Design And Analysis Of Algorithm

#]Slip-1

Q.1)

```
#include <stdio.h>
#include <time.h>
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    selectionSort(arr, n);
    end = clock();
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time required to sort the elements: %f seconds\n", cpu_time_used);
    return 0;
}
```

Q.2)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (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(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;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time required to sort the elements: %f seconds\n", cpu_time_used);
    return 0;
}
```

#]Slip-2

Q.1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
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);
    }
}
void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    srand(time(NULL));
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100;
    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    heapSort(arr, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Q.2)

```
#include <stdio.h>
void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {
}
int main() {
    int n = 2;
    int A[2][2] = {{1, 2}, {3, 4}};
    int B[2][2] = {{5, 6}, {7, 8}};
    int C[2][2] = {0};
    strassen(n, A, B, C);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

#]Slip-3

Q.1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.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 - 1; 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;

    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
```

```
        scanf("%d", &arr[i]);
    }
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nTime taken to sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

Q.2)

```
#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}
void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    int mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printMST(parent, graph);
```

```
}
int main() {
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    primMST(graph);
    return 0;
}
```

#]SLip-4

Q1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void merge(int arr[], int left, int middle, int right) {
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
    i = 0;
    j = 0;
    k = left;
    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++;
    }
}
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
```

```
        merge(arr, left, middle, right);
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    mergeSort(arr, 0, n - 1);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nTime taken to sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

Q2)

```
#include <stdio.h>
struct Item {
    int value;
    int weight;
};
int knapsack(int W, struct Item items[], int n) {
    int i, j;
    int dp[n + 1][W + 1];
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= W; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (items[i - 1].weight <= j) {
                dp[i][j] = max(items[i - 1].value + dp[i - 1][j - items[i -
1].weight], dp[i - 1][j]);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][W];
}
int main() {
    int W;
    printf("Enter the maximum weight of the knapsack: ");
    scanf("%d", &W);
    int n;
    printf("Enter the number of items: ");
    scanf("%d", &n);
    struct Item items[n];
    printf("Enter the values and weights of the items:\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("Item %d:\n", i + 1);
        printf("Enter value: ");
        scanf("%d", &items[i].value);
        printf("Enter weight: ");
        scanf("%d", &items[i].weight);
    }
    int maxValue = knapsack(W, items, n);
    printf("Maximum value that can be put in the knapsack: %d\n", maxValue);
    return 0;
}
```

#]SLip-5

Q1)

```
#include <stdio.h>
#include <stdlib.h>
struct Edge {
    int src, dest, weight;
};
struct Graph {
    int V, E;
    struct Edge* edge;
};
struct Subset {
    int parent;
    int rank;
};
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}
void Union(struct Subset subsets[], int x, int y) {
    int xRoot = find(subsets, x);
    int yRoot = find(subsets, y);
    if (subsets[xRoot].rank < subsets[yRoot].rank) {
        subsets[xRoot].parent = yRoot;
    } else if (subsets[xRoot].rank > subsets[yRoot].rank) {
        subsets[yRoot].parent = xRoot;
    } else {
        subsets[yRoot].parent = xRoot;
        subsets[xRoot].rank++;
    }
}
int myCompare(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}
void KruskalMST(struct Graph* graph) {
    int i, count;
    int src, dest, cost;
    struct Edge result[graph->V - 1];
    struct Subset subsets[graph->V];
    for (i = 0; i < graph->V; i++) {
```

```
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }
    i = 0;
    count = 0;
    while (count < graph->V - 1 && i < graph->E) {
        if (find(subsets, graph->edge[i].src) != find(subsets, graph->edge[i].dest))
{
            result[count].src = graph->edge[i].src;
            result[count].dest = graph->edge[i].dest;
            result[count].weight = graph->edge[i].weight;
            count++;
            Union(subsets, graph->edge[i].src, graph->edge[i].dest);
        }
        i++;
    }
    printf("Edge \tSource \tDestination \tWeight\n");
    for (i = 0; i < graph->V - 1; i++) {
        printf("%d \t%d \t%d \t%d\n", i + 1, result[i].src, result[i].dest,
result[i].weight);
    }
}
int main() {
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(graph->E * sizeof(struct Edge));
    printf("Enter the source, destination, and weight of each edge:\n");
    for (int i = 0; i < graph->E; i++) {
        scanf("%d %d %d", &src, &dest, &cost);
        graph->edge[i].src = src;
        graph->edge[i].dest = dest;
        graph->edge[i].weight = cost;
    }
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myCompare);
    KruskalMST(graph);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TREE_HT 100
struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode *left, *right;
};
struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct MinHeapNode **array;
};
```

```
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;
}
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;
}
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
}
int main() {
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr) / sizeof(arr[0]);
    struct MinHeapNode* root = buildHuffmanTree(arr, freq, size);
    return 0;
}
```

