1. Sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the brute force method works along with its time complexity analysis: worst case, average case and best case.

```
package DAALAB;
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;
public class selectionSortSmi {
static int max = 10000;
static void selectionSort(int a[],int n)
{
int min;
for (int i=0;i<n-1;i++)
{
min=i;
for (int j=i+1;j<n;j++)
{
if (a[j] < a[min])
{
min = j;
}
}
int temp;
temp = a[min];
a[min] = a[i];
a[i] = temp;
}
}
public static void main(String[] args)
{
long start, end;
int n;
Scanner in = new Scanner(System.in);
System.out.println("Enter a value of n");
n=in.nextInt();
int a[]=new int[n];
int ch;
System.out.println("Selection Sort");
System.out.println("1. Best case");
System.out.println("2. Average case");
System.out.println("3. Worst case");
ch=in.nextInt();
System.out.println("Array before sorting");
switch(ch)
{
```

```java
case 1:
for(int i=0;i<n;i++)
{
a[i]=i+1;
System.out.print("\t"+a[i]);
}
break;
case 2:
Random random=new Random();
for(int i = 0;i<n;i++)
{
a[i]=random.nextInt(100);
System.out.print("\t"+a[i]);
}
break;
case 3:
for(int i=0;i<n;i++)
{
a[i]=n-i;
System.out.print("\t"+a[i]);
}
break;
}
start=System.currentTimeMillis();
selectionSort(a,n);
end=System.currentTimeMillis();
System.out.println("\nThe sorted elements are : ");
for(int i=0; i<n; i++)
System.out.print("\t"+a[i]);
System.out.println("\nThe time taken to sort is "+(end-start)+"ms");
}
}
```

2. Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide-and-conquer method works along with its time complexity analysis: worst case,average case and best case

```
package p;

import java.util.Random;

import java.util.Scanner;

class quicksort {

static int compari=0;

static int[] arr;

static void quicksort(int low,int high)

{

if(low<high) {

compari+=1;

int j=partition(low,high);

quicksort(low,j-1);

quicksort(j+1,high);

}

}

static int partition(int low,int high) {

int pivot=arr[low];

int i=low,j=high;

while (i<j) {

compari +=1;

while(i<high&&arr[i]<=pivot) {

compari +=2;
```

```java
i=i+1;
}
while(j>low&&arr[j]>=pivot) {
compari+=2;
j=j-1;
}
if(i<j) {
compari +=1;
interchange(i,j);
}
}
arr[low]=arr[j];
arr[j]=pivot;
return j;
}
static void interchange(int i,int j) {
int temp=arr[i];
arr[i]=arr[j];
arr[j]=temp;
}
public static void main(String[]args) {
int n;
Scanner scanner=new Scanner(System.in);
System.out.print("enter value of n:");
n=scanner.nextInt();
arr=new int[n];
System.out.println("quicksort");
System.out.println("1.best/avg case");
```

```
System.out.println("2.worst case");
int ch=scanner.nextInt();
switch(ch) {
case 1:
Random random=new Random(3000);
for(int i=0;i<n;i++) {
arr[i]=random.nextInt(5000);
}
break;
case 2:
for(int i=0;i<n;i++) {
arr[i]=i+1;
}
break;
}
long start=System.nanoTime();
quicksort(0,n-1);
long end=System.nanoTime();
System.out.println("sorted array");
for(int i=0;i<n;i++) {
System.out.println(arr[i]);
}
System.out.println("time taken:"+(end-start));
System.out.println("compari:"+compari);
}
}
```

3. Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide-and-conquer method works along with its time complexity analysis: worst case,average case and best case.

```
package DAALAB;
import java.util.Random;
import java.util.Scanner;
public class MergeSort {
static int a[];
static void mergesort(int low, int high){
if(low<high) {
int mid=(low+high)/2;
mergesort(low,mid);
mergesort(mid+1,high);
merge(low,mid,high);
}
}
static void merge(int low, int mid, int high) {
int n=high-low+1;
int[]temp_arr=new int[n];
int i=low, j=mid+1, k=0;
while((i<=mid) && (j<=high)) {
if(a[i]<=a[j]) {
temp_arr[k]=a[i];
i++;
}
else {
temp_arr[k]=a[j];
j++;
}
k++;
}
while(i<=mid) {
temp_arr[k]=a[i];
i++;
k++;
}
while(j<=high) {
temp_arr[k]=a[j];
j++;
k++;
}
for(k=0;k<n;k++) {
a[low+k]=temp_arr[k];
}
}
public static void main(String[] args) {
// TODO Auto-generated method stub
```

```
int n;
Scanner in = new Scanner(System.in);
System.out.println("Enter a value");
n=in.nextInt();
a=new int[n];
int ch;
System.out.println("Merge Sort");
System.out.println("1. Best case");
System.out.println("2. Average case");
System.out.println("3. Worst case");
ch=in.nextInt();
System.out.println("Array before sorting");
switch(ch) {
case 1: for(int i=0;i<n;i++) {
a[i]=i+1;
System.out.print("\t"+a[i]);
}
break;
case 2: Random random=new Random();
for(int i = 0;i<n;i++) {
a[i]=random.nextInt(100);
System.out.print("\t"+a[i]);
}
break;
case 3: for(int i=0;i<n;i++) {
a[i]=n-i;
System.out.print("\t"+a[i]);
}
break;
}
long start=System.currentTimeMillis();
mergesort(0,n-1);
long end=System.currentTimeMillis();
long timeTaken=end-start;
System.out.println("\nSorted Array");
for(int i=0;i<n;i++) {
System.out.print("\t"+a[i]);
}
System.out.println("\nTime taken: " +timeTaken + " ms");
in.close();
}
}
```

Program 4

Write & Execute Java Program. To solve Knapsack problem using Greedy method.

```
package p;

import java.util.Scanner;

public class lab4

{

public static void main(String[] args)

{

int i,j=0,max_qty,m,n;

float sum=0,max;

Scanner sc = new Scanner(System.in);

int array[][]=new int[2][20];

System.out.println("Enter no of items");

n=sc.nextInt();

System.out.println("Enter the weights of each items");

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

array[0][i]=sc.nextInt();

System.out.println("Enter the values of each items");

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

array[1][i]=sc.nextInt();

System.out.println("Enter maximum volume of knapsack :");

max_qty=sc.nextInt();

m=max_qty;

while(m>=0)

{

max=0;

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

{
```

```
if(((float)array[1][i])/((float)array[0][i])>max)

{

max=((float)array[1][i])/((float)array[0][i]);

j=i;

}

}

if(array[0][j]>m)

{

System.out.println("Quantity of item number: "+ (j+1) + " added is " +m);

sum+=m*max;

m=-1;

}

else

{

System.out.println("Quantity of item number: " + (j+1) + " added is " +
array[0][j]);

m-=array[0][j];

sum+=(float)array[1][j];

array[1][j]=0;

}

}

System.out.println("The total profit is " + sum);

sc.close();

}

}
```

********* KNAPSACK PROBLEM *******
Enter the total number of items:
5
Enter the weight of each item:
5
10
20
30
40
Enter the profit of each item:
30
20
10
90
160
Enter the knapsack capacity:
60
Information about knapsack problem are

ITEM	WEIGHT	PROFIT	RATIO(PROFIT/WEIGHT)
1	5.0	30.0	6.0
2	10.0	20.0	2.0
3	20.0	10.0	0.5
4	30.0	90.0	3.0
5	40.0	160.0	4.0

Capacity = 60
Details after sorting items based on Profit/Weight ratio in descending order:

