INDEX

Ex No:1 **Date:**	**Implement GCD using Euclidian algorithm**

AIM:

Write a program to find GCD of two and three numbers using Euclidean algorithm

ALGORITHM:

```
Step 1:Initialize two variables x and y to a and b respectively.
Step 2:If y is equal to 0, return x as the GCD.
Step 3:Compute the remainder r when x is divided by y.
Step 4:Set x to y and y to r.
Step 5:Repeat steps 2 to 4 until y is equal to 0.
Step 6:Return x as the GCD.
```

PROGRAM:

```
#include <stdio.h> intgcd(int
a, int b)
{   if (a == 0)
return b;  return
gcd(b%a, a);
}  int
main()
{  int a = 10, b = 15;  printf("GCD(%d, %d) =
%dn", a, b, gcd(a, b));  a = 35, b = 10;
printf("GCD(%d, %d) = %dn", a, b, gcd(a, b));
a = 31, b = 2;  printf("GCD(%d, %d) = %dn",
a, b, gcd(a, b));  return 0;
}
```

OUTPUT:

```
GCD(10, 15) = 5

GCD(35, 10) = 5

GCD(31, 2) = 1
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:2 **Date:**	**Implement Towers of Hanoi problem and analyze it**

AIM:

Write a program to find Towers of Hanoi problem and analyze it

ALGORITHM:

```
Step 1:If n == 1, move disk from A to C and return.
Step 2:Recursively move the top n-1 disks from A to B using C as the auxiliary peg.
        a. Call the function recursively with inputs n-1, A, B, C.
Step 3:Move the nth disk from A to C.
Step 4:Recursively move the n-1 disks from B to C using A as the auxiliary peg.
        a. Call the function recursively with inputs n-1, B, C, A. Step
5:Return the list of moves.
```

PROGRAM:

```
#include <stdio.h> voidtowers_of_hanoi(int n, char source, char
destination, char auxiliary) { if (n == 1) { printf("Move disk 1 from %c to
%c\n", source, destination); return;
   }
towers_of_hanoi(n-1, source, auxiliary, destination); printf("Move
disk %d from %c to %c\n", n, source, destination);
towers_of_hanoi(n-1, auxiliary, destination, source);
} int main() { int n; printf("Enter the
number of disks: "); scanf("%d",
&n); towers_of_hanoi(n, 'A', 'C',
'B'); return 0;}
```

OUTPUT:

```
Enter the number of disks: 3
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:3 Date:	**Sorting mechanism using quick sort**

AIM:

Write a program to find the sorting mechanism which uses the pivot value as the key component to sort all the values. Write algorithm and derive the time complexity. Sort the following using the same method:45, 67,12,34,09.

ALGORITHM:

```
Step 1:Choose a pivot value from the array (e.g., the last element).
Step 2:Iterate over the array from left to right, maintaining an index of the last
element that is less than the pivot (initialized to -1).
Step 3:If an element is less than the pivot, swap it with the element at the index of the
last element that is less than the pivot, and increment that index.
Step 4:Swap the pivot element with the element at the index of the last element that is
less than the pivot plus 1. This puts the pivot in its final sorted position.
Step 5:Recursively apply the above steps to the subarrays to the left and right of the
pivot until the subarrays are of length 0 or 1, which are already sorted by definition.
```

PROGRAM:

```
#include <stdio.h>


void swap(int* a, int* b) {

int temp = *a;     *a = *b;

   *b = temp;

}


int partition(intarr[], int low, int high) {

int pivot = arr[high]; inti = (low - 1);


for (int j = low; j <= high - 1; j++) {
```

```
if (arr[j] < pivot) {
i++; swap(&arr[i],
&arr[j]);
        }
    }
swap(&arr[i + 1], &arr[high]); return
(i + 1);
}

void quicksort(intarr[], int low, int high) {
if (low < high) { int pi = partition(arr,
low, high); quicksort(arr, low, pi - 1);
quicksort(arr, pi + 1, high);
    }
}

voidprint_array(intarr[], int n) { for
(inti = 0; i< n; i++) { printf("%d ",
arr[i]);
    } printf("\n");
}

int main() { intarr[] = {45, 67,
12, 34, 9}; int n = sizeof(arr) /
sizeof(arr[0]); printf("Original
array: "); print_array(arr, n);
quicksort(arr, 0, n - 1);
printf("Sorted array: ");
print_array(arr, n); return
0;
```

```
}
```