#]SLip-6

Q1)

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define V 5
int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}
void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
    key[0] = 0;
```

```
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printMST(parent, graph);
}
int main() {
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };
    primMST(graph);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <string.h>
int longestCommonSubsequenceLength(char *str1, char *str2) {
    int m = strlen(str1);
    int n = strlen(str2);
    int dp[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (str1[i - 1] == str2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = 0;
            }
        }
    }
    return dp[m][n];
}
int main() {
    char str1[] = "ABCDGH";
    char str2[] = "AEDFHR";
    int length = longestCommonSubsequenceLength(str1, str2);
    printf("Length of the longest common subsequence: %d\n", length);
    return 0;
}
```

#]SLip-7

Q1)

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define V 9
int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    printSolution(dist);
}
int main() {
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };
    dijkstra(graph, 0);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int vertex;
    struct Node* next;
};
struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* inDegree;
};
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
    graph->inDegree = (int*)malloc(vertices * sizeof(int));
    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->inDegree[i] = 0;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    graph->inDegree[dest]++;
}
void topologicalSort(struct Graph* graph) {
    int* result = (int*)malloc(graph->numVertices * sizeof(int));
    int front = 0, rear = 0;
    int* queue = (int*)malloc(graph->numVertices * sizeof(int));
    for (int i = 0; i < graph->numVertices; i++) {
        if (graph->inDegree[i] == 0) {
            queue[rear++] = i;
        }
    }

    while (front != rear) {
        int current = queue[front++];
        result[front - 1] = current;
        struct Node* temp = graph->adjLists[current];
        while (temp != NULL) {
            graph->inDegree[temp->vertex]--;
            if (graph->inDegree[temp->vertex] == 0) {
                queue[rear++] = temp->vertex;
            }
            temp = temp->next;
        }
    }
    if (front != graph->numVertices) {
```

```
        printf("The graph has a cycle!\n");
        return;
    }
    printf("Topological order: ");
    for (int i = 0; i < graph->numVertices; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");
}
int main() {
    int vertices = 6;
    struct Graph* graph = createGraph(vertices);
    addEdge(graph, 5, 2);
    addEdge(graph, 5, 0);
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);
    topologicalSort(graph);
    return 0;
}
```

#]SLip-8

Q1)

```
#include <stdio.h>
#include <stdlib.h>
struct Item {
    int value;
    int weight;
};
int compare(const void *a, const void *b) {
    double ratio1 = ((double)((struct Item *)a)->value / (double)((struct Item *)a)-
>weight);
    double ratio2 = ((double)((struct Item *)b)->value / (double)((struct Item *)b)-
>weight);
    if (ratio1 < ratio2) {
        return 1;
    } else {
        return -1;
    }
}
double fractionalKnapsack(int W, struct Item arr[], int n) {
    qsort(arr, n, sizeof(struct Item), compare);
    int currentWeight = 0;
    double finalValue = 0.0;
    for (int i = 0; i < n; i++) {
        if (currentWeight + arr[i].weight <= W) {
            currentWeight += arr[i].weight;
            finalValue += arr[i].value;
        } else {
            int remaining = W - currentWeight;
            finalValue += arr[i].value * ((double)remaining / (double)arr[i].weight);
            break;
        }
    }
    return finalValue;
}
```

```
int main() {
    int W = 50; // Knapsack capacity
    struct Item arr[] = {{60, 10}, {100, 20}, {120, 30}}; // {value, weight}
    int n = sizeof(arr) / sizeof(arr[0]);
    double maxValue = fractionalKnapsack(W, arr, n);
    printf("Maximum value that can be obtained: %.2f\n", maxValue);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#define V 4
int minDistance(int dist[], bool sptSet[]) {
    int min = __INT_MAX__, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
void printPath(int parent[], int graph[V][V]) {
    printf("Path: ");
    for (int i = 0; i < V; i++) {
        printf("%d -> ", parent[i]);
    }
    printf("0\n");
}
void nearestNeighbor(int graph[V][V]) {
    int parent[V];
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = __INT_MAX__;
        sptSet[i] = false;
    }
    dist[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != __INT_MAX__ && dist[u] +
graph[u][v] < dist[v]) {
                parent[v] = u;
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    printf("Minimum cost: %d\n", dist[V - 1]);
    printPath(parent, graph);
}
int main() {
    int graph[V][V] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
```

```
    };
    nearestNeighbor(graph);
    return 0;}
```

#]SLip-9

Q1)

