Q1. C program to demonstrate the working of fork() system call

Source Code:

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t pid;

printf("Before fork()\n");

pid = fork();

if (pid == 0) {

printf("Child process: PID = %d\n", getpid());

} else if (pid > 0) {

printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);

} else {

printf("Fork failed\n");

return 1;

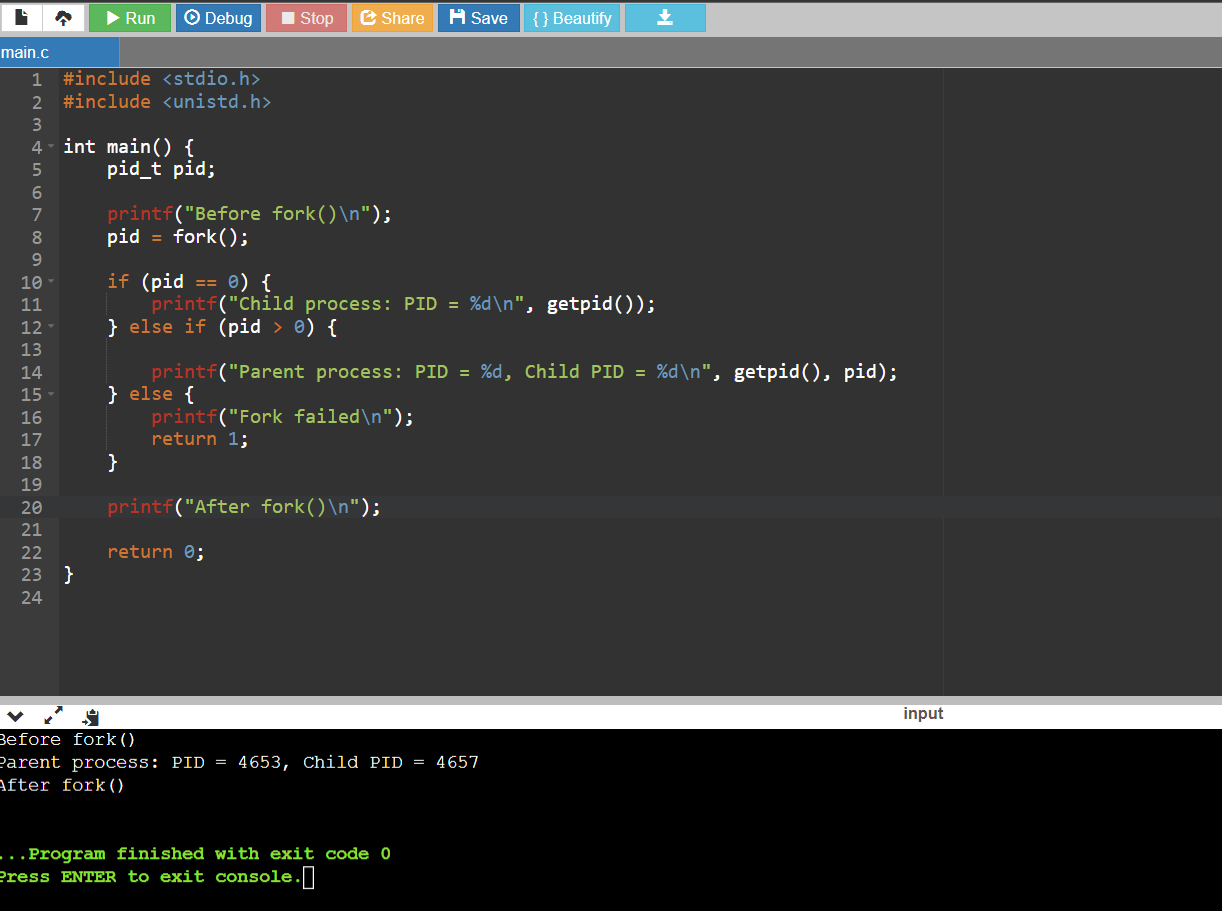
}

printf("After fork()\n");

return 0;

}

Output:



Q2. C program in which parent process computes the SUM OF ODD NUMBERS and child process computes the sum of EVEN NUMBERS stored in array using fork().

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define ARRAY\_SIZE 10

int main() {

int arr[ARRAY\_SIZE];

int sum\_even = 0, sum\_odd = 0;

printf("Enter %d elements for the array:\n", ARRAY\_SIZE);

for (int i = 0; i < ARRAY\_SIZE; i++) {

printf("Enter element %d: ", i + 1);

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

}

pid\_t pid = fork();

if (pid == 0) {

for (int i = 0; i < ARRAY\_SIZE; i++) {

if (arr[i] % 2 == 0)

sum\_even += arr[i];

}

printf("Child process: Sum of even numbers = %d\n", sum\_even);

} else if (pid > 0) {

for (int i = 0; i < ARRAY\_SIZE; i++) {

if (arr[i] % 2 != 0)

sum\_odd += arr[i];

}

printf("Parent process: Sum of odd numbers = %d\n", sum\_odd);