ITEM	WEIGHT	PROFIT	RATIO (PROFIT/WEIGHT)
1	5.0	30.0	6.0
2	40.0	160.0	4.0
3	30.0	90.0	3.0
4	10.0	20.0	2.0
5	20.0	10.0	0.5

The result is =
1.0 1.0 0.5 0.0 0.0
Maximum profit is = 235.0
****

```
Program 5
```

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program in Java.

```
package p;

import java.util.Scanner;

public class Diji {

	public static void main(String[] args) {

	Scanner scan = new Scanner(System.in);

	System.out.println("Enter Number of Vertices");

	int n = scan.nextInt();

	int adj[][] = new int[n][n];

	System.out.println("Enter Adjacency Matrix");

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

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

	adj[i][j] = scan.nextInt();

	}

	}

	System.out.println("Enter Source vertex");

	int src = scan.nextInt();

	int[] dist = dijkstra(adj, src);

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

	if ((src - 1) == i) {

	continue;

	}

	System.out.println("Shortest Distance from " + src + " to " + (i + 1) +
" is " + dist[i]);

	}

	scan.close();

	}
```

```java
static int[] dijkstra(int adj[][], int src) {
int n = adj.length;
int[] dist = new int[n];
boolean[] visited = new boolean[n];
int min_dist, unvis = -1;
for (int i = 0; i < n; i++) {
dist[i] = adj[src - 1][i];
visited[i] = false;
}
visited[src - 1] = true;
for (int i = 1; i < n; i++) {
min_dist = Integer.MAX_VALUE;
for (int j = 0; j < n; j++) {
if (!visited[j] && dist[j] < min_dist) {
unvis = j;
min_dist = dist[j];
}
}
visited[unvis] = true;
for (int v = 0; v < n; v++) {
if (!visited[v] && dist[unvis] + adj[unvis][v] < dist[v]) {
dist[v] = dist[unvis] + adj[unvis][v];
}
}
}
return dist;
}
}
```

Output

```
Enter Number of Vertices
5
Enter Adjacency Matrix
0 3 99 7 99
3 0 4 2 99
99 4 0 5 6
5 2 0 4 4
99 99 6 4 0
Enter Source vertex
1
Shortest Distance from 1 to 2 is 3
Shortest Distance from 1 to 3 is 5
Shortest Distance from 1 to 4 is 5
Shortest Distance from 1 to 5 is 9
```

Program 6

Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's Algorithm. Use Union-Find algorithms in your program.

```
package p;

import java.util.Arrays;

import java.util.Scanner;

class Edge {

int src;

int dest;

int weight;

Edge(int src, int dest, int weight) {

this.src = src;

this.dest = dest;

this.weight = weight;

}

}

class Kruskal {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Enter number of Vertices");

int n = scan.nextInt();

int adj[][] = new int[n][n];

System.out.println("Enter Adjacency Matrix");

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

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

adj[i][j] = scan.nextInt();

}
```

```
}
scan.close();
// Maximum Edges without any Loops can be ((n * (n - 1)) / 2).
Edge[] edges = new Edge[(n * (n - 1)) / 2];
int k = 0;
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
edges[k] = new Edge(i, j, adj[i][j]);
k++;
}
}
sort(edges);
// Declare an array of size vertices to keep track of respective leaders of
each element.
int[] parent = new int[n];
// Assign each element of array of with value -1.
Arrays.fill(parent, -1);
int minCost = 0;
System.out.println("Edges: ");
for (int i = 0; i < k; i++) {
// Find the super most of leader of source vertex.
int lsrc = find(parent, edges[i].src);
// Find the super most of leader of destination vertex.
int ldest = find(parent, edges[i].dest);
// If those two leaders are different then they belong to isolated groups.
if (lsrc != ldest) {
System.out.println((edges[i].src + 1) + " <-> " + (edges[i].dest + 1));
minCost += edges[i].weight;
union(parent, lsrc, ldest);
```

```java
}
}
System.out.println();
System.out.println("Minimum Cost of Spanning Tree: " + minCost);
}
static void sort(Edge[] edges) {
// Sort Edges according to their weights using Bubble Sort.
for (int i = 1; i < edges.length; i++) {
for (int j = 0; j < edges.length - i; j++) {
if (edges[j].weight > edges[j + 1].weight) {
Edge temp = edges[j];
edges[j] = edges[j + 1];
edges[j + 1] = temp;
}
}
}
}
static int find(int[] parent, int i) {
if (parent[i] == -1) {
// Super Most Leader Element Found.
return i;
}
// Find Above Leader in recurrsive manner.
return find(parent, parent[i]);
}
static void union(int[] parent, int lsrc, int ldest) {
// Make destination vertex leader of source vertex.
parent[lsrc] = ldest;
```

```
}
}
```

