CSE3004 (DAA) Lab 1 – 6

KHAN MOHD OWAIS RAZA (20BCD7138)

Lab-1: Implement Prim and Kruskal algorithms

Lab-2: Optimum ordering of Matrix Multiplication using Dynamic Approach

Lab-3: Implement Backtracking to avoid 8*8 Queens Problem

Lab-4: Strongly connected digraph components, Compression & Decompression

Lab-5: All pairs shortest path, greedy algorithm, dynamic programming

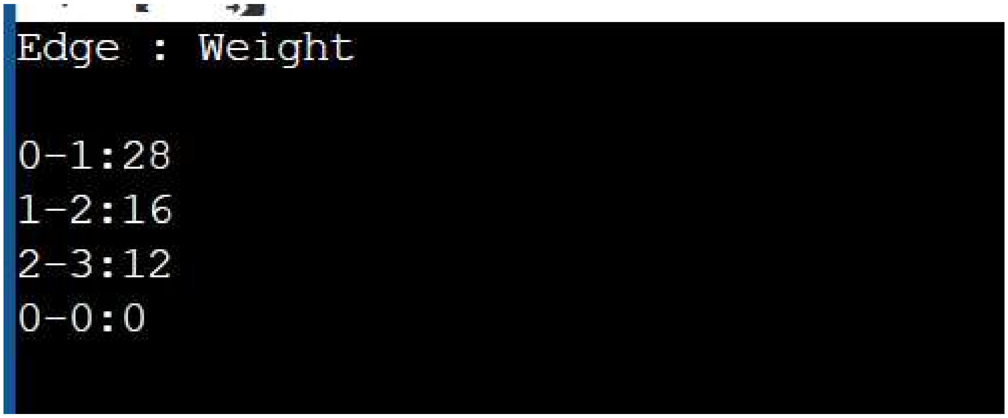
Lab-6 : Solve 0/1 knapsack prob using Backtracking & Branch and Bound

Design & Analysis of Algorithms Lab-1

Name: KHAN MOHD OWAIS RAZA
ID: 20BCD7138

Q1]

```
## KHAN MOHD OWAIS RAZA
## 20BCD7138
## DAA Lab-1
## Que-1
INF = 9999999
V = 5
G =  [[0,28,0,0,0,10,0],
     [28,0,16,0,0,0,14],
     [0,16,0,12,0,0,0],
     [0,0,0,22,0,25,24],
           [10,0,0,0,25,0,0],
           [0,14,0,18,24,0,0]]
selected = [0, 0, 0, 0, 0]
no_edge = 0
selected[0] = True
print("Edge : Weight\n")
while (no_edge < V - 1):
   minimum = INF
   x = 0
   y = 0
   for i in range(V):
      if selected[i]:
         for j in range(V):
            if ((not selected[j]) and G[i][j]):
               if minimum > G[i][j]:
                  minimum = G[i][j]
                  x = i
                  y = j
   print(str(x) + "-" + str(y) + ":" + str(G[x][y]))
   selected[y] = True
   no_edge += 1
```



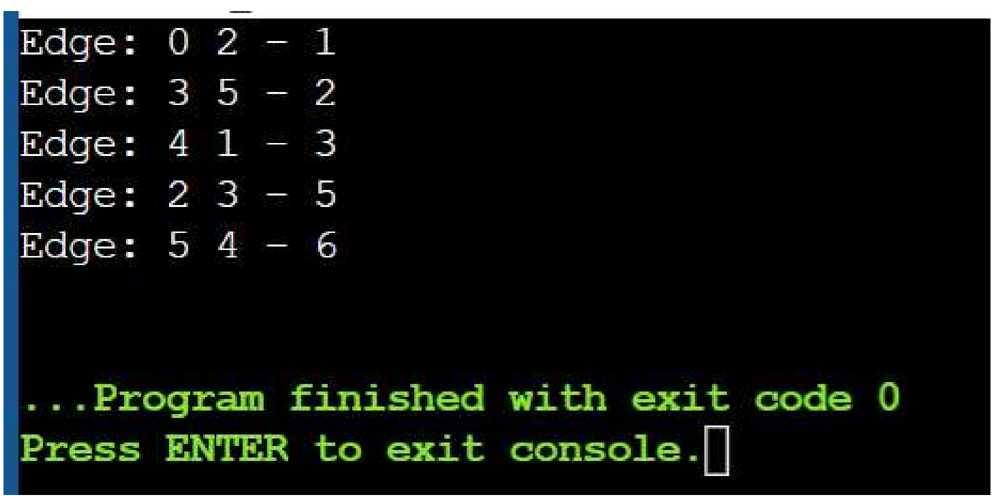
```
Edge : Weight

0-1:28
1-2:16
2-3:12
0-0:0
```