} else {

printf("Fork failed\n");

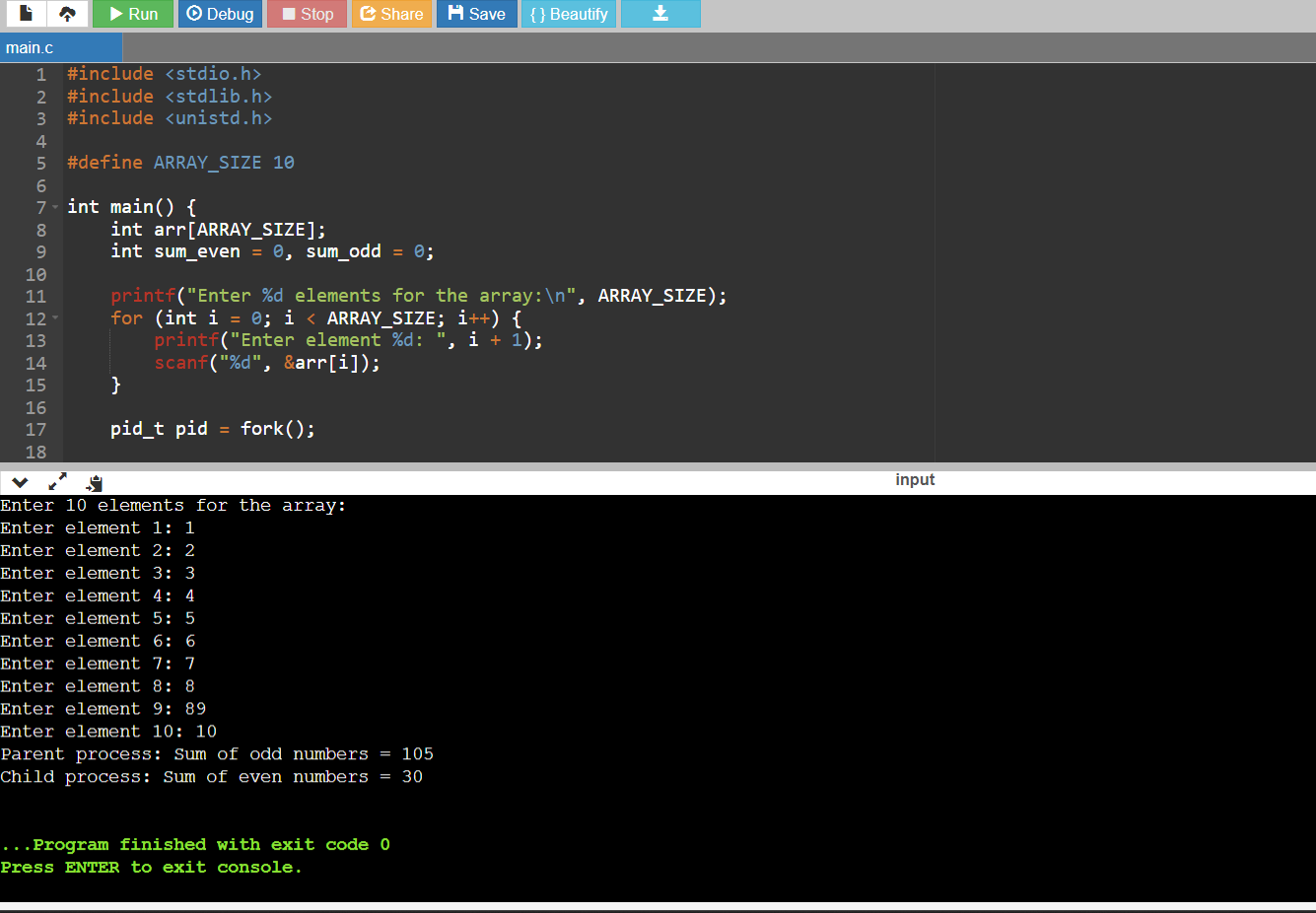
return 1;

}

return 0;

}

Output:



Q3.C program to implement the Zombie process

Source code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

printf("Child process created, PID = %d\n", getpid());

exit(EXIT\_SUCCESS);

} else {

printf("Parent process, PID = %d\n", getpid());