OUTPUT:

```
Original array: 45 67 12 34 9
Sorted array: 9 12 34 45 67
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:4 **Date:**	**Sorting mechanism using merge sort**

AIM:

Write a program to find the sorting mechanism which exactly divides the given problem into two proper subsets during the iteration. Write the algorithm and derive the time complexity. Sort the following using the same method:45,67,12,34,09

ALGORITHM:

```
Step 1:Divide the unsorted array into two halves, recursively, until each subset has
only one element.
Step 2:Merge the two halves back together by comparing the smallest elements of
each half and placing them in order into a new array.
Step 3:Continue merging the halves, in the same manner, until there is only one array
left.
```

PROGRAM:

```
#include <stdio.h>


void merge(intarr[], int l, int m, int r) {

inti, j, k; int n1 = m - l + 1; int n2 = r -

m;


int L[n1], R[n2];


for (i = 0; i< n1; i++)

L[i] = arr[l + i]; for (j =

0; j < n2; j++)

    R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;
while (i< n1 && j < n2) {
if (L[i] <= R[j]) { arr[k]
= L[i]; i++;
        }
else { arr[k]
= R[j]; j++;
        }
k++;
    }

while (i< n1) {
arr[k] = L[i];
i++; k++;
    }

while (j < n2) {
arr[k] = R[j];
j++; k++;
    }
}

voidmergeSort(intarr[], int l, int r) { if (l < r) { int m = l + (r - l) / 2;

mergeSort(arr, l, m); mergeSort(arr,
m + 1, r);

merge(arr, l, m, r);
    }
}
```

```
int main() { intarr[] = {45, 67,
12, 34, 9}; int n = sizeof(arr) /
sizeof(arr[0]);

printf("Given array is \n"); for
(inti = 0; i< n; i++)
printf("%d ", arr[i]);

mergeSort(arr, 0, n - 1);

printf("\nSorted array is \n"); for
(inti = 0; i< n; i++) printf("%d
", arr[i]);

return 0; }
```

OUTPUT

:

```
Given array is
45 67 12 34 9
Sorted array is
9 12 34 45 67
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:5 **Date:**	**Searching algorithm using binary search**

AIM:

Write a program to reduces the searching process by half based on the root node value in the constructed tree. Write the algorithm and time complexity. Construct the tree for applying the above algorithm using the same properties:55,63,31,17,22,40, 67,83.

ALGORITHM:

```
Step 1:Start by creating a binary search tree with the given elements.
Step 2:Traverse the tree from the root node.
Step 3:If the value at the root node is greater than the value to be searched, move to the left subtree.
Step 4:If the value at the root node is smaller than the value to be searched, move to the right subtree.
Step 5:If the value to be searched matches the value at the root node, return the node containing the value.
Step 6:Repeat steps 3-5 until either the node containing the value is found or the end of the tree is reached.
```

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


struct node { int data;

struct node *left, *right;

};


struct node* create_node(int value) { struct node* new_node =

(struct node*) malloc(sizeof(struct node)); new_node->data =

value;
```

```
new_node->left = NULL; new_node->right
= NULL; returnnew_node;
}

struct node* insert(struct node* node, int value) {
if (node == NULL) return create_node(value); if
(value < node->data) node->left = insert(node-
>left, value); else if (value > node->data) node-
>right = insert(node->right, value); return node;
}

struct node* search(struct node* node, int value) {
if (node == NULL || node->data == value) return
node; if (node->data > value) return search(node-
>left, value); return search(node->right, value);
}

int main() { struct node* root = NULL;
intarr[] = {55, 63, 31, 17, 22, 40, 67, 83};
int n = sizeof(arr)/sizeof(arr[0]); int value
= 40;

for (inti=0; i<n; i++)
root = insert(root, arr[i]);

struct node* result = search(root, value);
if (result == NULL) printf("Value not
found in tree\n"); else printf("Value
found in tree\n");
```

```
return 0;
}
```

OUTPUT:

```
Enter the value : 40
Value found in tree
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:6 Date:	**Minimum Spanning tree using prims algorithm**

AIM:

Write a program give an optimal solution always in finding minimum spanning tree. Write an algorithm and time complexity.