Q2]

```
class Graph:
    def __init__(self, vertex):
        self.V = vertex
        self.graph = []
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
    def search(self, parent, i):
        if parent[i] == i:
            return i
        return self.search(parent, parent[i])
    def apply_union(self, parent, rank, x, y):
        xroot = self.search(parent, x)
        yroot = self.search(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
    def kruskal(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
```

```
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.search(parent, u)
            y = self.search(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("Edge:",u, v,end =" ")
            print("-",weight)
g = Graph(6)
g.add_edge(0,2,1)
g.add_edge(2,3,5)
g.add_edge(3,5,2)
g.add_edge(5,4,6)
g.add_edge(4,1,3)
g.kruskal()
```



```
Edge: 0 2 - 1
Edge: 3 5 - 2
Edge: 4 1 - 3
Edge: 2 3 - 5
Edge: 5 4 - 6


...Program finished with exit code 0
Press ENTER to exit console.
```

CSE3004 (DAA) Lab-2

KHAN MOHD OWAIS RAZA
20BCD7138

Implement optimum ordering of matrix multiplication using dynamic approach

```
class Main {
static int MatrixChainOrder(int p[], int n){
int m[][] = new int[n][n];
int i, j, k, L, q;
for (i = 1; 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;
if (j == n)
continue;
m[i][j] = Integer.MAX_VALUE;
for (k = i; k <= j - 1; k++) {
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];
}
public static void main(String args[]){
int arr[] = new int[] { 1, 2, 3, 4 };
int size = arr.length;
System.out.println("Minimum number of multiplications is "
+ MatrixChainOrder(arr, size));
}}
```

```
Minimum number of multiplications is 18


...Program finished with exit code 0
Press ENTER to exit console.
```

CSE3004 (DAA) Lab-3

KHAN MOHD OWAIS RAZA
20BCD7138

Topic :- Implement Backtracking to avoid 8*8 Queens Problem

C program :-

```
// KHAN MOHD OWAIS RAZA
// 20BCD7138
// CSE3004 (DAA) Lab-3
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int **board;
    int size;
}
board;
board* createBoard(int size) {
    board *b = malloc(sizeof(board));
    b->board = malloc(sizeof(int*) * size);
    for (int i = 0; i < size; i++) {
        b->board[i] = malloc(sizeof(int) * size);
        for (int j = 0; j < size; j++) {
            b->board[i][j] = 0;
        }
    }
    b->size = size;
    return b;
}
void destroyBoard(board *b) {
    for (int i = 0; i < b->size; i++) {
        free(b->board[i]);
    }
    free(b->board);
    free(b);
}
void printBoard(board *b) {
    for (int i = 0; i < b->size; i++) {
        for (int j = 0; j < b->size; j++) {
            printf("%d ", b->board[i][j]);
        }
        printf("\n");
}}
int isSafe(board *b, int row, int col) {
```

```
    int i, j;
    for (i = 0; i < col; i++) {
        if (b->board[row][i]) {
            return 0;
    }}
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (b->board[i][j]) {
            return 0;
    }}
    for (i = row, j = col; j >= 0 && i < b->size; i++, j--) {
        if (b->board[i][j]) {
            return 0;
    }}
    return 1;
}
int solveBoard(board *b, int col) {
    if (col >= b->size) {
        return 1;
    }
    for (int i = 0; i < b->size; i++) {
        if (isSafe(b, i, col)) {
            b->board[i][col] = 1;
            if (solveBoard(b, col + 1)) {
                return 1;
            }
            b->board[i][col] = 0;
    }}
    return 0;
}
int solve(board *b) {
    if (solveBoard(b, 0) == 0) {
        printf("Solution does not exist\n");
        return 0;
    }
    printBoard(b);
    return 1;
}
int main() {
    board *b = createBoard(8);
    solve(b);
    destroyBoard(b);

    return 0;
}
```