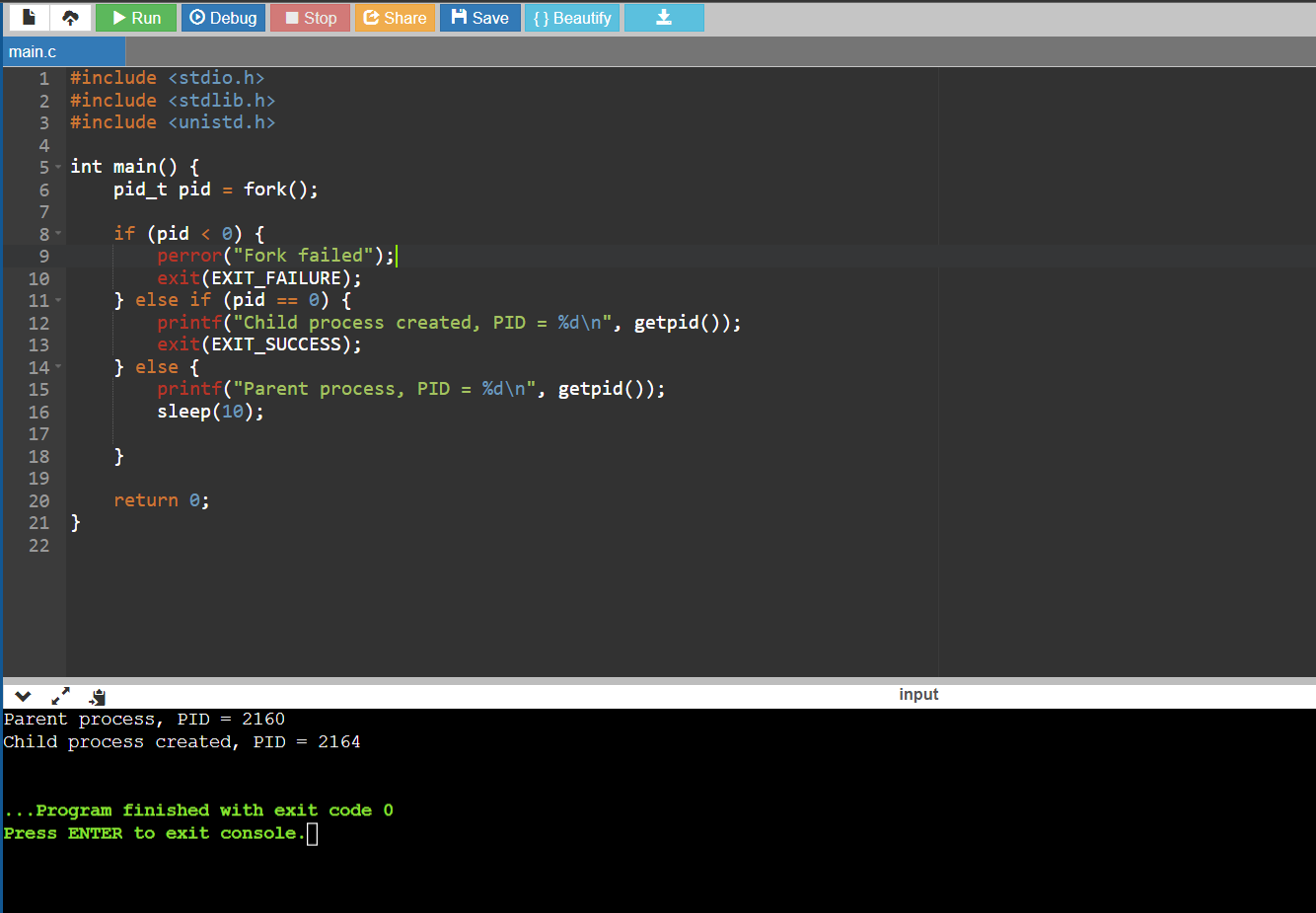
sleep(10);

}

return 0;

}

Output:



Q4. C program to implement the orphan process

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

printf("Child process created, PID = %d\n", getpid());

sleep(10);

printf("Child process exiting\n");

} else {

printf("Parent process, PID = %d\n", getpid());