ALGORITHM:

Step 1:Initialize the minimum cost as 0 and the visited array as all 0's.
Step 2:Mark the first node as visited.
Step 3:Repeat the following until all nodes are visited:

a. Find the edge with the minimum weight among all the edges that connect a visited node and an unvisited node.
b. Mark the unvisited node as visited.
c. Add the weight of the edge found in step (a) to the minimum cost. Step 4:Output the minimum cost..

PROGRAM:

```
#include<stdio.h> inta,b,u,v,n,i,j,ne=1; int
visited[10]={0},min,mincost=0,cost[10][10];
void main()
{ printf("\n Enter the number ofnodes:");
scanf("%d",&n); printf("\n Enter the adjacency
matrix:\n"); for(i=1;i<=n;i++) for(j=1;j<=n;j++){
scanf("%d",&cost[i][j]); if(cost[i][j]==0) cost[i][j]=999;
} visited[1]=1;
printf("\n");
while(ne<n) {
for(i=1,min=9
99;i<=n;i++)
for(j=1;j<=n;j
++)
if(cost[i][j]<m
in)
if(visited[i]!=
0)
{ min=cost[i][j]; a=u=i; b=v=j;
} if(visited[u]==0 ||
visited[v]==0)
{ printf("\n Edge %d:(%d %d)cost:%d",ne++,a,b,min);
mincost+=min; visited[b]=1; } cost[a][b]=cost[b][a]=999;
} printf("\n Minimun
cost=%d",mincost);
}
```

OUTPUT:

```
Enter no of nodes : 4
Enter the Adjcency matrix
1 4 5 6
6 1 4 5
5 4 6 1
4 5 1 6
Edge 1(1, 2) cost 4

Edge 2(2,3) cost 4


Edge 3(3,4)cost 1
Minimum cost 9
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:7 **Date:**	**Huffman Tree**

AIM:

User wants to send his data in secured manner from one place to another place through communication channel. His encoding mechanism should support variable length encoding mechanism. Write a program for this situation to solve the user problem and write the time complexity using Huffman tree

ALGORITHM:

Step 1: Compute the frequency of each character in the data.
Step 2:Create a leaf node for each character and assign its frequency as its weight.
Step 3:Create a priority queue of nodes ordered by their weights.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_TREE_HT 50


struct MinHNode {

  char item;   unsigned freq;

struct MinHNode *left, *right;

};


struct MinHeap {

unsigned size;   unsigned

capacity;   struct

MinHNode **array;

};
```

```
// Create nodes struct MinHNode *newNode(char item, unsigned freq) {   struct
MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
temp->item = item;   temp->freq =
freq;

  return temp;
}

// Create min heap struct MinHeap *createMinH(unsigned capacity) {   struct
MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct
MinHNode *));   return minHeap;
}

// Function to swap void swapMinHNode(struct MinHNode **a,
struct MinHNode **b) {   struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}

// Heapify void minHeapify(struct MinHeap
*minHeap, int idx) {   int smallest = idx;   int left = 2
* idx + 1;   int right = 2 * idx + 2;
```

```
  if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]-
>freq)
smallest = left;

  if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]-
>freq)     smallest
= right;

  if (smallest != idx) {     swapMinHNode(&minHeap->array[smallest],
&minHeap->array[idx]);     minHeapify(minHeap, smallest);
  }
}

// Check if size if 1 int checkSizeOne(struct
MinHeap *minHeap) {   return (minHeap->size
== 1);
}

// Extract min struct MinHNode *extractMin(struct
MinHeap *minHeap) {   struct MinHNode *temp =
minHeap->array[0];   minHeap->array[0] = minHeap-
>array[minHeap->size - 1];

  --minHeap->size;
minHeapify(minHeap, 0);

  return temp;
}
```

```
// Insertion function void insertMinHeap(struct MinHeap *minHeap, struct
MinHNode *minHeapNode) {
  ++minHeap->size;   int i
= minHeap->size - 1;

  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {     minHeap-
>array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
int n = minHeap->size - 1;
  int i;

  for (i = (n - 1) / 2; i >= 0; --i)
minHeapify(minHeap, i);
}
int isLeaf(struct MinHNode *root) {
return !(root->left) && !(root->right);
}

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)     minHeap->array[i]
= newNode(item[i], freq[i]);
```

```
  minHeap->size = size;
buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
struct MinHNode *left, *right, *top;   struct MinHeap *minHeap =
createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(minHeap)) {
left = extractMin(minHeap);     right
= extractMin(minHeap);

    top = newNode('$', left->freq + right->freq);

    top->left = left;     top-
>right = right;
insertMinHeap(minHeap,
top);

  }
  return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top) {
if (root->left) {     arr[top] = 0;     printHCodes(root->left, arr,
top + 1);
  }   if (root->right) {     arr[top] = 1;
printHCodes(root->right, arr, top + 1);
```

```
  }   if (isLeaf(root)) {     printf("
%c              |     ",     root->item);
printArray(arr, top);
  }
}

// Wrapper function void HuffmanCodes(char item[], int freq[],
int size) {   struct MinHNode *root = buildHuffmanTree(item,
freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}
// Print the array void
printArray(int arr[], int n) {
  int i;   for (i = 0; i <
n; ++i)
printf("%d", arr[i]);

  printf("\n");
}

int main() {   char arr[] = {'A',
'B', 'C', 'D'};   int freq[] = {5,
1, 6, 3};

  int size = sizeof(arr) / sizeof(arr[0]);
  printf(" Char | Huffman code ");   printf("\n--------------------
\n");

  HuffmanCodes(arr, freq, size);
}
```

