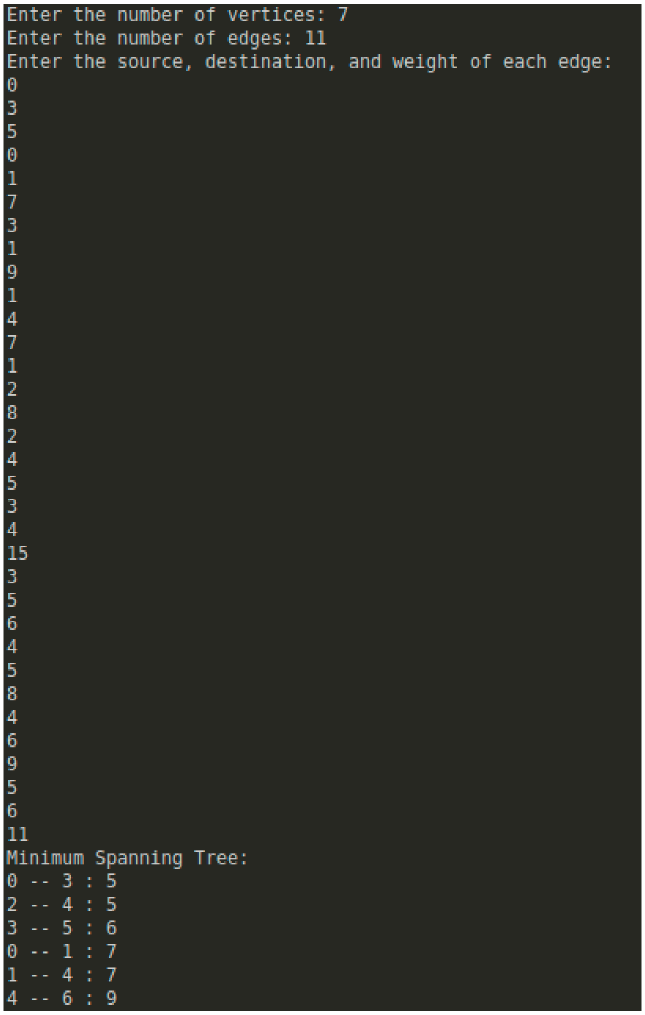
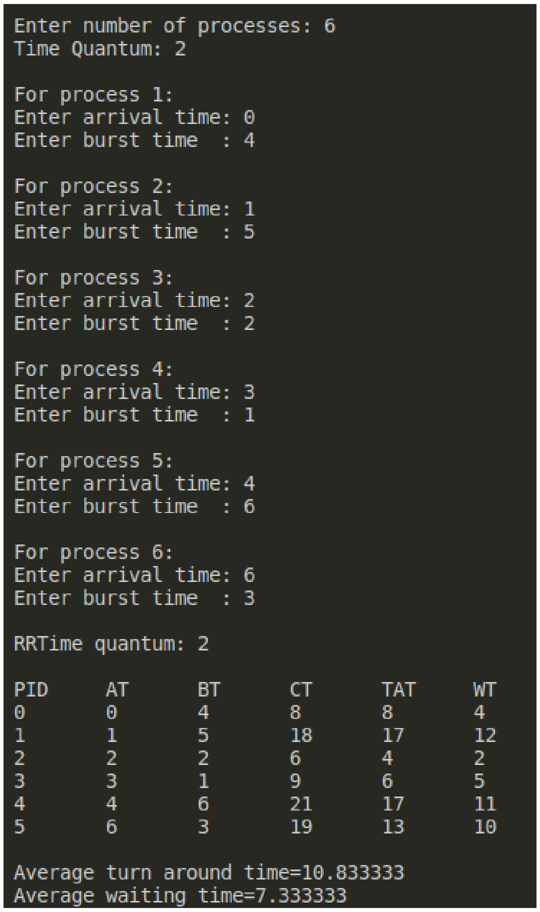
```
//ROUND ROBIN
#include <stdio.h>
#include <stdlib.h>

struct process
{
	int pid, at, wt, tat, bt_print, bt, ct, scheduled;
} p[20],temp;

int n, tq;
float twt, ttat, avwt, avtat;
int time_units=0;
int tail=-1, readyqueue[20];

void push(int idx)
{
	readyqueue[++tail]=idx;
}

int pop()
{
	int idx,i;
	if(tail==-1)
		return -1;
	idx=readyqueue[0];
	for(i=1;i<=tail;++i)
		readyqueue[i-1]=readyqueue[i];
	--tail;
	return idx;
}

void display_item(int i)
{

printf("%d\t%d\t%d\t%d\t%d\t%d\n",p[i].pid,p[i].at,p[i].bt_print,p[i].ct,p[i].tat,p[i].wt);
}

void display()
{
	int i;
	printf("Time quantum: %d\n",tq);
	printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");
	for(i=0;i<n;i++)
		display_item(i);
	printf("\nAverage turn around time=%f\nAverage waiting time=%f\n",avtat,avwt);
}

void swap(int i, int j)
{
	temp=p[i];
	p[i]=p[j];
	p[j]=temp;
}

void pid_sort()
{
	int i,j;
	for(i=0;i<n-1;i++)
```

```
                for(j=0;j<n-i-1;j++)
                        if(p[j].pid>p[j+1].pid)
                                swap(j,j+1);
}

void at_sort()
{
        int i,j;
        for(i=0;i<n-1;i++)
                for(j=0;j<n-i-1;j++)
                        if(p[j].at>p[j+1].at)
                                swap(j,j+1);
}

int execute(int i)
{
        int time=0;
        if(p[i].bt>tq)
        {
                time=tq;
                time_units+=time;
        }
        else
        {
                time=p[i].bt;
                time_units+=time;
                p[i].ct=time_units;
                p[i].tat=p[i].ct-p[i].at;
                p[i].wt=p[i].tat-p[i].bt_print;
                ttat+=p[i].tat;
                twt+=p[i].wt;
        }
        p[i].bt=p[i].bt-time;
        return time;
}

void anat(int last_exec)
{
        int i;
        for(i=0;i<n;++i)
        {
                if(p[i].bt==0)
                        continue;
                if(p[i].scheduled==1)
                        continue;
                if(p[i].at<=time_units)
                {
                        push(i);
                        p[i].scheduled=1;
                }
        }
}

void rr()
{
        int current_task;
        int time;
        at_sort();
        push(0);
        p[0].scheduled=1;
        while(1)
        {
                current_task=pop();
                if(current_task==-1)
                        break;
                time=execute(current_task);
                anat(current_task);
                if(p[current_task].bt>0)
                        push(current_task);
        }
        avtat=ttat/n;
        avwt=twt/n;
        pid_sort();
        printf("\nRR");
        display();
}

int main()
{
        int i;
        printf("\nEnter number of processes: ");
        scanf("%d",&n);
        printf("Time Quantum: ");
        scanf("%d",&tq);
        for(i=0;i<n;i++)
        {
                printf("\nFor process %d: \n",i+1);
         p[i].pid = i;
         printf("Enter arrival time: ");
         scanf("%d",&p[i].at);
         printf("Enter burst time  : ");
         scanf("%d",&p[i].bt);
                p[i].bt_print=p[i].bt;
        }
        rr();
```



```
Enter the number of vertices: 7
Enter the number of edges: 11
Enter the source, destination, and weight of each edge:
0
3
5
0
1
7
3
1
9
1
4
7
1
2
8
2
4
5
3
4
15
3
5
6
4
5
8
4
6
9
5
6
11
Minimum Spanning Tree:
0 -- 3 : 5
2 -- 4 : 5
3 -- 5 : 6
0 -- 1 : 7
1 -- 4 : 7
4 -- 6 : 9
```



```
Enter number of processes: 6
Time Quantum: 2

For process 1:
Enter arrival time: 0
Enter burst time  : 4

For process 2:
Enter arrival time: 1
Enter burst time  : 5

For process 3:
Enter arrival time: 2
Enter burst time  : 2

For process 4:
Enter arrival time: 3
Enter burst time  : 1

For process 5:
Enter arrival time: 4
Enter burst time  : 6

For process 6:
Enter arrival time: 6
Enter burst time  : 3

RRTime quantum: 2

PID     AT      BT      CT      TAT     WT
0       0       4       8       8       4
1       1       5       18      17      12
2       2       2       6       4       2
3       3       1       9       6       5
4       4       6       21      17      11
5       6       3       19      13      10

Average turn around time=10.833333
Average waiting time=7.333333
```

```
//KRUSKALS ALGORITHM
#include <stdio.h>
#include <stdlib.h>

// Structure to represent an edge in the graph
struct Edge {
    int source, destination, weight;
};

// Function prototypes
void kruskalMST(struct Edge graph[], int V, int E);
int find(int parent[], int i);
void unionSets(int parent[], int rank[], int x, int y);

int main() {
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    struct Edge* graph = (struct Edge*)malloc(E * sizeof(struct Edge));

    printf("Enter the source, destination, and weight of each edge:\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &graph[i].source, &graph[i].destination, &graph[i].weight);
    }

    kruskalMST(graph, V, E);

    free(graph);
    return 0;
}

// Comparison function used by qsort() to sort edges in non-decreasing order of their weight
int compareEdges(const void* a, const void* b) {
    struct Edge* edge1 = (struct Edge*)a;
    struct Edge* edge2 = (struct Edge*)b;
    return edge1->weight - edge2->weight;
}

// Kruskal's algorithm to find the Minimum Spanning Tree
void kruskalMST(struct Edge graph[], int V, int E) {
    // Sort all the edges in non-decreasing order of their weight
    qsort(graph, E, sizeof(struct Edge), compareEdges);

    int* parent = (int*)malloc(V * sizeof(int));
    int* rank = (int*)malloc(V * sizeof(int));

    // Initialize parent and rank arrays
    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }

    struct Edge* MST = (struct Edge*)malloc((V - 1) * sizeof(struct Edge)); // MST will have
V-1 edges
    int edgeCount = 0; // Number of edges included in the MST
    int i = 0; // Index variable for the sorted edges

    while (edgeCount < V - 1 && i < E) {
        struct Edge nextEdge = graph[i++];

        int sourceParent = find(parent, nextEdge.source);
        int destParent = find(parent, nextEdge.destination);

        // If including this edge does not create a cycle, add it to the MST
        if (sourceParent != destParent) {
            MST[edgeCount++] = nextEdge;
            unionSets(parent, rank, sourceParent, destParent);
        }
    }

    printf("Minimum Spanning Tree:\n");
    for (int j = 0; j < edgeCount; j++) {
        printf("%d -- %d : %d\n", MST[j].source, MST[j].destination, MST[j].weight);
    }

    free(MST);
    free(rank);
    free(parent);
}

// Find the subset of an element 'i'
int find(int parent[], int i) {
    if (parent[i] != i) {
        parent[i] = find(parent, parent[i]);
    }
    return parent[i];
}

// Perform union of two subsets 'x' and 'y'
void unionSets(int parent[], int rank[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);

    if (rank[xroot] < rank[yroot]) {
        parent[xroot] = yroot;
    } else if (rank[xroot] > rank[yroot]) {
        parent[yroot] = xroot;
    } else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}
```