//EXE 1 A

PROGRAM

```
#include <stdio.h> #include <stdlib.h>
// Structure for a tree node struct Node {
int data;
struct Node *left, *right;
};
// Function to create a new node struct Node* newNode(int data) {
struct Node* node = (struct Node*)malloc(sizeof(struct Node)); node->data = data;
node->left = node->right = NULL; return node;
}
 // Inorder traversal (Left, Root, Right) void inorder(struct Node* root) {
if (root == NULL) return;
inorder(root->left); printf("%d ", root->data); inorder(root->right);
}
// Preorder traversal (Root, Left, Right) void preorder(struct Node* root) {
if (root == NULL) return;
printf("%d ", root->data); preorder(root->left); preorder(root->right);
}
// Postorder traversal (Left, Right, Root) void postorder(struct Node* root) {
if (root == NULL) return;
postorder(root->left); postorder(root->right); printf("%d ", root->data);
}
// Main function int main() {
struct Node* root = newNode(1); root->left = newNode(2);
root->right = newNode(3); root->left->left = newNode(4);
root->left->right = newNode(5);
printf("Inorder traversal: "); inorder(root);
printf("\n");
printf("Preorder traversal: "); preorder(root);
printf("\n");
printf("Postorder traversal: "); postorder(root);
printf("\n");
return 0;
}
```

SAMPLE INPUT / OUTPUT

Input (Tree structure created in code):

```
1
/ \
2  3
/ \
4  5
```

Output:

```
Inorder traversal: 4 2 5 1 3
Preorder traversal: 1 2 4 5 3
Postorder traversal: 4 5 2 3 1
```

//EXE 1 B

PROGRAM

```
#include <stdio.h>
// Recursive function to find nth Fibonacci number int fibonacci(int n) {
if (n == 0) return 0;
else if (n == 1) return 1;
else
return (fibonacci(n - 1) + fibonacci(n - 2));
}
int main() { int n, i;
printf("Enter number of terms: "); scanf("%d", &n);
printf("Fibonacci Series: "); for (i = 0; i < n; i++)
printf("%d ", fibonacci(i));
printf("\n"); return 0;
}
```

SAMPLE INPUT / OUTPUT

Input:

```
Enter number of terms: 7
```

Output:

```
Fibonacci Series: 0 1 1 2 3 5 8
```

//EXE 2 A

PROGRAM

```
#include <stdio.h> #include <stdlib.h>
#define MAX 100
// Structure for a tree node struct Node {
int data;
struct Node *left, *right;
};
// Stack structure for storing nodes struct Stack {
struct Node* arr[MAX]; int top;
};
// Function to create a new node struct Node* newNode(int data) {
struct Node* node = (struct Node*)malloc(sizeof(struct Node)); node->data = data;
node->left = node->right = NULL; return node;
}
// Stack operations
void push(struct Stack* s, struct Node* node) { s->arr[++(s->top)] = node;
}
struct Node* pop(struct Stack* s) { return s->arr[(s->top)--];
}
int isEmpty(struct Stack* s) { return s->top == -1;
}
// Iterative Inorder Traversal
void inorderIterative(struct Node* root) { struct Stack s;
s.top = -1;
struct Node* curr = root;
while (curr != NULL || !isEmpty(&s)) { while (curr != NULL) {
push(&s, curr); curr = curr->left;
}
curr = pop(&s); printf("%d ", curr->data); curr = curr->right;
}
}
// Iterative Preorder Traversal
void preorderIterative(struct Node* root) { if (root == NULL)
return;
struct Stack s; s.top = -1; push(&s, root);
while (!isEmpty(&s)) {
struct Node* node = pop(&s); printf("%d ", node->data);
if (node->right)
push(&s, node->right); if (node->left)
push(&s, node->left);
}
}
// Iterative Postorder Traversal using two stacks void postorderIterative(struct Node* root) {
if (root == NULL) return;
struct Stack s1, s2; s1.top = s2.top = -1;
push(&s1, root);
while (!isEmpty(&s1)) {
struct Node* node = pop(&s1); push(&s2, node);
if (node->left)
push(&s1, node->left); if (node->right)
push(&s1, node->right);
}
while (!isEmpty(&s2)) {
struct Node* node = pop(&s2); printf("%d ", node->data);
}
}
// Main function int main() {
struct Node* root = newNode(1); root->left = newNode(2);
root->right = newNode(3); root->left->left = newNode(4);
root->left->right = newNode(5);
printf("Inorder Traversal (Iterative): "); inorderIterative(root);
printf("\n");
printf("Preorder Traversal (Iterative): "); preorderIterative(root);
printf("\n");
printf("Postorder Traversal (Iterative): "); postorderIterative(root);
printf("\n");
return 0;
}
```
SAMPLE INPUT / OUTPUT