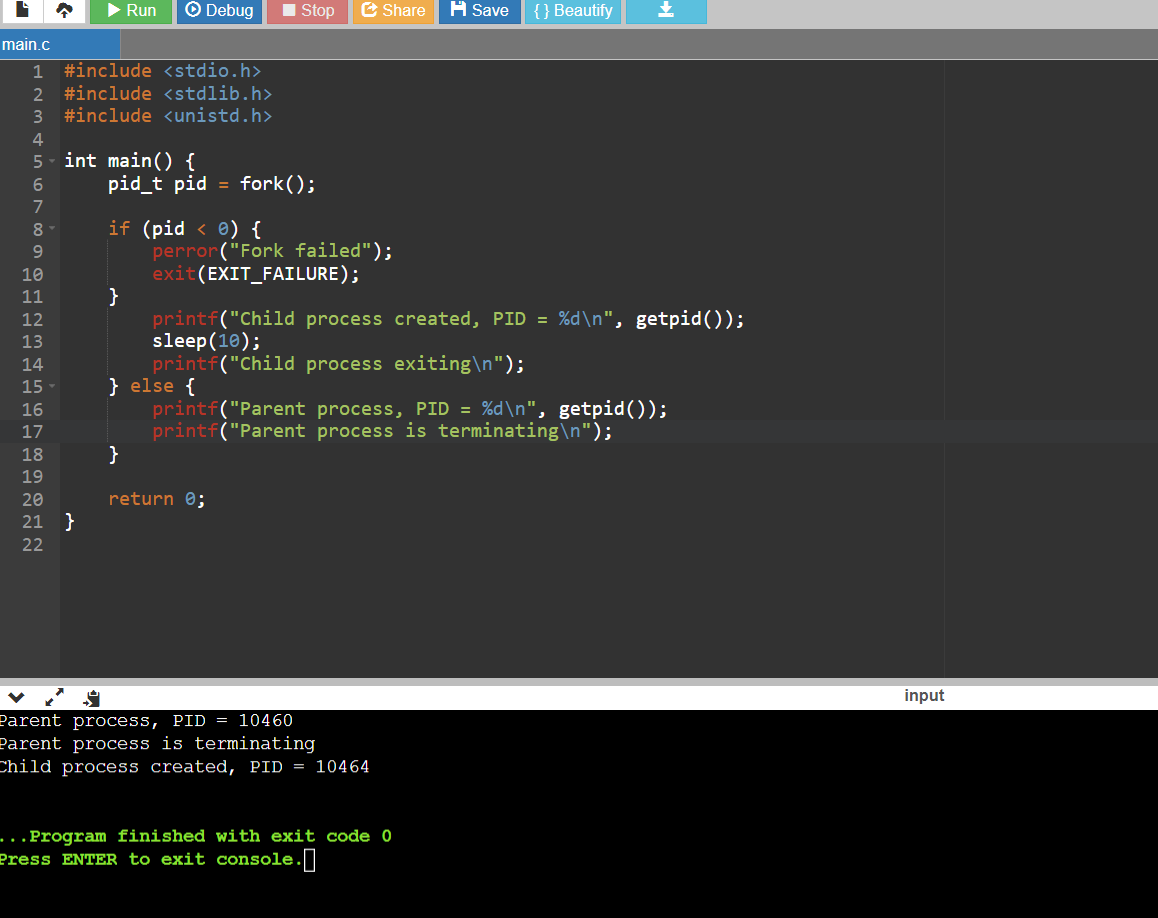
printf("Parent process is terminating\n");

}

return 0;

}

Output:



Q5. C program to implement FCFS CPU Scheduling Algorithm

Source Code:

#include <stdio.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

};

void calculateWaitingTime(struct Process processes[], int n, int waiting\_time[]) {

waiting\_time[0] = 0;

for (int i = 1; i < n; i++) {

waiting\_time[i] = waiting\_time[i - 1] + processes[i - 1].burst\_time;

}

}

void calculateTurnaroundTime(struct Process processes[], int n, int waiting\_time[], int turnaround\_time[]) {

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

turnaround\_time[i] = processes[i].burst\_time + waiting\_time[i];

}

}

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

int waiting\_time[n], turnaround\_time[n];

calculateWaitingTime(processes, n, waiting\_time);

calculateTurnaroundTime(processes, n, waiting\_time, turnaround\_time);

float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

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

avg\_waiting\_time += waiting\_time[i];

avg\_turnaround\_time += turnaround\_time[i];

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("Process\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", processes[i].pid, processes[i].burst\_time, waiting\_time[i], turnaround\_time[i]);

}

printf("Average Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

printf("Enter arrival time for process %d: ", i + 1);

processes[i].pid = i + 1;

printf("Enter arrival time for process %d: ", i + 1);

scanf("%d", &processes[i].arrival\_time);

printf("Enter burst time for process %d: ", i + 1);

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

}

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].arrival\_time > processes[j + 1].arrival\_time) {