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return node;
}
struct Node* insert(struct Node* root, int key) {
    if (root == NULL) {
        return newNode(key);
    }
    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;
}
int height(struct Node* node) {
    if (node == NULL) {
        return 0;
    }
    int leftHeight = height(node->left);
    int rightHeight = height(node->right);
    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
}
int getBestCaseComplexity(int n) {
    int height = (int)(log2(n + 1)) + 1;
    return (1 << height) - 1;
}
int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
    int n = 7;
    int bestCaseComplexity = getBestCaseComplexity(n);
    printf("Best-case complexity: %d\n", bestCaseComplexity);
    return 0;
}
```

Q2)

```
#include <stdio.h>
void sumOfSubset(int arr[], int n, int sum) {
    int i, subset[n];
    int currSum = 0;
    int start = 0;
    void backtrack(int pos) {
        if (currSum == sum) {
            for (int i = 0; i < pos; i++) {
                printf("%d ", subset[i]);
            }
            printf("\n");
            return;
        }
        if (currSum > sum || pos == n) {
            return;
        }
        subset[pos] = arr[pos];
        currSum += arr[pos];
        backtrack(pos + 1);
        currSum -= arr[pos];
        backtrack(pos + 1);
    }
    backtrack(start);
}
int main() {
    int arr[] = {3, 2, 4, 6, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int sum = 12;
    sumOfSubset(arr, n, sum);
    return 0;
}
```

#]SLip-10

Q1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Node {
    char data;
    int freq;
    struct Node* left;
    struct Node* right;
};
struct Node* newNode(char data, int freq) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->freq = freq;
    node->left = NULL;
    node->right = NULL;
    return node;
}
struct Node* minValueNode(struct Node** harr, int* count) {
    struct Node* min_node = NULL;
    int min = INT_MAX;
    for (int i = 0; i < *count; ++i) {
        if (harr[i]->freq < min) {
            min = harr[i]->freq;
            min_node = harr[i];
```

```
        }
    }
    return min_node;
}
void huffmanCodes(struct Node* root, char* huffman[], int* count, char* arr, int
arr_size) {
    if (!root) {
        return;
    }
    if (root->left == NULL && root->right == NULL) {
        huffman[root->data] = (char*)malloc(sizeof(char) * (*count + 1));
        strcpy(huffman[root->data], arr);
        return;
    }
    char left_arr[200];
    strcpy(left_arr, arr);
    strcat(left_arr, "0");
    char right_arr[200];
    strcpy(right_arr, arr);
    strcat(right_arr, "1");
    huffmanCodes(root->left, huffman, count, left_arr, arr_size);
    huffmanCodes(root->right, huffman, count, right_arr, arr_size);
}
void printHuffmanCodes(char* huffman[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%c: %s\n", i, huffman[i]);
    }
}
int main() {
    struct Node* harr[] = {newNode('a', 5), newNode('b', 15), newNode('c', 12),
newNode('d', 3), newNode('e', 7), newNode('f', 9)};
    int count = sizeof(harr) / sizeof(harr[0]);
    struct Node* root = harr[0];
    for (int i = 1; i < count; ++i) {
        struct Node* min1 = minValueNode(harr, &count);
        struct Node* min2 = minValueNode(harr + i, &count);
        struct Node* left = (struct Node*)malloc(sizeof(struct Node));
        left->left = min1;
        left->right = min2;
        left->freq = min1->freq + min2->freq;
        struct Node* right = (struct Node*)malloc(sizeof(struct Node));
        right->left = NULL;
        right->right = NULL;
        right->data = ' ';
        right->freq = 0;
        for (int j = 0; j < i; ++j) {
            harr[j]->freq += left->freq;
        }
        harr[i] = left;
        harr[0] = right;
        count++;
    }
    char* huffman[256];
    int count_codes = 0;
    char arr[200];
    strcpy(arr, "");
    huffmanCodes(root, huffman, &count_codes, arr, count);
    printHuffmanCodes(huffman, count_codes);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#define N 4
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }}
bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    for (i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    return true;
}
bool solveNQUtil(int board[N][N], int col) {
    if (col >= N) {
        return true;
    }
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1)) {
                return true;
            }
            board[i][col] = 0; // Backtrack
        }
    }
    return false;
}
void solveNQ() {
    int board[N][N] = { {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0} };
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return;
    }
    printSolution(board);
}
int main() {
    solveNQ();
    return 0;
}
```