Input (Tree created in program):
```
1
/ \
2  3
/ \
4  5
```
Output:
```
Inorder Traversal (Iterative): 4 2 5 1 3
Preorder Traversal (Iterative): 1 2 4 5 3
Postorder Traversal (Iterative): 4 5 2 3 1
```

//EXE 2 B

PROGRAM
```
#include <stdio.h>
int main() { int n, i;
int first = 0, second = 1, next;
printf("Enter number of terms: "); scanf("%d", &n);
printf("Fibonacci Series: "); for (i = 0; i < n; i++) {
if (i <= 1) next = i;
else {
next = first + second; first = second; second = next;
}
printf("%d ", next);
}
printf("\n"); return 0;
}
```
SAMPLE INPUT / OUTPUT

Input:
```
Enter number of terms: 7
```
Output:
```
Fibonacci Series: 0 1 1 2 3 5 8
```
//EXE 3

PROGRAM
```
#include <stdio.h>
void swap(int *a, int *b) { int temp = *a;
*a = *b;
*b = temp;
}
int partition(int arr[], int low, int high) { int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; j++) { if (arr[j] < pivot) {
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]); return (i + 1);
}
void quickSort(int arr[], int low, int high) { if (low < high) {
int pi = partition(arr, low, high); quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high);
}
}
int main() { int n, i;
printf("Enter number of elements: "); scanf("%d", &n);
int arr[n];
printf("Enter %d elements: ", n); for (i = 0; i < n; i++)
scanf("%d", &arr[i]); quickSort(arr, 0, n - 1);
printf("Sorted array using Quick Sort: "); for (i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n"); return 0;
}
```
SAMPLE INPUT / OUTPUT

Input:

```
Enter number of elements: 6
Enter 6 elements: 45 23 11 89 77 98
```

Output:

```
Sorted array using Quick Sort: 11 23 45 77 89 98
```

//EXE 3 B

PROGRAM

```
#include <stdio.h>
void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}
int partition(int arr[], int low, int high) { int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; j++) { if (arr[j] < pivot) {
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]); return (i + 1);
}
void quickSort(int arr[], int low, int high) { if (low < high) {
int pi = partition(arr, low, high); quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high);
}
}
int main() { int n, i;
printf("Enter number of elements: "); scanf("%d", &n);
int arr[n];
printf("Enter %d elements: ", n); for (i = 0; i < n; i++)
scanf("%d", &arr[i]); quickSort(arr, 0, n - 1);
printf("Sorted array using Quick Sort: "); for (i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n"); return 0;
}
```

SAMPLE INPUT / OUTPUT

Input:

```
Enter number of elements: 6
Enter 6 elements: 45 23 11 89 77 98
```

Output:

```
Sorted array using Quick Sort: 11 23 45 77 89 98
```

//EXE 4

PROGRAM

```
#include <stdio.h> #include <stdlib.h>
// Structure for a node in BST struct Node {
int data;
struct Node *left, *right;
};
// Function to create a new node struct Node* createNode(int value) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); newNode->data = value;
newNode->left = newNode->right = NULL; return newNode;
}
// Function to insert a node into BST
struct Node* insert(struct Node* root, int value) { if (root == NULL)
return createNode(value);
if (value < root->data)
root->left = insert(root->left, value); else if (value > root->data)
root->right = insert(root->right, value);
return root;
}
// Inorder Traversal (Left, Root, Right) void inorder(struct Node* root) {
if (root != NULL) { inorder(root->left); printf("%d ", root->data); inorder(root->right);
}
}
// Preorder Traversal (Root, Left, Right) void preorder(struct Node* root) {
if (root != NULL) { printf("%d ", root->data); preorder(root->left); preorder(root->right);
}
}
// Postorder Traversal (Left, Right, Root) void postorder(struct Node* root) {
if (root != NULL) { postorder(root->left); postorder(root->right); printf("%d ", root->data);
}
}
// Search for a value in BST
struct Node* search(struct Node* root, int key) { if (root == NULL || root->data == key)
return root;
if (key < root->data)
return search(root->left, key); else
return search(root->right, key);
}
int main() {
struct Node* root = NULL;
int n, value, key, i;
printf("Enter number of nodes: "); scanf("%d", &n);
printf("Enter %d elements: ", n); for (i = 0; i < n; i++) {
scanf("%d", &value); root = insert(root, value);
}
printf("\nInorder Traversal: "); inorder(root); printf("\nPreorder Traversal: "); preorder(root);
printf("\nPostorder Traversal: "); postorder(root);
printf("\n\nEnter element to search: "); scanf("%d", &key);
struct Node* found = search(root, key); if (found != NULL)
printf("Element %d found in BST.\n", key); else
printf("Element %d not found in BST.\n", key);
return 0;
}
```