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

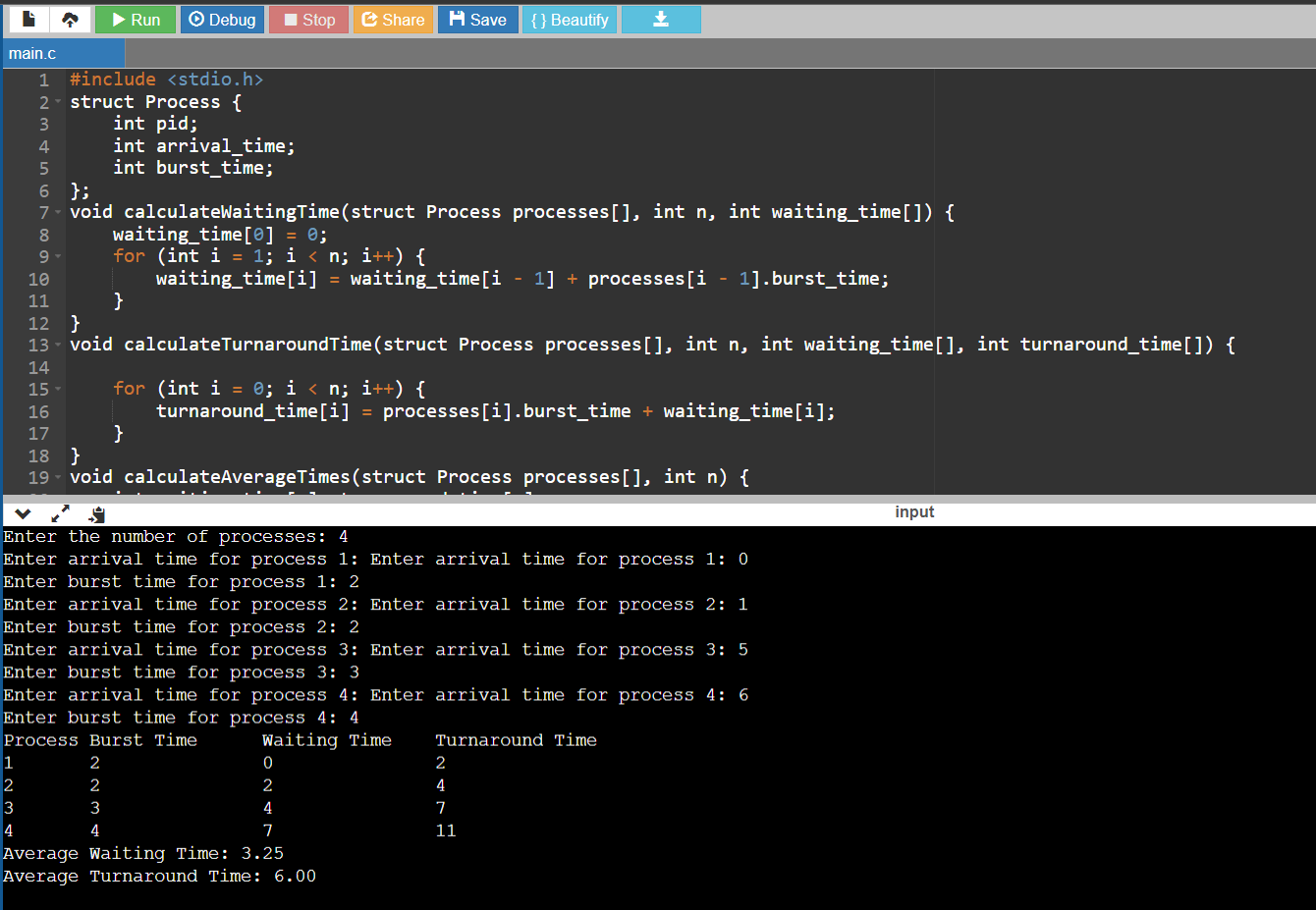
}

calculateAverageTimes(processes, n);

return 0;

}

Output:



Q6. C program to implement SRTF CPU Scheduling Algorithm

Source Code:

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define MAX\_PROCESSES 10

typedef struct {

int pid; // Process ID

int burst\_time; // Burst time

int remaining\_time; // Remaining burst time

} Process;

void srtf\_schedule(Process processes[], int n) {

int current\_time = 0;

int completed = 0;

int shortest = INT\_MAX;

int shortest\_index = -1;

printf("SRTF Schedule:\n");

printf("Time\tProcess\n");

while (completed < n) {

shortest = INT\_MAX;

shortest\_index = -1;

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

if (processes[i].remaining\_time > 0 && processes[i].remaining\_time < shortest && processes[i].burst\_time > 0 && processes[i].pid != -1) {

shortest = processes[i].remaining\_time;

shortest\_index = i;

}

}

if (shortest\_index == -1) {

current\_time++;

continue;

}

processes[shortest\_index].remaining\_time--;

current\_time++;

if (processes[shortest\_index].remaining\_time == 0) {

printf("%d-%d\tP%d\n", current\_time - 1, current\_time, processes[shortest\_index].pid);

completed++;

processes[shortest\_index].pid = -1;

}

}

}

int main() {

int n;

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

scanf("%d", &n);

if (n > MAX\_PROCESSES || n <= 0) {

printf("Invalid number of processes. Maximum allowed: %d\n", MAX\_PROCESSES);

return 1;

}

Process processes[MAX\_PROCESSES];

printf("Enter burst times for each process:\n");

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

processes[i].pid = i + 1;

printf("Burst time for P%d: ", i + 1);

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

processes[i].remaining\_time = processes[i].burst\_time;