main.c Output

```c
// KHAN MOHD OWAIS RAZA
// 20BCD7138
// CSE3004 (DAA) Lab-3
#include <stdio.h>
#include <stdlib.h>
typedef struct {
int **board;
int size;
}
board;
board* createBoard(int size) {
board *b = malloc(sizeof(board));
b->board = malloc(sizeof(int*)*size);
for (int i = 0; i < size; i++) {
b->board[i] = malloc(sizeof(int)*size);
for (int j = 0; j < size; j++) {
b->board[i][j] = 0;
}}
b->size = size;
return b;
}
void destroyBoard(board *b) {
for (int i = 0; i < b->size; i++) {
free(b->board[i]);
}
free(b->board);
free(b);
}
void printBoard(board *b) {
for (int i = 0; i < b->size; i++) {
for (int j = 0; j < b->size; j++) {
printf("%d ", b->board[i][j]);
}
printf("\n");
}}
int isSafe(board *b, int row, int col){
int i, j;
for (i = 0; i < col; i++) {
if (b->board[row][i]) {
return 0;
}}
for (i = row, j = col; i >= 0
&& j >= 0; i--, j--) {
if (b->board[i][j]) {
return 0;
}}
for (i = row, j = col; j >= 0
&& i < b->size; i++, j--) {
if (b->board[i][j]) {
return 0;
}}
```

```c
return 1;
}
int solveBoard(board *b, int col) {
if (col >= b->size) {
return 1;
}
for (int i = 0; i < b->size; i++) {
if (isSafe(b, i, col)) {
b->board[i][col] = 1;
if (solveBoard(b, col + 1)) {
return 1;
}
b->board[i][col] = 0;
}}
return 0;
}
int solve(board *b) {
if (solveBoard(b, 0) == 0) {
printf("Solution does not exist\n");
return 0;
}
printBoard(b);
return 1;
}
int main() {
board *b = createBoard(8);
solve(b);
destroyBoard(b);
return 0;
}
```

Output :-

```
main.c    Output

/tmp/zyEpoNKiPW.o
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

CSE2007 (DBMS) Lab-4

KHAN MOHD OWAIS RAZA
20BCD7138

Find the strongly connected components in a digraph.

```
import java.io.*;
import java.util.*;
class Graph {
    private int V;
    private LinkedList<Integer> adj[];
    private int Time;
    @SuppressWarnings("unchecked")
 Graph(int v){
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
        adj[i] = new LinkedList();
        Time = 0;
    }
    void addEdge(int v, int w) {
    adj[v].add(w);
    }
    void SCCUtil(int u, int low[], int disc[],
    boolean stackMember[], Stack<Integer> st){
        disc[u] = Time;
        low[u] = Time;
        Time += 1;
        stackMember[u] = true;
        st.push(u);
```

```
        int n;
        Iterator<Integer> i = adj[u].iterator();
while (i.hasNext()) {
            n = i.next();
            if (disc[n] == -1) {
                SCCUtil(n, low, disc, stackMember,
st);
                low[u] = Math.min(low[u], low[n]);
            }
            else if (stackMember[n] == true) {
                low[u] = Math.min(low[u], disc[n]);
            }
        }
        int w = -1;
        if (low[u] == disc[u]) {
            while (w != u) {
                w = (int)st.pop();
                System.out.print(w + " ");
                stackMember[w] = false;
            }
            System.out.println();
        }
    }
    void SCC() {
        int disc[] = new int[V];
        int low[] = new int[V];
        for (int i = 0; i < V; i++) {
            disc[i] = -1;
            low[i] = -1;
        }
        boolean stackMember[] = new boolean[V];
        Stack<Integer> st = new Stack<Integer>();
```