OUTPUT:

```
Enter no of nodes : 4
Enter the Adjcency matrix
1 4 5 6
6 1 4 5
5 4 6 1
4 5 1 6
Edge 1(1, 2) cost 4

Edge 2(2,3) cost 4

Edge 3(3,4)cost 1
Minimum cost 9
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:8 Date:	Floyds algorithm

AIM:

Write a program to find shortest path between all the vertices. Give solution to the user using dynamic programming methodology, Write an algorithm and time complexity using Floyds algorithm

ALGORITHM:

```
Step 1: Initialize the matrix dist[][] as follows:
Step 2: For all pairs of vertices i and j:            if i == j, set dist[i][j] to 0.
if there is an edge from i to j, set dist[i][j] to the weight of that edge.
otherwise, set dist[i][j] to infinity.
Step 3:For k from 1 to n, where n is the number of vertices in the graph:
Step 4: For all pairs of vertices i and j:
          If dist[i][j] > dist[i][k] + dist[k][j], set dist[i][j] to dist[i][k] + dist[k][j].
Return the matrix dist[][].
```

PROGRAM:

```
#include<stdio.h>
#define INFINITY 9999 #define MAX 10 void
dijkstra(int G[MAX][MAX],intn,intstartnode); int
main() { int G[MAX][MAX],i,j,n,u; printf("Enter
no. of vertices:"); scanf("%d",&n); printf("\nEnter
the adjacency matrix:\n"); for(i=0;i<n;i++)
for(j=0;j<n;j++) scanf("%d",&G[i][j]);
printf("\nEnter the starting node:");
scanf("%d",&u); dijkstra(G,n,u); return 0; } void
dijkstra(int G[MAX][MAX],intn,intstartnode)
{
intcost[MAX[MAX],distance[MAX],pred[MAX];
intvisited[MAX],count,mindistance,nextnode,i,j; for(i=0;i<n;i++) for(j=0;j<n;j++) {
if(G[i][j]==0) cost[i][j]=INFINITY; else cost[i][j]=G[i][j];
} for(i=0;i<n;i++)
{
distance[i]=cost[startnode][i];
pred[i]=startnode;
visited[i]=0; }
distance[startnode]=0;
visited[startnode]=1;
count=1; while(count<n-1)
{
mindistance=INFINITY; for(i=0;i<n;i++)
if(distance[i]<mindistance&&!visited[i])
{ mindistance=distance[i]; nextnode=i; }
visited[nextnode]=1; for(i=0;i<n;i++)
if(!visited[i])
if(mindistance+cost[nextnode][i]<distance[i])
{ distance[i]=mindistance+cost[nextnode][i]pred[i]=nextnode;
} count++; } for(i=0;i<n;i++) if(i!=startnode) {
printf("\nDistance of node%d=%d",i,distance[i]);
printf("\nPath=%d",i); j=i; do { j=pred[j];
printf("<-%d",j);
}while(j!=startnode);
}
}
```

Output:

```
Enter the no.of vertices:5
Enter the adjacency matrix:
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
Enter the starting mode:0
Distance of node 1=10 Path=1<-0
Distance of node 2=50 Path=2<-3<-0
Distance of node 3=30 Path=3<-0
Distance of node 4=60
Path=4<-2<-3<-0
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:9 Date:	**Naive String Matching Algorithm**

AIM:

Write a program for Naive String Matching Algorithm

ALGORITHM:

```
Step 1: For i from 0 to n-m:
Step 2: For j from 0 to m-1:
        If T[i+j] != P[j], break out of the inner loop.
        If j == m-1, add i to the list of indices where the pattern is found.
Step 3: Return the list of indices where the pattern is found.
```

PROGRAM:

```
#include <stdio.h>

#include <string.h>


void naiveStringMatch(char* text, char* pattern) {

int n = strlen(text);     int m = strlen(pattern);

   int i, j;

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

< m; ++j) {            if (text[i+j] != pattern[j])

break;        }        if (j == m) {

printf("Pattern found at index %d\n", i);

      }

   }

}


int main() {

   char text[] = "ABABDABACDABABCABAB";

char pattern[] = "ABABCABAB";

naiveStringMatch(text, pattern);     return 0;
}
```

Output:

```
Pattern found at index 10
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:10 Date:	**Ford Fulkerson algorithm**

AIM:

Write a program for ford Fulkerson algorithm

ALGORITHM:

```
Step 1: Initialize the flow f(u,v) to 0 for all edges (u,v) in the graph.
Step 2 :While there is a path p from the source to the sink in the residual graph:
        Find the minimum residual capacity c_f(p) of the edges in the path p.
Augment the flow along path p by c_f(p), i.e., increase the flow along each edge in p
by c_f(p).
Step 3:Update the residual capacities of the edges in the residual graph based on the
new   flow.
Step 4:Return the total flow across all edges leaving the source.
```

PROGRAM:

```
#include <stdio.h>

#include <string.h>


void naiveStringMatch(char* text, char* pattern) {

int n = strlen(text);     int m = strlen(pattern);

   int i, j;

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

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

if (text[i+j] != pattern[j])

break;

       }        if (j == m) {           printf("Pattern

found at index %d\n", i);        }

   }

}
```

```
int main() {
    char text[] = "ABABDABACDABABCABAB";
char pattern[] = "ABABCABAB";
naiveStringMatch(text, pattern);     return 0;
}
```

Output:

```
Pattern found at index 10
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:11 Date:	**Implementation of knapsack problem using branch and bound.**

AIM:

Write a program to implement knapsack problem using branch and bound.

ALGORITHM:

Step 1 :Calculate the upper bound of the root node by sorting the items by their value-to-weight ratio and filling the knapsack with items in this order until it is full.

Step 2 :Initialize the queue with the root node, which represents the state where no item is selected.

Step 3 :Process nodes in the queue in decreasing order of their bound until the queue is empty.

Step 4 :For each node, check if its bound is less than or equal to the maximum value found so far. If it is, prune this part of the tree and move on to the next node.

Step 5 :If the node is a leaf node, update the maximum value found so far if the value of this node is greater than the current maximum value.

Step 6 :If the node is not a leaf node, create two child nodes by including and excluding the next item in the knapsack. Calculate their bounds and add them to the queue in decreasing order of their bound.

Step 7 :Repeat steps 3 to 6 until the queue is empty.

Step 8 :Output the maximum value found.