Output

```
Enter number of Vertices
6
Enter Adjacency Matrix
0 3 99 99 6 5
3 0 1 99 99 4
99 1 0 6 99 4
99 99 6 0 8 5
6 99 99 8 0 2
5 4 5 2 2 0
Edges:
2 <-> 3
5 <-> 6
1 <-> 2
2 <-> 6
4 <-> 6
Minimum Cost of Spanning Tree: 15
```

Program 7

Find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm

```
package p;

import java.util.Scanner;

class Prim {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Enter Number of Vertices");

int n = scan.nextInt();

int[][] costMatrix = new int[n][n];

boolean[] visited = new boolean[n];

System.out.println("Enter Cost Adjacency Matrix");

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

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

costMatrix[i][j] = scan.nextInt();

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

visited[i] = false;

System.out.println("Enter Source Vertex");

int srcVertex = scan.nextInt();

scan.close();

visited[srcVertex - 1] = true;

int source = 0, cost = 0, target = 0;

System.out.print("Edges: ");

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

int min = Integer.MAX_VALUE;

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

if (visited[j]) {
```

```
for (int k = 0; k < n; k++) {
if (!visited[k] && min > costMatrix[j][k]) {
min = costMatrix[j][k];
source = j;
target = k;
}
}
}
}
visited[target] = true;
System.out.print("(" + (source + 1) + "," + (target + 1) + ")");
cost += min;
}
System.out.println("\nMinimum cost of Spanning Tree: " + cost);
}
}
```

Output

```
Enter Number of Vertices
6
Enter Cost Adjacency Matrix
0 3 99 99 6 5
3 0 1 99 99 4
99 1 0 6 99 4
99 99 6 0 8 5
6 99 99 8 0 2
6 4 5 2 2 0
```

```
Enter Source Vertex

1

Edges: (1,2)(2,3)(2,6)(6,4)(6,5)

Minimum cost of Spanning Tree: 12
```

Program 8

Write Java programs toImplement All-Pairs Shortest Paths problem using Floyd's algorithm.

```
package p;

import java.util.Scanner;

class Floyd {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Enter Number of Vertices");

int n = scan.nextInt();

int[][] D = new int[10][10];

System.out.println("Enter Distance Matrix");

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

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

D[i][j] = scan.nextInt();

}

}

scan.close();

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

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

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

D[i][j] = Math.min(D[i][j], D[i][k] + D[k][j]);

}

}

}

System.out.println("Shortest Distance Matrix");

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

for (int j = 1; j <= n; j++) {
```

```
System.out.print(D[i][j] + " ");

}

System.out.println();

}

}

}
```

Output

```
Enter Number of Vertices

4

Enter Distance Matrix

0 99 3 99

2 0 99 99

99 7 0 7

6 99 99 0

Shortest Distance Matrix

0 10 3 10

2 0 5 12

9 7 0 7

6 16 9 0
```