#]SLip-11

Q1)

```
#include <stdio.h>
#include <time.h>
#define MAX_NODES 10000
int visited[MAX_NODES];
int adjacencyMatrix[MAX_NODES][MAX_NODES];
void initializeVisited() {
    for (int i = 0; i < MAX_NODES; i++) {
        visited[i] = 0;
    }
}
void addEdge(int graph[MAX_NODES][MAX_NODES], int u, int v) {
    graph[u][v] = 1;
    graph[v][u] = 1;
}
void depthFirstSearch(int graph[MAX_NODES][MAX_NODES], int start) {
    visited[start] = 1;
    printf("%d ", start);
    for (int v = 0; v < MAX_NODES; v++) {
        if (graph[start][v] && !visited[v]) {
            depthFirstSearch(graph, v);
        }
    }
}
int main() {
    int graph[MAX_NODES][MAX_NODES];
    initializeVisited();
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 0, 3);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);
    addEdge(graph, 4, 5);
    addEdge(graph, 5, 6);
    addEdge(graph, 6, 7);
    addEdge(graph, 7, 8);
    addEdge(graph, 8, 9);
    clock_t start = clock();
    depthFirstSearch(graph, 0);
    clock_t end = clock();
    double timeTaken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken: %.5f seconds\n", timeTaken);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX_NODES 100
struct Edge {
    int src;
    int dest;
    int weight;
};
struct Graph {
    int numVertices;
    struct Edge* edges;
};
struct Vertex {
    int vertex;
    int distance;
};
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;
    graph->edges = (struct Edge*)malloc(sizeof(struct Edge) * numVertices);
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest, int weight) {
    struct Edge edge;
    edge.src = src;
    edge.dest = dest;
    edge.weight = weight;
    graph->edges[src] = edge;
}
int minDistance(int distances[], bool visited[], int numVertices) {
    int min = INT_MAX;
    int minIndex = -1;
    for (int i = 0; i < numVertices; i++) {
        if (!visited[i] && distances[i] <= min) {
            min = distances[i];
            minIndex = i;
        }
    }
    return minIndex;
}
void dijkstra(struct Graph* graph, int src) {
    int numVertices = graph->numVertices;
    int* distances = (int*)malloc(numVertices * sizeof(int));
    bool* visited = (bool*)malloc(numVertices * sizeof(bool));
    for (int i = 0; i < numVertices; i++) {
        distances[i] = INT_MAX;
        visited[i] = false;
    }
    distances[src] = 0;
    for (int i = 0; i < numVertices; i++) {
        int u = minDistance(distances, visited, numVertices);
        visited[u] = true;
        struct Edge edge = graph->edges[u];
        while (edge.dest != -1) {
            int v = edge.dest;
            int weight = edge.weight;
            if (!visited[v] && distances[u] + weight < distances[v]) {
                distances[v] = distances[u] + weight;
```

```
            }
            edge = graph->edges[u];
        }
    }
    printf("Vertex\tDistance from Source\n");
    for (int i = 0; i < numVertices; i++) {
        printf("%d\t%d\n", i, distances[i]);
    }
    free(distances);
    free(visited);
}
int main() {
    int numVertices = 6;
    struct Graph* graph = createGraph(numVertices);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 2, 2);
    addEdge(graph, 1, 2, 1);
    addEdge(graph, 1, 3, 5);
    addEdge(graph, 2, 4, 6);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 3, 5, 3);
    addEdge(graph, 4, 5, 1);
    int src = 0;
    dijkstra(graph, src);
    return 0;
}
```

#]SLip-12]

Q1)

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#define MAX_NODES 10000
int visited[MAX_NODES];
int adjacencyMatrix[MAX_NODES][MAX_NODES];
void initializeVisited() {
    for (int i = 0; i < MAX_NODES; i++) {
        visited[i] = 0;
    }
}
void addEdge(int graph[MAX_NODES][MAX_NODES], int u, int v) {
    graph[u][v] = 1;
    graph[v][u] = 1;
}
void breadthFirstSearch(int graph[MAX_NODES][MAX_NODES], int start) {
    int queue[MAX_NODES];
    int front = 0, rear = 0;
    visited[start] = 1;
    queue[rear++] = start;
    while (front < rear) {
        int current = queue[front++];
        printf("%d ", current);
        for (int v = 0; v < MAX_NODES; v++) {
            if (graph[current][v] && !visited[v]) {
                visited[v] = 1;
                queue[rear++] = v;
            }
        }
    }
}
```