```
        for (int i = 0; i < V; i++) {
            if (disc[i] == -1)
            SCCUtil(i, low, disc, stackMember, st);
        }}
    public static void main(String args[]) {
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        System.out.println("SSC in first graph ");
        g1.SCC();
        Graph g2 = new Graph(4);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        g2.addEdge(2, 3);
        System.out.println("\nSSC in second graph ");
        g2.SCC();
        Graph g3 = new Graph(7);
        g3.addEdge(0, 1);
        g3.addEdge(1, 2);
        g3.addEdge(2, 0);
        g3.addEdge(1, 3);
        g3.addEdge(1, 4);
        g3.addEdge(1, 6);
        g3.addEdge(3, 5);
        g3.addEdge(4, 5);
        System.out.println("\nSSC in third graph ");
        g3.SCC();
        Graph g4 = new Graph(11);
        g4.addEdge(0, 1);
```

```
        g4.addEdge(0, 3);
        g4.addEdge(1, 2);
        g4.addEdge(1, 4);
        g4.addEdge(2, 0);
        g4.addEdge(2, 6);
        g4.addEdge(3, 2);
        g4.addEdge(4, 5);
        g4.addEdge(4, 6);
        g4.addEdge(5, 6);
        g4.addEdge(5, 7);
        g4.addEdge(5, 8);
        g4.addEdge(5, 9);
        g4.addEdge(6, 4);
        g4.addEdge(7, 9);
        g4.addEdge(8, 9);
        g4.addEdge(9, 8);
        System.out.println("\nSSC in fourth graph ");
        g4.SCC();
        Graph g5 = new Graph(5);
        g5.addEdge(0, 1);
        g5.addEdge(1, 2);
        g5.addEdge(2, 3);
        g5.addEdge(2, 4);
        g5.addEdge(3, 0);
        g5.addEdge(4, 2);
        System.out.println("\nSSC in fifth graph ");
        g5.SCC();
    }
}
```

Output

```
java -cp /tmp/XP4EbhtNOX Graph
SSC in first graph 4 3
1 2 0

SSC in second graph
3
2
1
0

SSC in third graph
5 3
4
6
2 1 0

SSC in fourth graph
8 9
7
5 4 6
3 2 1 0
10

SSC in fifth graph
4 3 2 1 0
```

Implement Compression and Decompression using Huffman's Algorithm.

```
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;
class Huffman {
	public static void printCode(HuffmanNode root, String s){
		if (root.left
				== null
			&& root.right
				== null
			&& Character.isLetter(root.c)) {
	System.out.println(root.c + ":" + s);
			return;
		}
		printCode(root.left, s + "0");
		printCode(root.right, s + "1");
	}
	public static void main(String[] args){
Scanner s = new Scanner(System.in);
		int n = 6;
		char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
		int[] charfreq = { 5, 9, 12, 13, 16, 45 };
	PriorityQueue<HuffmanNode> q
			= new PriorityQueue<HuffmanNode>(n, new
MyComparator());
		for (int i = 0; i < n; i++){
			HuffmanNode hn = new HuffmanNode();
			hn.c = charArray[i];
			hn.data = charfreq[i];
```

```
                hn.left = null;
                hn.right = null;
                q.add(hn);
            }
            HuffmanNode root = null;
            while (q.size() > 1) {
                HuffmanNode x = q.peek();
                q.poll();
                HuffmanNode y = q.peek();
                q.poll();
                HuffmanNode f = new HuffmanNode();
                f.data = x.data + y.data;
                f.c = '-';
                f.left = x;
                f.right = y;
                root = f;
                q.add(f);
            }
            printCode(root, "");
        }
}
class HuffmanNode {
        int data;
        char c;
        HuffmanNode left;
        HuffmanNode right;
}
class MyComparator implements Comparator<HuffmanNode> {
        public int compare(HuffmanNode x, HuffmanNode y){
            return x.data - y.data;
        }
}
```

Output