SAMPLE INPUT / OUTPUT

Input:

```
Enter number of nodes: 6
Enter 6 elements: 50 30 70 20 40 60
```

Output:

```
Inorder Traversal: 20 30 40 50 60 70
Preorder Traversal: 50 30 20 40 70 60
Postorder Traversal: 20 40 30 60 70 50
Enter element to search: 40 Element 40 found in BST.
```

//EXE 5

PROGRAM

```
#include <stdio.h> #include <stdlib.h>
#define RED 0
#define BLACK 1
// Structure for a node struct Node {
int data, color;
struct Node *left, *right, *parent;
};
// Function to create a new node struct Node* createNode(int data) {
struct Node* node = (struct Node*)malloc(sizeof(struct Node)); node->data = data;
node->color = RED; // New nodes are always RED initially node->left = node->right = node->parent = NULL;
return node;
}
// Helper function to perform left rotation
void rotateLeft(struct Node **root, struct Node *x) { struct Node *y = x->right;
x->right = y->left;
if (y->left != NULL) y->left->parent = x;
y->parent = x->parent; if (x->parent == NULL)
*root = y;
else if (x == x->parent->left)
 x->parent->left = y; else
x->parent->right = y; y->left = x;
x->parent = y;
}
// Helper function to perform right rotation
void rotateRight(struct Node **root, struct Node *y) { struct Node *x = y->left;
y->left = x->right;
if (x->right != NULL) x->right->parent = y;
x->parent = y->parent; if (y->parent == NULL)
*root = x;
else if (y == y->parent->left) y->parent->left = x;
else
y->parent->right = x; x->right = y;
y->parent = x;
}
// Fix Red-Black Tree properties after insertion
void fixViolation(struct Node **root, struct Node *z) {
while (z->parent != NULL && z->parent->color == RED) { struct Node *grandparent = z->parent->parent;
if (z->parent == grandparent->left) {
struct Node *uncle = grandparent->right;
if (uncle != NULL && uncle->color == RED) { z->parent->color = BLACK;
uncle->color = BLACK; grandparent->color = RED; z = grandparent;
} else {
if (z == z->parent->right) { z = z->parent; rotateLeft(root, z);
}
z->parent->color = BLACK; grandparent->color = RED; rotateRight(root, grandparent);
}
} else {
struct Node *uncle = grandparent->left;
if (uncle != NULL && uncle->color == RED) { z->parent->color = BLACK;
uncle->color = BLACK; grandparent->color = RED;
 z = grandparent;
} else {
if (z == z->parent->left) { z = z->parent; rotateRight(root, z);
}
z->parent->color = BLACK; grandparent->color = RED; rotateLeft(root, grandparent);
}
}
}
(*root)->color = BLACK;
}
// Insert a node into Red-Black Tree
void insert(struct Node **root, int data) { struct Node *z = createNode(data); struct Node *y = NULL;
struct Node *x = *root;
while (x != NULL) { y = x;
if (z->data < x->data) x = x->left;
else
x = x->right;
}
z->parent = y; if (y == NULL)
*root = z;
else if (z->data < y->data) y->left = z;
else
y->right = z;
fixViolation(root, z);
}
// Inorder traversal displaying color void inorder(struct Node *root) {
if (root == NULL) return;
inorder(root->left);
printf("%d(%s) ", root->data, (root->color == RED) ? "R" : "B"); inorder(root->right);
}
// Main function
 int main() {
struct Node *root = NULL; int n, value;
printf("Enter number of nodes: "); scanf("%d", &n);
printf("Enter %d elements: ", n); for (int i = 0; i < n; i++) {
scanf("%d", &value); insert(&root, value);
}
printf("\nInorder Traversal with Colors: "); inorder(root);
printf("\n");
return 0;
}
```
SAMPLE INPUT / OUTPUT

Input:

```
Enter number of nodes: 6
Enter 6 elements: 10 20 30 15 25 5
```
Output:
```
Inorder Traversal with Colors:
5(B) 10(B) 15(R) 20(B) 25(R) 30(B)
```
//EXE 6