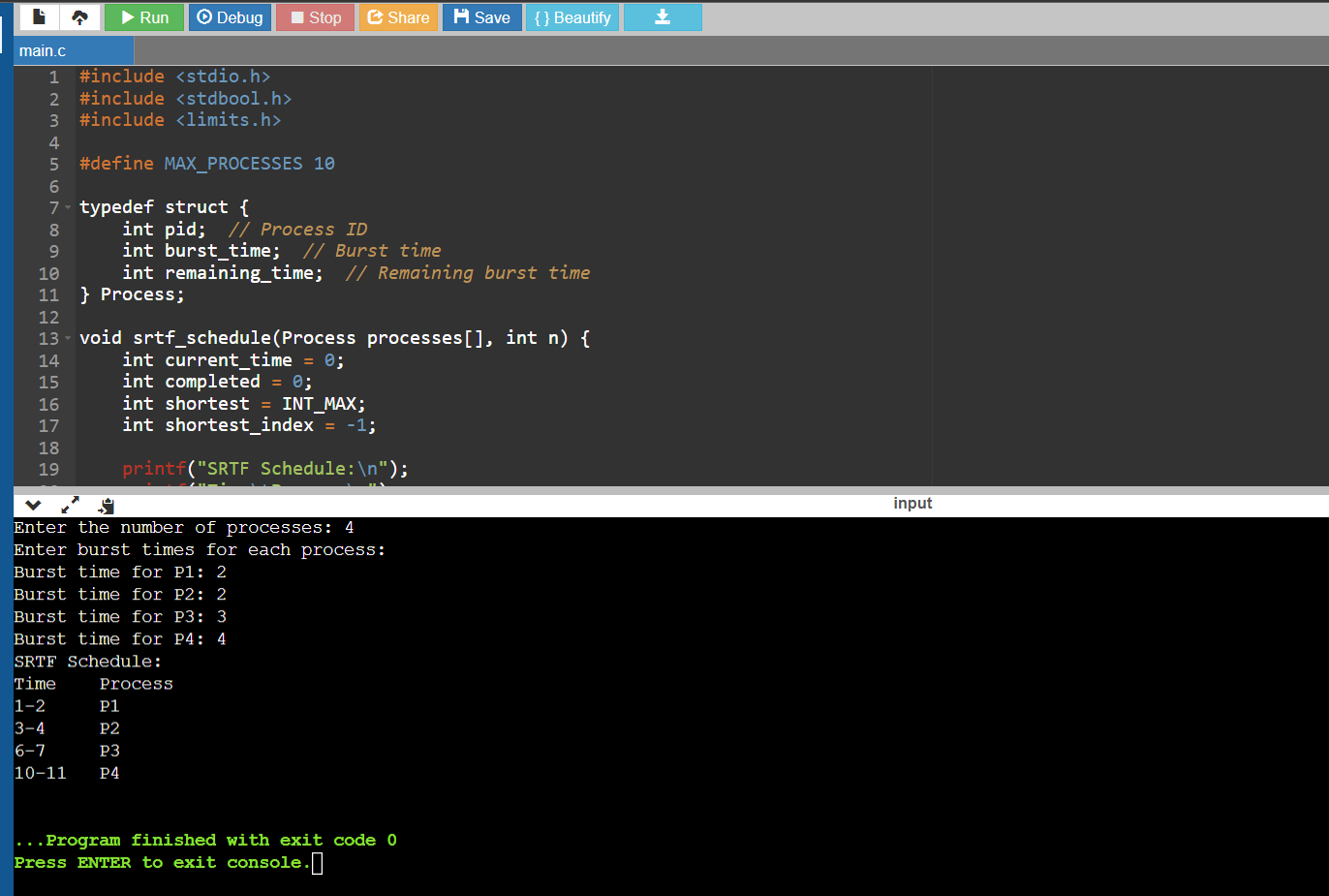
}

srtf\_schedule(processes, n);

return 0;

}

Output:



Q7. C program to implement Round Robin CPU Scheduling Algorithm

Source Code:

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int remaining\_time;

};

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

int remaining\_processes = n;

int current\_time = 0;

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

processes[i].remaining\_time = processes[i].burst\_time;

}

while (remaining\_processes > 0) {

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

if (processes[i].remaining\_time <= 0) {

continue;

}

int execute\_time = (processes[i].remaining\_time < quantum) ? processes[i].remaining\_time : quantum;

processes[i].remaining\_time -= execute\_time;

current\_time += execute\_time;

printf("Time %d: Process %d is running for %d units.\n", current\_time, processes[i].pid, execute\_time);

if (processes[i].remaining\_time <= 0) {

remaining\_processes--;

int turnaround\_time = current\_time;

printf("Time %d: Process %d completed. Turnaround time = %d\n", current\_time, processes[i].pid, turnaround\_time);

}

}

}

}

int main() {

int n, quantum;

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

scanf("%d", &n);

printf("Enter the time quantum: ");

scanf("%d", &quantum);

struct Process processes[n];

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

processes[i].pid = i + 1;

printf("Enter burst time for process %d: ", i + 1);