.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_N 1000
```

```
struct Item {
int weight;
int value;
};

struct Node {
   int level;
int weight;
int value;
float bound;
};

int n, W; struct Item
items[MAX_N]; struct Node
queue[MAX_N];
int front, rear; int
max_value;

// Function to calculate the bound of a given node
float bound(struct Node node) {     if
(node.weight >= W) {
       return 0;
   }

   float remaining_weight = W - node.weight;
float result = node.value;

   int i = node.level;
   while (i < n && remaining_weight > 0) {
if (remaining_weight >= items[i].weight) {
```

```
result += items[i].value;
remaining_weight -= items[i].weight;
        } else {              result += (remaining_weight / (float) items[i].weight) *
items[i].value;              remaining_weight = 0;
        }
i++;
    }

    return result;
}

// Function to add a node to the queue in sorted order void
add_node(struct Node node) {
    int i = rear;     while (i > front && node.bound >
queue[i - 1].bound) {          queue[i] = queue[i - 1];
        i--;
    }
    queue[i] = node;
rear++;
}

// Function to remove a node from the queue struct
Node remove_node() {
    front++;     return
queue[front - 1]; }


// Function to solve the knapsack problem using branch and bound void
knapsack() {
   // Initialize the queue with the root node     front = rear = 0;     struct
Node root_node = {0, 0, 0, bound((struct Node) {0, 0, 0, 0})};
add_node(root_node);

   // Process nodes in the queue until it is empty
while (front < rear) {         struct Node node =
remove_node();

       // If the bound of this node is less than the current maximum value, skip it
if (node.bound <= max_value) {
            continue;
       }

       // If this is a leaf node, update the maximum value if necessary
if (node.level == n) {             if (node.value > max_value) {
max_value = node.value;
            }
continue;
       }

       // Create the left child node by including the current item
       struct Node left_node = {node.level + 1, node.weight + items[node.level].weight,
node.value + items[node.level].value, bound((struct Node) {node.level + 1, node.weight +
items[node.level].weight, node.value + items[node.level].value, 0})};
add_node(left_node);

       // Create the right child node by excluding the current item
       struct Node right_node = {node.level + 1, node.weight, node.value, bound((struct Node)
{node.level + 1, node.weight, node.value, 0})};
add_node(right_node);
   }
```

```
}

int main() {     // Read input
scanf("%d%d", &n, &W);
    for (int i = 0; i < n; i++) {         scanf("%d%d",
&items[i].weight, &items[i].value);
    }

    // Solve the problem using branch and bound
max_value = 0;     knapsack();

    // Output the maximum value printf("Maximum
value: %d\n", max_value);

return 0;
}
```

Output:

```
5 10
5 10
4 40
6 30
2 50
3 35

Maximum value: 90
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully

Ex No:12 **Date:**	**Traveling Sales Person problem**

AIM:

Write a program implement any scheme to find the optimal solution for the Traveling Sales Person problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation

ALGORITHM:

```
Step 1:Create a priority queue of nodes, initially containing the root node, which
represents the state where the first city is visited.

Step 2: Initialize the best path and best cost variables to an arbitrarily large value.

Step 3: While the priority queue is not empty, perform the following steps:

    a. Dequeue the node with the smallest lower bound from the priority queue.

    b. If the lower bound of the node is greater than or equal to the best cost, prune
    this part of the search tree and move on to the next node.

    c. If all cities have been visited, update the best path and best cost if the path
    cost of the node plus the distance from the last city to the starting city is less
    than the current best cost.

    d. Otherwise, for each unvisited city, create a child node by adding the city to
    the end of the path and calculating its lower bound. Add the child node to the
    priority queue.

Step 4:Return the best path found..
```

PROGRAM:

```
#include<stdio.h> int
ary[10][10],completed[10],n,cost=0;
void takeInput() {
int i,j;
printf("Enter the number of villages: ");
scanf("%d",&n); printf("\nEnter the Cost
Matrix\n"); for(i=0;i < n;i++) {
printf("\nEnter Elements of Row: %d\n",i+1);
for( j=0;j < n;j++) scanf("%d",&ary[i][j]);
completed[i]=0; } printf("\n\nThe cost list
is:"); for( i=0;i < n;i++) { printf("\n");

for(j=0;j < n;j++) printf("\t%d",ary[i][j]);
} } void mincost(int
city)
{ int
i,ncity;
complete
d[city]=1;
printf("%
d---
>",city+1)
;
ncity=leas
t(city);
if(ncity==
999) {
```

```
ncity=0;
printf("%
d",ncity+
1);
cost+=ary
[city][ncit
y]; return;
}
mincost(n
city); }
int
least(int
c) { int
i,nc=999;
int
min=999,
kmin;
for(i=0;i
< n;i++) {
if((ary[c][
i]!=0)&&
(complete
d[i]==0))
if(ary[c][i
]+ary[i][c
] < min)
{
min=ary[i][0]+ary[c][i];
kmin=ary[c][i]; nc=i; }
```

```
} if(min!=999)
cost+=kmin; return nc;
} int main() { takeInput(); printf("\n\nThe
Path is:\n"); mincost(0);
printf("\n\nMinimum cost is %d\n ",cost);
return 0; }
```

Output:

```
Enter the number of villages: 4
Enter the Cost Matrix
Enter Elements of Row: 1
0 4 1 3
Enter Elements of Row: 2
4 0 2 1
Enter Elements of Row: 3
1 2 0 5
Enter Elements of Row: 4
3 1 5 0
The cost list is:
0 4 1 3
4 0 2 1
1 2 0 5
3 1 5 0
The Path is:
1—>3—>2—>4—>1
Minimum cost is 7
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.

Ex No:13 Date:	**Subset Program**

AIM:

Write a program` to 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}. Display a suitable message, if the given problem instance doesn't have a solution

ALGORITHM:

```
Step 1: Create a 2D Boolean table DP of size (n+1) x (T+1), initialized with False.

Step 2:  For each i from 0 to n, set DP[i][0] to True, because there is always an empty subset that adds up to 0.

Step 3: For each i from 1 to n, and each j from 1 to T, perform the following:

    a.    If S[i-1] is greater than j, set DP[i][j] to the value of DP[i-1][j], because we cannot include S[i-1] in the subset.

    b.    Otherwise, set DP[i][j] to the value of DP[i-1][j] OR DP[i-1][j-S[i-1]]. The first term represents the case where S[i-1] is not included in the subset, and the second term represents the case where S[i-1] is included in the subset.

Step 4: If DP[n][T] is True, return True, because there exists a subset that adds up to T. Otherwise, return False.
```

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))


static int total_nodes;
```

```
// prints subset found void
printSubset(int A[], int size)
{    for(int i = 0; i < size;
i++)
    {
        printf("%*d", 5, A[i]);
    }

    printf("\n");
}

// qsort compare function int comparator(const void
*pLhs, const void *pRhs)
{    int *lhs = (int
*)pLhs;    int *rhs = (int
*)pRhs;

    return *lhs > *rhs;
}

// inputs
// s           - set vector
// t           - tuplet vector
// s_size      - set size
// t_size      - tuplet size so far
// sum         - sum so far
// ite         - nodes count
// target_sum  - sum to be found void
subset_sum(int s[], int t[],
int s_size, int t_size,              int
```

```
sum, int ite,                int const
target_sum)
{
total_nodes++;

   if( target_sum == sum )
   {
      // We found sum
printSubset(t, t_size);

      // constraint check         if( ite + 1 < s_size && sum - s[ite] +
s[ite+1] <= target_sum )
      {
         // Exclude previous added item and consider next candidate
subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
      }
return;    }
else
   {
      // constraint check         if( ite < s_size && sum
+ s[ite] <= target_sum )
      {
         // generate nodes along the breadth
for( int i = ite; i < s_size; i++ )
         {
t[t_size] = s[i];
if( sum + s[i] <=
target_sum )

            {
                    // consider next level node (along depth)
subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
                }
            }
        }
    }
}

// Wrapper that prints subsets that sum to target_sum void
generateSubsets(int s[], int size, int target_sum)
{     int *tuplet_vector = (int *)malloc(size *
sizeof(int));

    int total = 0;

    // sort the set      qsort(s, size,
sizeof(int), &comparator);

    for( int i = 0; i < size; i++ )
    {         total +=
s[i];
    }

    if( s[0] <= target_sum && total >= target_sum )
    {

        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);

    }
    free(tuplet_vector);
}

int main() {     int weights[] = {15, 22, 14, 26,
32, 9, 16, 8};     int target = 53;

    int size = ARRAYSIZE(weights);

    generateSubsets(weights, size, target);

    printf("Nodes generated %d\n", total_nodes);

    return 0;
}
```

Output:

```
8    9   14   22
   8   14   15   16
  15   16   22
Nodes generated 68
```

Department of AIML		
Average Performance	**25**	
Average Record	**15**	
Average Viva	**10**	
Total	**50**	

RESULT:

The above program is compiled and executed successfully.