PROGRAM
```
#include <stdio.h>
// Function to swap two elements void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}
// Heapify function for Max Heap void heapify(int arr[], int n, int i) {
int largest = i;
int left = 2 * i + 1; int right = 2 * i + 2;
if (left < n && arr[left] > arr[largest]) largest = left;
if (right < n && arr[right] > arr[largest]) largest = right;
if (largest != i) {
swap(&arr[i], &arr[largest]); heapify(arr, n, largest);
}
}
// Function to build a Max Heap void buildMaxHeap(int arr[], int n) {
for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
}
// Function to perform Heap Sort void heapSort(int arr[], int n) {
buildMaxHeap(arr, n);
for (int i = n - 1; i > 0; i--) {
swap(&arr[0], &arr[i]); heapify(arr, i, 0);
}
}
 // Function to print the heap void printHeap(int arr[], int n) {
for (int i = 0; i < n; ++i) printf("%d ", arr[i]);
printf("\n");
}
int main() {
int n, arr[50];
printf("Enter number of elements: "); scanf("%d", &n);
printf("Enter %d elements: ", n); for (int i = 0; i < n; i++)
scanf("%d", &arr[i]);
printf("\nOriginal array: "); printHeap(arr, n);
buildMaxHeap(arr, n); printf("Max Heap: "); printHeap(arr, n);
heapSort(arr, n);
printf("Sorted array (Heap Sort): "); printHeap(arr, n);
return 0;
}
```
SAMPLE INPUT / OUTPUT

Input:
```
Enter number of elements: 6
Enter 6 elements: 20 15 30 5 10 25
```
Output:
```
Original array: 20 15 30 5 10 25
Max Heap: 30 15 25 5 10 20
Sorted array (Heap Sort): 5 10 15 20 25 30
```
//EXE 7

PROGRAM
```
#include <stdio.h> #include <stdlib.h> #include <math.h>
typedef struct FibNode { int key;
int degree;
struct FibNode *parent; struct FibNode *child; struct FibNode *left; struct FibNode *right; int mark;
} FibNode;
typedef struct FibHeap { FibNode *min;
int n;
} FibHeap;
// Create a new node
FibNode* createNode(int key) {
```

```
FibNode* node = (FibNode*)malloc(sizeof(FibNode)); node->key = key;
node->degree = 0;
node->parent = node->child = NULL; node->left = node->right = node; node->mark = 0;
return node;
 }
// Create empty heap FibHeap* createHeap() {
FibHeap* H = (FibHeap*)malloc(sizeof(FibHeap)); H->min = NULL;
H->n = 0;
return H;
}
// Insert node
void insert(FibHeap* H, FibNode* x) { if (H->min == NULL) {
H->min = x;
} else {
// insert into root list x->left = H->min;
x->right = H->min->right; H->min->right->left = x; H->min->right = x;
if (x->key < H->min->key) { H->min = x;
}                }
H->n++;
}
// Link two trees
void fibHeapLink(FibHeap* H, FibNode* y, FibNode* x) {
// remove y from root list y->left->right = y->right; y->right->left = y->left;
// make y a child of x y->parent = x;
if (x->child == NULL) { x->child = y;
y->left = y->right = y;
} else {
y->left = x->child;
y->right = x->child->right; x->child->right->left = y; x->child->right = y;
}
x-               >degree++; y->mark = 0;
}
// Consolidate trees
void consolidate(FibHeap* H) {
int D = (int)(log(H->n)/log(2)) + 1; FibNode* A[D];
for (int i = 0; i < D; i++) A[i] = NULL;
FibNode* w = H->min; if (w == NULL) return;
 FibNode* start = w; do {
FibNode* x = w; int d = x->degree;
while (A[d] != NULL) { FibNode* y = A[d];
if (x->key > y->key) { FibNode* temp = x; x = y;
y = temp;
}
fibHeapLink(H, y, x);
A[d] = NULL;
d++;
}
A[d] = x;
w = w->right;
} while (w != start);
H->min = NULL;
for (int i = 0; i < D; i++) { if (A[i] != NULL) {
if (H->min == NULL) {
H->min = A[i];
H->min->left = H->min->right = H->min;
} else {
A[i]->left = H->min;
A[i]->right = H->min->right; H->min->right->left = A[i]; H->min->right = A[i];
if (A[i]->key < H->min->key) { H->min = A[i];
}                }                }                }                }
// Extract minimum
FibNode* extractMin(FibHeap* H) { FibNode* z = H->min;
if (z != NULL) {
if (z->child != NULL) { FibNode* c = z->child; do {
FibNode* next = c->right;
// add child to root list c->left = H->min;
c->right = H->min->right; H->min->right->left = c; H->min->right = c;
c->parent = NULL; c = next;
} while (c != z->child);
}
 // remove z
z->left->right = z->right; z->right->left = z->left; if (z == z->right) {
H->min = NULL;
} else {
H->min = z->right; consolidate(H);
}
H->n--;
}
return z;
}
// Demo
int main() {
FibHeap* H = createHeap(); insert(H, createNode(10)); insert(H, createNode(3)); insert(H, createNode(15)); insert(H, createNode(6));
printf("Min: %d\n", H->min->key); FibNode* minNode = extractMin(H);
printf("Extracted Min: %d\n", minNode->key); printf("New Min: %d\n", H->min->key);
return 0;
}
```
SAMPLE INPUT / OUTPUT