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

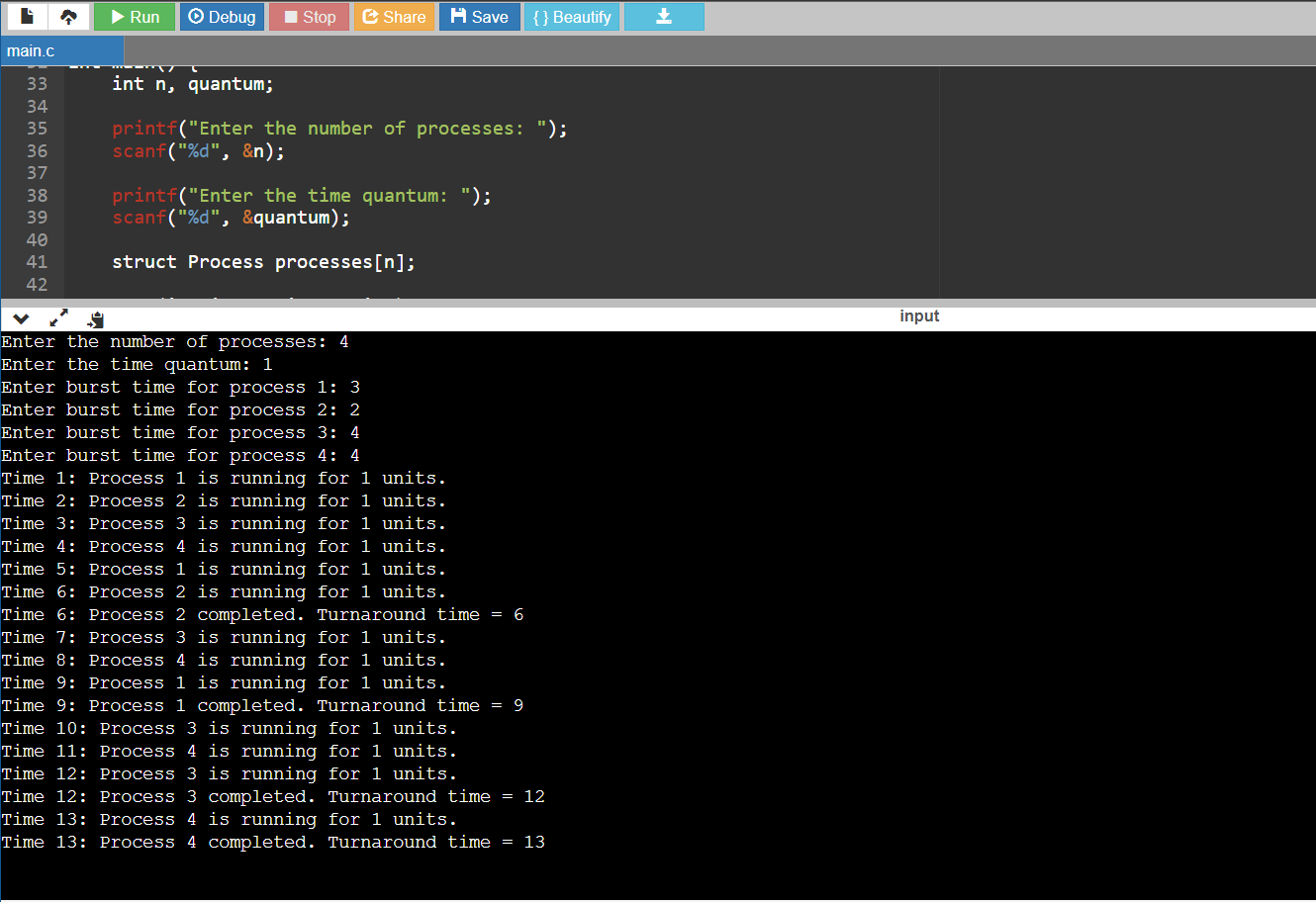
}

roundRobin(processes, n, quantum);

return 0;

}

Output:



1.Write a c code for non-preemptive Shortest Job First (SJF) scheduling is a scheduling algorithm

Source Code:

#include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct {

int process\_id;

int burst\_time;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void sjf\_nonpreemptive(Process processes[], int n) {

int waiting\_time[MAX\_PROCESSES], turnaround\_time[MAX\_PROCESSES];

float average\_waiting\_time = 0, average\_turnaround\_time = 0;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].burst\_time > processes[j + 1].burst\_time) {

swap(&processes[j], &processes[j + 1]);

}

}

}

waiting\_time[0] = 0;

turnaround\_time[0] = processes[0].burst\_time;

for (int i = 1; i < n; i++) {

waiting\_time[i] = waiting\_time[i - 1] + processes[i - 1].burst\_time;

turnaround\_time[i] = waiting\_time[i] + processes[i].burst\_time;

average\_waiting\_time += waiting\_time[i];

average\_turnaround\_time += turnaround\_time[i];

}

average\_waiting\_time /= n;

average\_turnaround\_time /= n;

printf("Process\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", processes[i].process\_id, processes[i].burst\_time,

waiting\_time[i], turnaround\_time[i]);

}

printf("\nAverage Waiting Time: %.2f\n", average\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", average\_turnaround\_time);

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

printf("Enter burst time for each process:\n");

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

processes[i].process\_id = i + 1;

printf("Burst time for process %d: ", i + 1);