```
int main() {
    int graph[MAX_NODES][MAX_NODES];
    initializeVisited();
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 0, 3);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);
    addEdge(graph, 4, 5);
    addEdge(graph, 5, 6);
    addEdge(graph, 6, 7);
    addEdge(graph, 7, 8);
    addEdge(graph, 8, 9);
    clock_t start = clock();
    breadthFirstSearch(graph, 0);
    clock_t end = clock();
    double timeTaken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken: %.5f seconds\n", timeTaken);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <time.h>
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    clock_t start = clock();
    selectionSort(arr, n);
    clock_t end = clock();
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    double timeTaken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken: %.5f seconds\n", timeTaken);
    return 0;
}
```

#]SLip-13

Q1)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
int matrixChainOrder(int p[], int n) {
    int m[n][n];
    int i, j, k, L;
    for (i = 0; i < n; i++) {
        m[i][i] = 0;
    }
    for (L = 2; L < n; L++) {
        for (i = 1; i < n - L + 1; i++) {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (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 p[] = {1, 2, 3, 4};
    int n = sizeof(p) / sizeof(p[0]);
    int minMultiplications = matrixChainOrder(p, n);
    printf("Minimum number of multiplications: %d\n", minMultiplications);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX_KEYS 10
int optimalBST(int keys[], int freq[], int n) {
    int cost[n][n];
    for (int i = 0; i < n; i++) {
        cost[i][i] = freq[i];
    }
    for (int L = 2; L <= n; L++) {
        for (int i = 0; i <= n - L + 1; i++) {
            int j = i + L - 1;
            cost[i][j] = INT_MAX;
            for (int r = i; r <= j; r++) {
                int c = ((r > i) ? cost[i][r - 1] : 0) +
                        ((r < j) ? cost[r + 1][j] : 0) +
                        sum(freq, i, j);
                if (c < cost[i][j]) {
                    cost[i][j] = c;
                }
            }
        }
    }
    return cost[0][n - 1];
}
```

```
int sum(int freq[], int i, int j) {
    int s = 0;
    for (int k = i; k <= j; k++) {
        s += freq[k];
    }
    return s;
}
int main() {
    int keys[MAX_KEYS] = {10, 12, 20};
    int freq[MAX_KEYS] = {34, 8, 50};
    int n = sizeof(keys) / sizeof(keys[0]);
    int minCost = optimalBST(keys, freq, n);
    printf("Minimum cost of optimal binary search tree: %d\n", minCost);
    return 0;
}
```

#]SLip-14

Q1)

```
#include <stdio.h>
#include <time.h>
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
int main() {
    int arr[] = {5, 2, 8, 6, 1, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    clock_t start = clock();
    insertionSort(arr, n);
    clock_t end = clock();
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    double timeTaken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken: %.5f seconds\n", timeTaken);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#define MAX_NODES 10000
int visited[MAX_NODES];
int adjacencyMatrix[MAX_NODES][MAX_NODES];
void initializeVisited() {
    for (int i = 0; i < MAX_NODES; i++) {
```

```
        visited[i] = 0;
    }
}
void addEdge(int graph[MAX_NODES][MAX_NODES], int u, int v) {
    graph[u][v] = 1;
    graph[v][u] = 1;
}
void depthFirstSearch(int graph[MAX_NODES][MAX_NODES], int current) {
    visited[current] = 1;
    printf("%d ", current);
    for (int i = 0; i < MAX_NODES; i++) {
        if (graph[current][i] && !visited[i]) {
            depthFirstSearch(graph, i);
        }
    }
}
void breadthFirstSearch(int graph[MAX_NODES][MAX_NODES], int start) {
    int queue[MAX_NODES];
    int front = 0, rear = 0;
    visited[start] = 1;
    queue[rear++] = start;
    while (front < rear) {
        int current = queue[front++];
        printf("%d ", current);
        for (int i = 0; i < MAX_NODES; i++) {
            if (graph[current][i] && !visited[i]) {
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
}
int main() {
    int graph[MAX_NODES][MAX_NODES];
    initializeVisited();
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 5);
    addEdge(graph, 2, 6);
    clock_t startDFS = clock();
    printf("Depth First Search: ");
    depthFirstSearch(graph, 0);
    printf("\n");
    clock_t endDFS = clock();
    initializeVisited();
    clock_t startBFS = clock();
    printf("Breadth First Search: ");
    breadthFirstSearch(graph, 0);
    printf("\n");
    clock_t endBFS = clock();
    double timeTakenDFS = (double)(endDFS - startDFS) / CLOCKS_PER_SEC;
    double timeTakenBFS = (double)(endBFS - startBFS) / CLOCKS_PER_SEC;
    printf("Time taken for DFS: %.5f seconds\n", timeTakenDFS);
    printf("Time taken for BFS: %.5f seconds\n", timeTakenBFS);
    return 0;
}
```

#]SLip-15

Q1)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX_ITEMS 100
typedef struct {
    int profit;
    int weight;
} Item;
int knapsack(Item items[], int n, int capacity) {
    int i, j;
    int** table = (int**)malloc(sizeof(int*) * (n + 1));
    for (i = 0; i <= n; i++) {
        table[i] = (int*)malloc(sizeof(int) * (capacity + 1));
    }
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= capacity; j++) {
            if (i == 0 || j == 0) {
                table[i][j] = 0;
            } else if (items[i - 1].weight <= j) {
                table[i][j] = max(items[i - 1].profit + table[i - 1][j - items[i -
1].weight], table[i - 1][j]);
            } else {
                table[i][j] = table[i - 1][j];
            }
        }
    }
    int maxProfit = table[n][capacity];
    for (i = 0; i <= n; i++) {
        free(table[i]);
    }
    free(table);
    return maxProfit;
}
int max(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
    Item items[] = {
        {60, 10},
        {100, 20},
        {120, 30}
    };
    int n = sizeof(items) / sizeof(items[0]);
    int capacity = 50;
    int maxProfit = knapsack(items, n, capacity);
    printf("Maximum profit: %d\n", maxProfit);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_NODES 100
int graph[MAX_NODES][MAX_NODES];
int colors[MAX_NODES];
bool isSafe(int node, int color, int graph[MAX_NODES][MAX_NODES], int colors[], int
currNode) {
    for (int i = 0; i < currNode; i++) {
        if (graph[i][node] && colors[i] == color) {
            return false;
        }
    }
    return true;
}
bool graphColoringUtil(int graph[MAX_NODES][MAX_NODES], int colors[], int currNode,
int n) {
    if (currNode == n) {
        return true;
    }
    for (int color = 1; color <= n; color++) {
        if (isSafe(currNode, color, graph, colors, currNode)) {
            colors[currNode] = color;
            if (graphColoringUtil(graph, colors, currNode + 1, n)) {
                return true;
            }
            colors[currNode] = 0; // Reset color if current color doesn't lead to a
solution
        }
    }
    return false;
}
bool graphColoring(int graph[MAX_NODES][MAX_NODES], int n) {
    for (int i = 0; i < n; i++) {
        colors[i] = 0;
    }
    if (graphColoringUtil(graph, colors, 0, n) == false) {
        printf("Solution does not exist\n");
        return false;
    }
    return true;
}
int main() {
    int n = 4;
    graph[0][1] = 1;
    graph[0][2] = 1;
    graph[1][2] = 1;
    graph[1][3] = 1;
    graph[2][3] = 1;
    if (graphColoring(graph, n)) {
        printf("Solution Exists\n");
        for (int i = 0; i < n; i++) {
            printf("Node %d - Color %d\n", i, colors[i]);
        }
    }
    return 0;
}
```

#]SLip-16

Q1)

```
#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int knapsack(int capacity, int weights[], int values[], int n) {
    int i, w;
    int K[n + 1][capacity + 1];
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0) {
                K[i][w] = 0;
            } else if (weights[i - 1] <= w) {
                K[i][w] = max(values[i - 1] + K[i - 1][w - weights[i - 1]], K[i -
1][w]);
            } else {
                K[i][w] = K[i - 1][w];
            }
        }
    }
    return K[n][capacity];
}
int main() {
    int values[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int capacity = 50;
    int n = sizeof(values) / sizeof(values[0]);
    int maxProfit = knapsack(capacity, weights, values, n);
    printf("Maximum profit: %d\n", maxProfit);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_NODES 100
int graph[MAX_NODES][MAX_NODES];
bool isSafe(int v, int graph[MAX_NODES][MAX_NODES], int path[], int pos, int V) {
    if (graph[path[pos - 1]][v] == 0) {
        return false;
    }
    for (int i = 0; i < pos; i++) {
        if (path[i] == v) {
            return false;
        }
    }
    return true;
}
bool hamCycleUtil(int graph[MAX_NODES][MAX_NODES], int path[], int pos, int V) {
    if (pos == V) {
        if (graph[path[pos - 1]][path[0]] == 1) {
            return true;
        } else {
            return false;
        }
    }
    for (int v = 1; v < V; v++) {
        if (isSafe(v, graph, path, pos, V)) {
            path[pos] = v;
```

```
            if (hamCycleUtil(graph, path, pos + 1, V)) {
                return true;
            }
            path[pos] = -1;
        }
    }
    return false;
}
bool hamCycle(int graph[MAX_NODES][MAX_NODES], int V) {
    int path[V];
    for (int i = 0; i < V; i++) {
        path[i] = -1;
    }
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1, V) == false) {
        printf("No Hamiltonian Cycle exists\n");
        return false;
    }
    printf("Hamiltonian Cycle exists\n");
    printf("Path: ");
    for (int i = 0; i < V; i++) {
        printf("%d ", path[i]);
    }
    printf("%d\n", path[0]);

    return true;
}
int main() {
    int V = 5;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph[i][j] = 0;
        }
    }
    graph[0][1] = 1;
    graph[1][2] = 1;
    graph[2][3] = 1;
    graph[3][4] = 1;
    graph[4][0] = 1;
    hamCycle(graph, V);
    return 0;
}
```

#]SLip-17

Q1)

```
#include <stdio.h>
#include <stdbool.h>
#define N 8
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}
bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    for (i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    return true;
}

bool solveNQueensUtil(int board[N][N], int col) {
    if (col >= N) {
        return true;
    }
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQueensUtil(board, col + 1)) {
                return true;
            }
            board[i][col] = 0;
        }
    }
    return false;
}
bool solveNQueens() {
    int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0} };
    if (solveNQueensUtil(board, 0) == false) {
        printf("Solution does not exist\n");
        return false;
```

```
    }
    printf("Solution found:\n");
    printSolution(board);
    return true;
}
int main() {
    solveNQueens();
    return 0;
}
```

Q2)

```
#include <stdio.h>
#define MAX_ITEMS 100
int knapSack(int W, int wt[], int val[], int n) {
    int i, w;
    int dp[MAX_ITEMS][MAX_ITEMS];
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (wt[i - 1] <= w) {
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][W];
}
int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    printf("Maximum value that can be put in a knapsack of capacity %d is %d\n", W,
knapSack(W, wt, val, n));
    return 0;
}
```

#]SLip-18

Q1)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 100
struct AdjListNode {
    int vertex;
    struct AdjListNode* next;
};
struct AdjList {
    struct AdjListNode* head;
};
struct Graph {
    int numVertices;
    struct AdjList adjList[MAX_VERTICES];
};
void addEdge(struct Graph* graph, int src, int dest) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
AdjListNode));
```

```
    newNode->vertex = dest;
    newNode->next = graph->adjList[src].head;
    graph->adjList[src].head = newNode;
    newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->vertex = src;
    newNode->next = graph->adjList[dest].head;
    graph->adjList[dest].head = newNode;
}
bool isSafe(int v, int color, int graph[MAX_VERTICES], int currVertex, int
numVertices) {
    struct AdjListNode* pCrawl = graph[v].head;
    while (pCrawl != NULL) {
        if (color == graph[pCrawl->vertex]) {
            return false;
        }
        pCrawl = pCrawl->next;
    }
    return true;
}

bool graphColoringUtil(struct Graph* graph, int m, int color[], int currVertex, int
numVertices) {
    if (currVertex == numVertices) {
        return true;
    }
    bool isColored = false;
    for (int c = 1; c <= m; c++) {
        if (isSafe(currVertex, c, graph, currVertex, numVertices)) {
            color[currVertex] = c;
            if (graphColoringUtil(graph, m, color, currVertex + 1, numVertices)) {
                return true;
            }
            color[currVertex] = 0;
        }
    }
    return false;
}
bool graphColoring(struct Graph* graph, int m, int numVertices) {
    int* color = (int*)malloc(numVertices * sizeof(int));
    for (int i = 0; i < numVertices; i++) {
        color[i] = 0;
    }
    if (!graphColoringUtil(graph, m, color, 0, numVertices)) {
        printf("Solution does not exist\n");
        free(color);
        return false;
    }
    printf("Solution found:\n");
    for (int i = 0; i < numVertices; i++) {
        printf("Vertex %d ---> Color %d\n", i, color[i]);
    }
    free(color);
    return true;
}
int main() {
    struct Graph graph;
    graph.numVertices = 4;
    addEdge(&graph, 0, 1);
    addEdge(&graph, 0, 2);
    addEdge(&graph, 1, 2);
```

```
    addEdge(&graph, 2, 3);
    addEdge(&graph, 3, 0);
    int m = 3;
    graphColoring(&graph, m, graph.numVertices);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 100
struct AdjListNode {
    int vertex;
    struct AdjListNode* next;
};
struct AdjList {
    struct AdjListNode* head;
};
struct Graph {
    int numVertices;
    struct AdjList adjList[MAX_VERTICES];
};
void addEdge(struct Graph* graph, int src, int dest) {
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
AdjListNode));
    newNode->vertex = dest;
    newNode->next = graph->adjList[src].head;
    graph->adjList[src].head = newNode;
    newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->vertex = src;
    newNode->next = graph->adjList[dest].head;
    graph->adjList[dest].head = newNode;
}
void findNodes(struct Graph* graph) {
    bool liveNodes[MAX_VERTICES] = {false};
    bool eNodes[MAX_VERTICES] = {false};
    bool visited[MAX_VERTICES] = {false};
    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            struct AdjListNode* pCrawl = graph->adjList[i].head;
            while (pCrawl != NULL) {
                liveNodes[i] = true;
                if (!visited[pCrawl->vertex]) {
                    visited[pCrawl->vertex] = true;
                }
                pCrawl = pCrawl->next;
            }
        }
    }
    for (int i = 0; i < graph->numVertices; i++) {
        if (!liveNodes[i]) {
            bool isE = false;
            struct AdjListNode* pCrawl = graph->adjList[i].head;
            while (pCrawl != NULL) {
                if (liveNodes[pCrawl->vertex]) {
                    eNodes[i] = true;
                    isE = true;
                    break;
                }
                pCrawl = pCrawl->next;
```

```
            }
            if (!isE) {
                printf("Dead node: %d\n", i);
            }
        }
    }
    printf("Live nodes: ");
    for (int i = 0; i < graph->numVertices; i++) {
        if (liveNodes[i]) {
            printf("%d ", i);
        }
    }
    printf("\n");
    printf("E nodes: ");
    for (int i = 0; i < graph->numVertices; i++) {
        if (eNodes[i]) {
            printf("%d ", i);
        }
    }
    printf("\n");
}
int main() {
    struct Graph graph;
    graph.numVertices = 4;
    addEdge(&graph, 0, 1);
    addEdge(&graph, 0, 2);
    addEdge(&graph, 1, 2);
    addEdge(&graph, 2, 3);
    addEdge(&graph, 3, 0);
    findNodes(&graph);
    return 0;
}
```