Input:

Insert: 10, 3, 15, 6, 8
Find-Min() Extract-Min()
Decrease-Key(15 → 2) Find-Min()

Output:

```
Initial Heap: [3, 6, 8, 10, 15]
Minimum Element: 3
After Extract-Min: [6, 8, 10, 15]
After Decrease-Key(15→2): [2, 6, 8, 10]
New Minimum: 2
```

//EXE 8

PROGRAM

```
#include <stdio.h> #define MAX 20
int adj[MAX][MAX]; int visited[MAX];
int n;
// Queue for BFS
int queue[MAX], front = -1, rear = -1;
void enqueue(int v) { if (rear == MAX - 1)
printf("Queue Overflow\n");
 else {
if (front == -1) front = 0;
queue[++rear] = v;
}
}
int dequeue() {
if (front == -1 || front > rear) return -1;
return queue[front++];
}
// Breadth First Search (BFS) void BFS(int start) {
int i, v;
for (i = 0; i < n; i++) visited[i] = 0;
enqueue(start); visited[start] = 1;
printf("BFS Traversal: "); while ((v = dequeue()) != -1) {
printf("%d ", v);
for (i = 0; i < n; i++) {
if (adj[v][i] == 1 && visited[i] == 0) { enqueue(i);
visited[i] = 1;
}
}
}
printf("\n");
}
// Depth First Search (DFS) void DFS(int v) {
int i;
printf("%d ", v); visited[v] = 1;
for (i = 0; i < n; i++) {
if (adj[v][i] == 1 && visited[i] == 0) DFS(i);
}
}
int main() { int i, j, start;
printf("Enter number of vertices: ");
 scanf("%d", &n);
printf("Enter adjacency matrix (%d x %d):\n", n, n); for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) { scanf("%d", &adj[i][j]);
}
}
printf("Enter starting vertex (0 to %d): ", n - 1); scanf("%d", &start);
BFS(start);
for (i = 0; i < n; i++) visited[i] = 0;
printf("DFS Traversal: "); DFS(start);
printf("\n");
return 0;
}
```

SAMPLE INPUT / OUTPUT

Input:

```
Enter number of vertices: 5 Enter adjacency matrix (5 x 5):
0 1 1 0 0
1 0 1 1 0
1 1 0 0 1
0 1 0 0 1
0 0 1 1 0
Enter starting vertex (0 to 4): 0
```

Output:

```
BFS Traversal: 0 1 2 3 4
DFS Traversal: 0 1 3 4 2
```

//EXE 9

PROGRAM

(A) Prim's Algorithm

```
#include <stdio.h> #define INF 9999
#define MAX 20
void prims(int G[MAX][MAX], int n) { int visited[MAX] = {0};
int i, j, edgeCount = 0; int minCost = 0;
visited[0] = 1; // Start from vertex 0
 printf("\nEdges in the Minimum Spanning Tree:\n"); while (edgeCount < n - 1) {
int min = INF, a = -1, b = -1; for (i = 0; i < n; i++) {
if (visited[i]) {
for (j = 0; j < n; j++) {
if (!visited[j] && G[i][j] && G[i][j] < min) { min = G[i][j];
a = i;
b = j;
}
}
}
}
if (a != -1 && b != -1) {
printf("%d -> %d cost = %d\n", a, b, G[a][b]); minCost += G[a][b];
visited[b] = 1; edgeCount++;
}
}
printf("\nMinimum Cost of Spanning Tree: %d\n", minCost);
}
int main() {
int G[MAX][MAX], n, i, j;
printf("Enter number of vertices: "); scanf("%d", &n);
printf("Enter the adjacency matrix:\n"); for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) { scanf("%d", &G[i][j]); if (G[i][j] == 0)
G[i][j] = INF;
}
}
prims(G, n); return 0;
}
```
(B) Kruskal's Algorithm

```
#include <stdio.h> #define MAX 30
 int parent[MAX];
int find(int i) { while (parent[i])
i = parent[i]; return i;
}
int uni(int i, int j) { if (i != j) {
parent[j] = i; return 1;
}
return 0;
}
int main() {
int n, i, j, a, b, u, v, edges = 1;
int min, mincost = 0, cost[MAX][MAX];
printf("Enter the number of vertices: "); scanf("%d", &n);
printf("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;
}
}
printf("\nEdges in the Minimum Spanning Tree:\n"); while (edges < n) {
min = 999;
for (i = 1; i <= n; i++) { for (j = 1; j <= n; j++) {
if (cost[i][j] < min) {
min = cost[i][j];
a = u = i;
b = v = j;
}
}
}
u = find(u); v = find(v);
if (uni(u, v)) {
printf("%d -> %d cost = %d\n", a, b, min);
 mincost += min; edges++;
}
cost[a][b] = cost[b][a] = 999;
}
printf("\nMinimum Cost of Spanning Tree: %d\n", mincost); return 0;
}
```

SAMPLE INPUT / OUTPUT

Input:

```
Enter number of vertices: 4 Enter the adjacency matrix:
0 10 6 5
10 0 0 15
6 0 0 4
5 15 4 0
```

Output:

```
(Prim's Algorithm):
Edges in the Minimum Spanning Tree:
0 -> 3 cost = 5
3 -> 2 cost = 4
0 -> 1 cost = 10
Minimum Cost of Spanning Tree: 19
Output (Kruskal's Algorithm):
Edges in the Minimum Spanning Tree:
3 -> 2 cost = 4
0 -> 3 cost = 5
0 -> 1 cost = 10
Minimum Cost of Spanning Tree: 19
```

//EXE 10 A

PROGRAM

```
#include <stdio.h> #include <limits.h>
#define V 5  // Number of vertices in the graph
// Function to find the vertex with the minimum distance value int minDistance(int dist[], int visited[]) {
int min = INT_MAX, min_index = -1; for (int v = 0; v < V; v++)
if (visited[v] == 0 && dist[v] <= min) min = dist[v], min_index = v;
```

```
return min_index;
}
// Function to print the final shortest distances void printSolution(int dist[]) { printf("Vertex\tDistance from Source\n");
for (int i = 0; i < V; i++) printf("%d\t\t%d\n", i, dist[i]); }
// Dijkstra's algorithm
void dijkstra(int graph[V][V], int src) {
int dist[V]; // The output array: shortest distance from src to i
int visited[V];  // visited[i] is true if vertex i is included in shortest path tree
 // Step 1: Initialize all distances as infinite and visited[] as false for (int i = 0; i < V; i++) {
dist[i] = INT_MAX; visited[i] = 0;
}
```

SAMPLE INPUT / OUTPUT

The adjacency matrix used in the program represents the following weighted graph:

From → To	0	1	2	3	4
0	0	10	0	30	100
1	10	0	50	0	0
2	0	50	0	20	10
3	30	0	20	0	60
4	100	0	10	60	0

OUTPUT:

```
Vertex Distance from Source
                                  0        0
                                  1        10
                                  2        50
                                  3        30
                                  4        60
```

//EXE 10 B

PROGRAM

```
#include <stdio.h> #include <stdlib.h> #include <limits.h>
// Structure to represent an edge struct Edge {
int src, dest, weight;
};
// Structure to represent a graph struct Graph {
int V, E;
struct Edge* edge;
};
// Create a graph with V vertices and E edges struct Graph* createGraph(int V, int E) {
struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph)); graph->V = V;
graph->E = E;
graph->edge = (struct Edge*) malloc(E * sizeof(struct Edge)); return graph;
}
// Function to print the solution void printArr(int dist[], int n) {
 printf("Vertex\tDistance from Source\n"); for (int i = 0; i < n; i++)
printf("%d\t\t%d\n", i, dist[i]);
}
// Bellman-Ford Algorithm
void BellmanFord(struct Graph* graph, int src) { int V = graph->V;
int E = graph->E; int dist[V];
// Step 1: Initialize distances from src to all other vertices as INFINITE for (int i = 0; i < V; i++)
dist[i] = INT_MAX; dist[src] = 0;
// Step 2: Relax all edges |V| - 1 times for (int i = 1; i <= V - 1; i++) {
for (int j = 0; j < E; j++) { int u = graph->edge[j].src;
int v = graph->edge[j].dest;
int w = graph->edge[j].weight;
if (dist[u] != INT_MAX && dist[u] + w < dist[v]) dist[v] = dist[u] + w;
}
}
// Step 3: Check for negative-weight cycles for (int j = 0; j < E; j++) {
int u = graph->edge[j].src; int v = graph->edge[j].dest;
int w = graph->edge[j].weight;
if (dist[u] != INT_MAX && dist[u] + w < dist[v]) { printf("\nGraph contains negative weight cycle!\n"); return;
}
}
// Print the shortest distances printArr(dist, V);
}
// Main function int main() {
int V = 5; // Number of vertices int E = 8; // Number of edges
 struct Graph* graph = createGraph(V, E);
// Adding edges (src, dest, weight)
graph->edge[0].src = 0; graph->edge[0].dest = 1; graph->edge[0].weight = -1;
graph->edge[1].src = 0; graph->edge[1].dest = 2; graph->edge[1].weight = 4;
graph->edge[2].src = 1; graph->edge[2].dest = 2; graph->edge[2].weight = 3;
graph->edge[3].src = 1; graph->edge[3].dest = 3; graph->edge[3].weight = 2;
graph->edge[4].src = 1; graph->edge[4].dest = 4; graph->edge[4].weight = 2;
graph->edge[5].src = 3; graph->edge[5].dest = 2; graph->edge[5].weight = 5;
graph->edge[6].src = 3; graph->edge[6].dest = 1; graph->edge[6].weight = 1;
graph->edge[7].src = 4; graph->edge[7].dest = 3; graph->edge[7].weight = -3; int source = 0;
BellmanFord(graph, source); return 0;
}
```

SAMPLE INPUT / OUTPUT

The graph edges are defined in the code as:

Edge	From	To	Weight
1	0	1	-1
2	0	2	4
3	1	2	3
4	1	3	2
5	1	4	2
6	3	2	5
7	3	1	1
8	4	3	-3

Source vertex = **0**

OUTPUT:

Vertex Distance from Source

```
0      0
1      -1
2      2
3      -2
4      1
```

//EXE 11

PROGRAM

```
#include <stdio.h> #include <limits.h>
#define MAX 100
int MatrixChainOrder(int p[], int n) { int m[MAX][MAX];
// cost is zero when multiplying one matrix for (int i = 1; i < n; i++)
 m[i][i] = 0;
// L is chain length
for (int L = 2; L < n; L++) {
for (int i = 1; i < n - L + 1; i++) { int j = i + L - 1;
m[i][j] = INT_MAX;
for (int k = i; k <= j - 1; k++) {
int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]; if (q < m[i][j])
m[i][j] = q;
}          }          }
return m[1][n - 1];
}
int main() { int n;
printf("Enter number of matrices: "); scanf("%d", &n);
int p[MAX];
printf("Enter dimensions (p0 p1 p2 ... pn): "); for (int i = 0; i <= n; i++)
scanf("%d", &p[i]);
int minCost = MatrixChainOrder(p, n + 1);
printf("\nMinimum number of multiplications is: %d\n", minCost);
return 0;
}
```

SAMPLE INPUT

```
Enter number of matrices: 6
Enter dimensions (p0 p1 p2 ... pn): 30 35 15 5 10 20 25
SAMPLE OUTPUT
Minimum number of multiplications is: 15125
```

//EXE 12 A

PROGRAM

```c
#include <stdio.h>
// Structure to represent an activity typedef struct {
int start, finish;
} Activity;
// Function to sort activities by finish time
void sortActivities(Activity activities[], int n) { Activity temp;
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (activities[j].finish > activities[j + 1].finish) { temp = activities[j];
activities[j] = activities[j + 1]; activities[j + 1] = temp;
}
}
}
}
 // Function to perform Activity Selection
void activitySelection(Activity activities[], int n) { printf("\nSelected activities are:\n");
// The first activity always gets selected int i = 0;
printf("A%d (Start: %d, Finish: %d)\n", i + 1, activities[i].start, activities[i].finish);
// Consider the rest of the activities for (int j = 1; j < n; j++) {
if (activities[j].start >= activities[i].finish) {
printf("A%d (Start: %d, Finish: %d)\n", j + 1, activities[j].start, activities[j].finish); i = j;
}
}
}
int main() { int n;
printf("Enter number of activities: "); scanf("%d", &n);
Activity activities[n];
printf("Enter start and finish times of activities:\n"); for (int i = 0; i < n; i++) {
printf("Activity %d - Start: ", i + 1); scanf("%d", &activities[i].start); printf("Activity %d - Finish: ", i + 1); scanf("%d", &activities[i].finish);
}
sortActivities(activities, n); activitySelection(activities, n);
return 0;
}
```