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

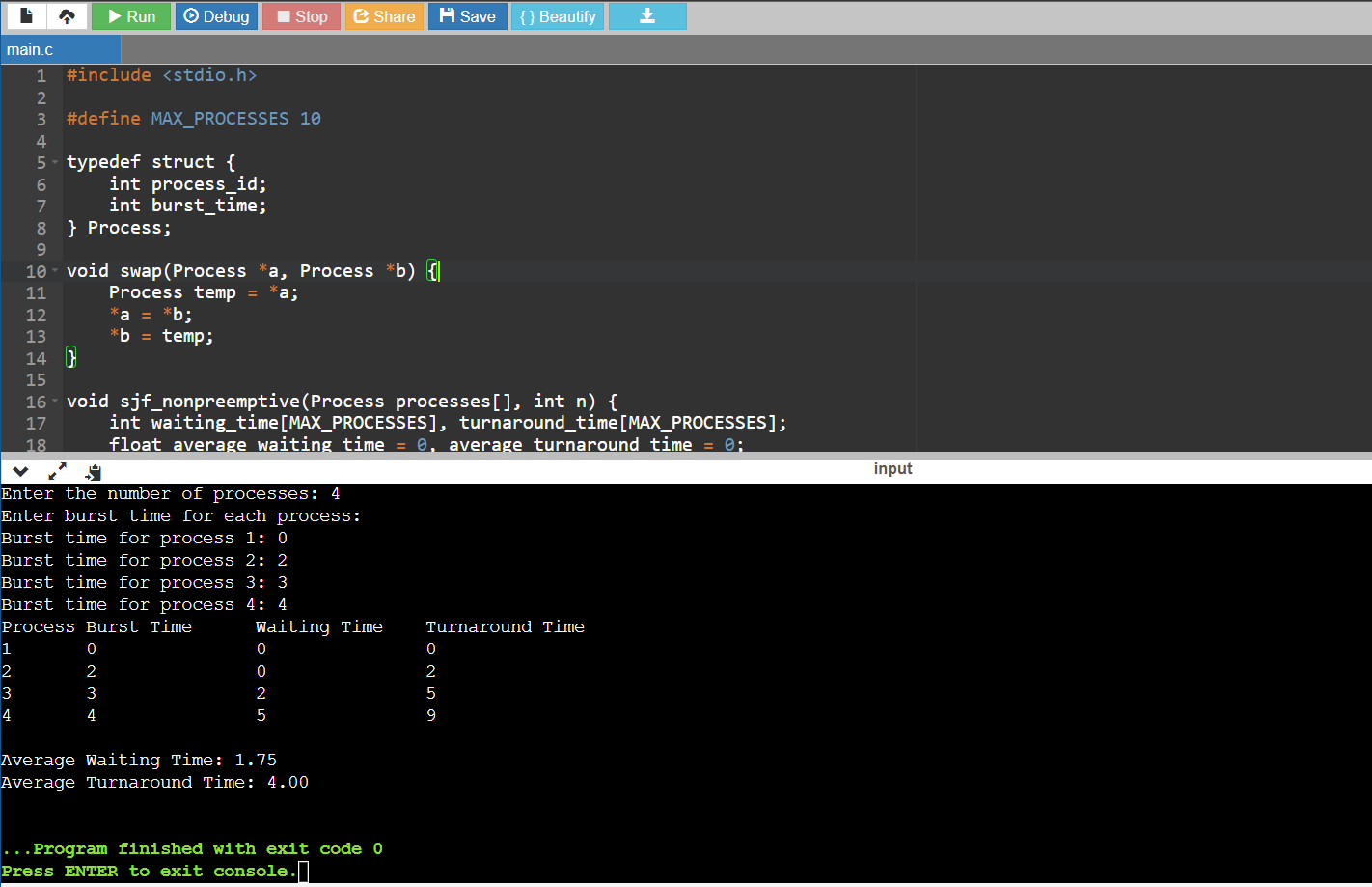
}

sjf\_nonpreemptive(processes, n);

return 0;

}

Output:



2. Write a c code for Non-preemptive Priority Scheduling is a scheduling algorithm

Source Code:

#include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct {

int process\_id;

int priority;

int burst\_time;

} Process;

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

void priority\_nonpreemptive(Process processes[], int n) {

int waiting\_time[MAX\_PROCESSES], turnaround\_time[MAX\_PROCESSES];

float average\_waiting\_time = 0, average\_turnaround\_time = 0;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].priority < processes[j + 1].priority) {

swap(&processes[j], &processes[j + 1]);

}

}

}

waiting\_time[0] = 0;

turnaround\_time[0] = processes[0].burst\_time;

for (int i = 1; i < n; i++) {

waiting\_time[i] = waiting\_time[i - 1] + processes[i - 1].burst\_time;

turnaround\_time[i] = waiting\_time[i] + processes[i].burst\_time;

average\_waiting\_time += waiting\_time[i];

average\_turnaround\_time += turnaround\_time[i];

}

average\_waiting\_time /= n;

average\_turnaround\_time /= n;

// Display the schedule and timing information

printf("Process\tPriority\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process\_id, processes[i].priority,

processes[i].burst\_time, waiting\_time[i], turnaround\_time[i]);

}

printf("\nAverage Waiting Time: %.2f\n", average\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", average\_turnaround\_time);

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

printf("Enter priority and burst time for each process:\n");

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

processes[i].process\_id = i + 1;

printf("Priority for process %d: ", i + 1);

scanf("%d", &processes[i].priority);

printf("Burst time for process %d: ", i + 1);

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

}

priority\_nonpreemptive(processes, n);

return 0;

}

Output:

