrival_Time\tWaiting_Time\tTurnaround_Time\n\n");
   for (i = 0; i< n; i++) {
printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, bursttime[i], priority[i], arrivaltime[i], waitingTime[i], turnaroundTime[i]);
   }

   float AvgWT = 0, AVGTaT = 0;
   for (i = 0; i< n; i++) {
AvgWT = waitingTime[i] + AvgWT;
AVGTaT = turnaroundTime[i] + AVGTaT;
   }

printf("Average waiting time = %f\n", AvgWT / n);
printf("Average turnaround time = %f\n", AVGTaT / n);

   return 0;
}
```

Arrival Time: 1

Burst Time: 2

Priority: 3

For Process 4:

Arrival Time: 1

Burst Time: 4

Priority: 4

For Process 5:

Arrival Time: 3

Burst Time: 2

Priority: 1

Process_Number	Burst_Time	Priority	Arrival_Time	Waiting_Time	Turnaround_Time
P1	5	2	0	0	5
P2	3	1	2	3	6
P3	2	3	1	9	11
P4	4	4	1	11	15

P5 2 1 3 5 7

Average waiting time = 5.600000

Average turnaround time = 8.800000

Gantt Chart:

____

| P1 | P2 | P5 | P3 | P4 |

| 0 | 5 | 8 | 10 | 12 | 16 |

PARENT CHILD PROCESS:

Shell program:

Largest of three number

```
echo "Enter three Integers:"

read a b c

if [ $a -gt $b -a $a -gt $c ];then

echo "$a is Greatest"

elif [ $b -gt $c -a $b -gt $a ];then

echo "$b is Greatest"

else

echo "$c is Greatest!"

fi
```

factorial number

```
echo"Enter a number"
read num

fact=1

while [ $num -gt 1 ];do
fact=$((fact * num))#fact = fact * num
num=$((num - 1))#num = num - 1
done
```

```
echo $fact
```

sum of digits:

```
#!/bin/bash
echo "Enter a Number:"
read  n
temp=$n
sd=0
sum=0
while [ $n -gt0 ];do
sd=$((  $n  %  10 ))
n=$(( $n  /  10 ))
sum=$((  $sum  +  $sd  ))
done
echo "Sum is $sum"
```

reverse a number:

```
echo enter n
read n
num=0
temp = $n
while [ $temp –gt 0 ];do
num=$(( $num  % 10))
k=$((k *  10 + num))
temp=$(( temp/10))
done
echo “number is” $k
```

Fibonacci series:

```
#!/bin/bash

echo "How many numbers do you want of Fibonacci series ?"
```

```
read total

x=0

y=1

i=2

echo "Fibonacci Series up to $total terms :: "

echo "$x"

echo "$y"

while [ $i -lt $total ]

do

i=`expr $i + 1 `

z=`expr $x + $y `

echo "$z"

x=$y

y=$z

done
```

Armstrong number:

```
#!/bin/bash


# Function to calculate the power of a number

power() {

  local base=$1

  local exp=$2

  local result=1


  for (( i=0; i<$exp; i++ )); do

    result=$(( result * base ))
```

```
  done

  echo $result
}

# Function to check if a number is an Armstrong number
is_armstrong() {
  local num=$1
  local sum=0
  local temp=$num
  local digits=${#num}

  while [ $temp -gt0 ]; do
    digit=$(( temp % 10 ))
    temp=$(( temp / 10 ))
    sum=$(( sum + $(power $digit $digits) ))
  done

  if [ $sum -eq $num ]; then
    echo "$num is an Armstrong number."
  else
    echo "$num is not an Armstrong number."
  fi
}

# Check if a number is provided as an argument
if [ $# -ne 1 ]; then
  echo "Usage: $0 <number>"
  exit 1
fi

# Assign the argument to a variable
number=$1

# Check if the provided argument is a positive integer
```

```
if ! [[ $number =~ ^[0-9]+$ ]]; then
  echo "Error: Argument must be a positive integer."
  exit 1
fi

# Check if the number is an Armstrong number
is_armstrong $number
```