#]SLip-19

Q1)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 100
int graph[MAX_VERTICES][MAX_VERTICES];
bool isHamiltonianCycle(int path[], int V) {
    if (graph[path[V - 1]][path[0]] != 1) {
        return false;
    }
    bool visited[MAX_VERTICES] = {false};
    for (int i = 0; i < V; i++) {
        if (i > 0 && graph[path[i - 1]][path[i]] != 1) {
            return false;
        }
        if (visited[path[i]]) {
            return false;
        }
        visited[path[i]] = true;
    }
    return true;}
int main() {
    int V = 5;
    int path[] = {0, 1, 2, 3, 4, 0};
    int exampleGraph[V][V] = {
        {0, 1, 0, 0, 1},
```

```
        {1, 0, 1, 0, 1},
        {0, 1, 0, 1, 1},
        {0, 0, 1, 0, 1},
        {1, 1, 1, 1, 0}
    };
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph[i][j] = exampleGraph[i][j];
        }
    }
    if (isHamiltonianCycle(path, V)) {
        printf("The given path is a Hamiltonian cycle.\n");
    } else {
        printf("The given path is not a Hamiltonian cycle.\n");
    }
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#define N 4
void printBoard(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] == 1) {
                printf("Q ");
            } else {
                printf("_ ");
            }
        }
        printf("\n");
    }
    printf("\n");
}
bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    for (i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }


    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    return true;
}
bool solveNQueensUtil(int board[N][N], int col) {
    if (col >= N) {
        printBoard(board);
```

```
        return true;
    }
    bool res = false;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            res = solveNQueensUtil(board, col + 1) || res;
            board[i][col] = 0;
        }
    }
    return res;
}
void solveNQueens() {
    int board[N][N] = { {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0} };
    if (solveNQueensUtil(board, 0) == false) {
        printf("Solution does not exist");
    }
}
int main() {
    solveNQueens();
    return 0;
}
```

#]SLip-20

Q1)

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 100
int graph[MAX_VERTICES][MAX_VERTICES];
int indegree[MAX_VERTICES];
void addEdge(int graph[][MAX_VERTICES], int u, int v) {
    graph[u][v] = 1;
    indegree[v]++;
}
bool isSourceVertex(int u, int V) {
    for (int v = 0; v < V; v++) {
        if (graph[v][u] == 1) {
            return false;
        }
    }
    return true;
}
void topologicalSort(int V) {
    int count = 0;
    int sourceCount = 0;
    for (int u = 0; u < V; u++) {
        if (isSourceVertex(u, V)) {
            sourceCount++;
        }
    }
    int stack[MAX_VERTICES];
    int top = -1;
    for (int u = 0; u < V; u++) {
        if (isSourceVertex(u, V)) {
            stack[++top] = u;
        }
    }
```

```
    while (top != -1) {
        int u = stack[top--];
        printf("%d ", u);
        count++;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] == 1) {
                indegree[v]--;
                if (indegree[v] == 0) {
                    stack[++top] = v;
                }
            }
        }
    }
    if (count == V) {
        printf("\nTopological sorting is done.\n");
    } else {
        printf("\nThe given graph is not a DAG (Directed Acyclic Graph).\n");
    }
}

int main() {
    int V = 6;
    addEdge(graph, 5, 2);
    addEdge(graph, 5, 0);
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);
    printf("Topological sorting of the given graph: ");
    topologicalSort(V);
    return 0;
}
```

Q2)

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#define N 8
void printBoard(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] == 1) {
                printf("Q ");
            } else {
                printf("_ ");
            }
        }
        printf("\n");
    }
    printf("\n");
}
bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    for (i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
```

```
        if (board[i][j]) {
            return false;
        }
    }
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    return true;
}


bool solveNQueensUtil(int board[N][N], int col) {
    if (col >= N) {
        printBoard(board);
        return true;
    }
    bool res = false;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            res = solveNQueensUtil(board, col + 1) || res;
            board[i][col] = 0;
        }
    }
    return res;
}
void solveNQueens() {
    int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0} };
    if (solveNQueensUtil(board, 0) == false) {
        printf("Solution does not exist");
    }
}
int main() {
    solveNQueens();
    return 0;
}
```