SAMPLE INPUT:

```
Enter number of activities: 6
Enter start and finish times of activities: Activity 1 - Start: 1
Activity 1 - Finish: 3
Activity 2 - Start: 2
Activity 2 - Finish: 5
Activity 3 - Start: 4
Activity 3 - Finish: 6
Activity 4 - Start: 6
Activity 4 - Finish: 7
Activity 5 - Start: 5
Activity 5 - Finish: 9
Activity 6 - Start: 8
Activity 6 - Finish: 10
```

SAMPLE OUTPUT:

```
Selected activities are:
A1 (Start: 1, Finish: 3)
A3 (Start: 4, Finish: 6)
A4 (Start: 6, Finish: 7)
A6 (Start: 8, Finish: 10)
```

//EXE 12 B

PROGRAM

```c
#include <stdio.h> #include <stdlib.h>
// A Huffman tree node struct MinHeapNode {
char data; unsigned freq;
struct MinHeapNode *left, *right;
```

```
};
// A Min Heap struct MinHeap {
unsigned size; unsigned capacity;
struct MinHeapNode **array;
};
 // Function to allocate a new min heap node
struct MinHeapNode* newNode(char data, unsigned freq) {
struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode)); temp->left = temp->right = NULL;
temp->data = data; temp->freq = freq; return temp;
}
// Function to create a min heap
struct MinHeap* createMinHeap(unsigned capacity) {
struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap)); minHeap->size = 0;
minHeap->capacity = capacity;
minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct MinHeapNode*));
return minHeap;
}
// Swap function
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) { struct MinHeapNode* t = *a;
*a = *b;
*b = t;
}
// Heapify function
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) {
swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]); minHeapify(minHeap, smallest);
}
}
// Extract minimum node
struct MinHeapNode* extractMin(struct MinHeap* minHeap) { struct MinHeapNode* temp = minHeap->array[0];
minHeap->array[0] = minHeap->array[minHeap->size - 1];
--minHeap->size;
 minHeapify(minHeap, 0); return temp;
}
// Insert a new node
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* 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;
}
// Create and build min heap
struct MinHeap* buildMinHeap(char data[], int freq[], int size) { struct MinHeap* minHeap = createMinHeap(size);
for (int i = 0; i < size; ++i)
minHeap->array[i] = newNode(data[i], freq[i]); minHeap->size = size;
for (int i = (minHeap->size - 2) / 2; i >= 0; --i) minHeapify(minHeap, i);
return minHeap;
}
// Build Huffman Tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) { struct MinHeapNode *left, *right, *top;
struct MinHeap* minHeap = buildMinHeap(data, freq, size);
while (minHeap->size != 1) { left = extractMin(minHeap); right = extractMin(minHeap);
top = newNode('$', left->freq + right->freq); top->left = left;
top->right = right;
insertMinHeap(minHeap, top);
}
return extractMin(minHeap);
}
// Print Huffman Codes
void printCodes(struct MinHeapNode* root, int arr[], int top) {
 if (root->left) { arr[top] = 0;
printCodes(root->left, arr, top + 1);
}
if (root->right) { arr[top] = 1;
printCodes(root->right, arr, top + 1);
}
if (!(root->left) && !(root->right)) {
printf("%c: ", root->data); for (int i = 0; i < top; i++)
printf("%d", arr[i]); printf("\n");
}
}
// Huffman Coding main function
void HuffmanCodes(char data[], int freq[], int size) {
struct MinHeapNode* root = buildHuffmanTree(data, freq, size); int arr[100], top = 0;
printCodes(root, arr, top);
}
// Main function int main() {
int n;
printf("Enter number of characters: "); scanf("%d", &n);
char data[n]; int freq[n];
for (int i = 0; i < n; i++) { printf("Enter character %d: ", i + 1); scanf(" %c", &data[i]);
printf("Enter frequency of %c: ", data[i]); scanf("%d", &freq[i]);
}
printf("\nHuffman Codes are:\n"); HuffmanCodes(data, freq, n);
return 0;
}
```

SAMPLE INPUT:

```
Enter number of characters: 6 Enter character 1: a
Enter frequency of a: 5 Enter character 2: b Enter frequency of b: 9 Enter character 3: c Enter frequency of c: 12 Enter character 4: d Enter frequency of d: 13 Enter character 5: e Enter frequency of e: 16 Enter character 6: f Enter frequency of f: 45
```

SAMPLE OUTPUT:

```
Huffman Codes are:
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```