Program 9

Solve Travelling Sales Person problem using Dynamic programming

```
package jj;

import java.util.Scanner;

public class TravSalesPerson

{

static int MAX = 100;

static final int infinity = 999;

public static void main(String args[])

{

int cost = infinity;

int c[][] = new int[MAX][MAX]; // cost matrix

int tour[] = new int[MAX]; // optimal tour

int n; // max. cities

System.out.println("Travelling Salesman Problem using Dynamic Programming\n");

System.out.println("Enter number of cities: ");

Scanner scanner = new Scanner(System.in);

n = scanner.nextInt();

System.out.println("Enter Cost matrix:\n");

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

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

{

c[i][j] = scanner.nextInt();

if (c[i][j] == 0)

c[i][j] = 999;

}

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

tour[i] = i;
```

```
cost = tspdp(c, tour, 0, n);
// print tour cost and tour
System.out.println("Minimum Tour Cost: " + cost);
System.out.println("\nTour:");
for (int i = 0; i < n; i++)
{
System.out.print(tour[i] + " -> ");
}
System.out.println(tour[0] + "\n");
scanner.close();
}
static int tspdp(int c[][], int tour[], int start, int n)
{
int i, j, k;
int temp[] = new int[MAX];
int mintour[] = new int[MAX];
int mincost;
int cost;
if (start == n - 2)
return c[tour[n - 2]][tour[n - 1]] + c[tour[n - 1]][0];
mincost = infinity;
for (i = start + 1; i < n; i++)
{
for (j = 0; j < n; j++)
temp[j] = tour[j];
temp[start + 1] = tour[i];
temp[i] = tour[start + 1];
if (c[tour[start]][tour[i]] + (cost = tspdp(c, temp, start + 1, n)) < mincost)
{
```

```
mincost = c[tour[start]][tour[i]] + cost;
for (k = 0; k < n; k++)
mintour[k] = temp[k];
}
}
for (i = 0; i < n; i++)
tour[i] = mintour[i];
return mincost;
}
}
```

```
Output
Travelling Salesman Problem using Dynamic Programming
Enter number of cities:
4
Enter Cost matrix:
0 1 2 4
1 0 999 1
2 999 0 1
4 1 1 0
Minimum Tour Cost: 5
Tour:
0 -> 1 -> 3 -> 2 -> 0
```

Program 10

Implement in Java, the 0/1 Knapsack problem using Dynamic Programming method.

```
package jj;
import java.util.Scanner;
class Knapsack {
int[] weight, profit;
int capacity, n;
Knapsack() {      //constructor automatically called
Scanner scan = new Scanner(System.in);
System.out.println("Enter Number of Items");
n = scan.nextInt();
weight = new int[n];
profit = new int[n];
System.out.println("Enter Weights of Items");
for (int i = 0; i < n; i++) {
weight[i] = scan.nextInt();
}
System.out.println("Enter Profits of Items");
for (int i = 0; i < n; i++) {
profit[i] = scan.nextInt();
}
System.out.println("Enter Capacity of Knapsack");
capacity = scan.nextInt();
scan.close();
}
void fill() {
int[][] K = new int[n + 1][capacity + 1];
```

```
for (int i = 0; i <= n; i++) {
for (int j = 0; j <= capacity; j++) {
if (i == 0 || j == 0) {
K[i][j] = 0;
} else if (j < weight[i - 1]) {
K[i][j] = K[i - 1][j];
} else {
K[i][j] = Math.max(K[i - 1][j], profit[i - 1] + K[i - 1][j - weight[i - 1]]);
}
}
}
System.out.println("Maximum Profit: " + (K[n][capacity]));
System.out.print("Items Considered: ");
int i = n, j = capacity;
while (i > 0 && j > 0) {
if (K[i][j] != K[i - 1][j]) {
System.out.print(i + " ");
j -= weight[i - 1];
}
i -= 1;
}
System.out.println();
}
public static void main(String[] args) {
Knapsack knapsack = new Knapsack();
knapsack.fill();
}
}
```

Program 11

Find a subset of a given set S = {sl,s2,. ...,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions{1,2,6}and{1,8}.A suitable message is to be displayed if the given problem instance doesn't have a solution.

```
package p;

import java.util.Scanner;

class Subset {

static int[] arr;

static int count;

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Enter n value");

int n = scan.nextInt();

arr = new int[n];

System.out.println("Enter Elements of Set");

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

arr[i] = scan.nextInt();

}

System.out.println("Enter Total Sum value");

int total = scan.nextInt();

scan.close();

subSet(total, n - 1, new boolean[n]);

if (count == 0) {

System.out.println("No solution");

}

}

static void subSet(int total, int index, boolean[] solution) {

if (total == 0) {

printSolution(solution);
```

```
} else if (total < 0 || index < 0) {
return;
} else {
boolean[] tempSolution = solution.clone();
tempSolution[index] = false;
subSet(total, index - 1, tempSolution);
tempSolution[index] = true;
subSet(total - arr[index], index - 1, tempSolution);
}
}
static void printSolution(boolean[] solution) {
count += 1;
System.out.print("Solution: ");
for (int i = 0; i < solution.length; i++) {
if (solution[i]) {
System.out.print(arr[i] + " ");
}
}
System.out.println();
}
}
```

Output

```
Enter n value
5
Enter Elements of Set
1 2 5 6 8
```

```
Enter Total Sum value
9
Solution: 1 2 6
Solution: 1 8
```

Program 12

Design and implement in Java to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle

```
package jj;

import java.util.Scanner;

class Hamiltonian {

static int[][] graph;

static int[] soln;

static int n, count = 0;

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.println("Enter Number of Vertices");

n = scan.nextInt();

// Read Adjacency Matrix in Graph array(1 Indexed)

graph = new int[n + 1][n + 1];

System.out.println("Enter Adjacency Matrix");

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

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

graph[i][j] = scan.nextInt();

}

}

scan.close();

// Instatiate Solution array(1 Indexed), (Default Value is 0)

soln = new int[n + 1];

System.out.println("Hamiltonian Cycle are");

// In a cycle source vertex doesn't matter  // Assign Starting Point to prevent repetitions

soln[1] = 1;

// Call Hamiltonian function to start backtracking from vertex 2
```

```
hamiltonian(2);
if (count == 0) {
System.out.println("No Hamiltonian Cycle");
}
}
static void hamiltonian(int k) {
while (true) {
nextValue(k);
// No next vertex so return
if (soln[k] == 0) {
return;
}
// if cycle is complete then print it else find next vertex
if (k == n) {
printArray();
} else {
hamiltonian(k + 1);
}
}
}
static void nextValue(int k) {
// Finds next feasible value
while (true) {
soln[k] = (soln[k] + 1) % (n + 1);
// If no next vertex is left, then return
if (soln[k] == 0) {
return;
}
```

```
// If there exists an edge
if (graph[soln[k - 1]][soln[k]] != 0) {
int j;
// Check if the vertex is not repeated
for (j = 1; j < k; j++) {
if (soln[j] == soln[k]) {
break;
}
}
// If vertex is not repeated
if (j == k) {
// If the vertex is not the last vertex or it completes the cycle then return
if (k < n || (k == n && graph[soln[n]][soln[1]] != 0)) {
return;
}
}
}
}
}
static void printArray() {
count += 1;
// Print Solution Array
for (int i = 1; i <= n; i++) {
System.out.print(soln[i] + " ");
}
System.out.println(soln[1]);
}
}
```

```
Output

Enter Number of Vertices

6

Enter Adjacency Matrix

0 1 1 1 0 0

1 0 1 0 0 1

1 1 0 1 1 0

1 0 1 0 1 0

0 0 1 1 0 1

0 1 0 0 1 0

Hamiltonian Cycle are

1 2 6 5 3 4 1

1 2 6 5 4 3 1

1 3 2 6 5 4 1

1 3 4 5 6 2 1

1 4 3 5 6 2 1

1 4 5 6 2 3 1
```