```
java -cp /tmp/KzYYx5hPCJ Huffman
f:0
c:100
d:101
a:1100
b:1101
e:111
```

CSE3004 (DAA) Lab-5

KHAN MOHD OWAIS RAZA
20BCD7138

Q1) *Implement dynamic programming algorithm to solve all pairs shortest path problem*

```
import java.util.Scanner;
public class AllPairShortestPath{
   private int distancematrix[][];
   private int numberofvertices;
   public static final int INFINITY = 999
   public AllPairShortestPath(int numberofvertices)
   {
      distancematrix = new int[numberofvertices + 1][numberofvertices + 1];
      this.numberofvertices = numberofvertices;
   }

   public void allPairShortestPath(int adjacencymatrix[][])
   {
      for (int source = 1; source <= numberofvertices; source++)
      {
         for (int destination = 1; destination <= numberofvertices; destination++)
         {
            distancematrix[source][destination] =
adjacencymatrix[source][destination];
         }
      }

      for (int intermediate = 1; intermediate <= numberofvertices; intermediate++)
      {
         for (int source = 1; source <= numberofvertices; source++)
         {
            for (int destination = 1; destination <= numberofvertices; destination++)
            {
```

```
                    if (distancematrix[source][intermediate] +
distancematrix[intermediate][destination]
                                        < distancematrix[source][destination])
                        distancematrix[source][destination] =
distancematrix[source][intermediate]
                                        + distancematrix[intermediate][destination];
                }
            }
        }

        for (int source = 1; source <= numberofvertices; source++)
            System.out.print("\t" + source);

        System.out.println();
        for (int source = 1; source <= numberofvertices; source++)
        {
            System.out.print(source + "\t");
            for (int destination = 1; destination <= numberofvertices; destination++)
            {
                System.out.print(distancematrix[source][destination] + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String... arg)
    {
        int adjacency_matrix[][];
        int numberofvertices;

        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the number of vertices");
        numberofvertices = scan.nextInt();

        adjacency_matrix = new int[numberofvertices + 1][numberofvertices + 1];
        System.out.println("Enter the Weighted Matrix for the graph");
        for (int source = 1; source <= numberofvertices; source++)
```

```
        {
            for (int destination = 1; destination <= numberofvertices; destination++)
            {
                adjacency_matrix[source][destination] = scan.nextInt();
                if (source == destination)
                {
                    adjacency_matrix[source][destination] = 0;
                    continue;
                }
                if (adjacency_matrix[source][destination] == 0)
                {
                    adjacency_matrix[source][destination] = INFINITY;
                }
            }
        }

        System.out.println("The Transitive Closure of the Graph");
        AllPairShortestPath allPairShortestPath= new
AllPairShortestPath(numberofvertices);
        allPairShortestPath.allPairShortestPath(adjacency_matrix);

        scan.close();
    }
}
```

```
$javac AllPairShortestPath.java
$java AllPairShortestPath

Enter the number of vertices
4

Enter the Weighted Matrix for the graph
0 0 3 0
2 0 0 0
0 7 0 1
6 0 0 0

The Transitive Closure of the Graph

    1   2   3   4
1   0   10  3   4
2   2   0   5   6
3   7   7   0   1
4   6   16  9   0
```

Q2) *Solve 0/1 knapsack problem using :*

a) Greedy algorithm

b) Dynamic programming algorithm

```
public class Knapsack {

    // Greedy algorithm
    public static int knapsackGreedy(int W, int[] wt, int[] val) {
        int n = wt.length;
        int maxVal = 0;

        // Calculate value-to-weight ratio for each item
        double[] ratio = new double[n];
        for (int i = 0; i < n; i++) {
            ratio[i] = (double) val[i] / wt[i];
        }

        // Sort items in descending order of value-to-weight ratio
        for (int i = 0; i < n; i++) {
            int maxIndex = i;
            for (int j = i+1; j < n; j++) {
                if (ratio[j] > ratio[maxIndex]) {
                    maxIndex = j;
                }
            }
            double temp = ratio[i];
            ratio[i] = ratio[maxIndex];
            ratio[maxIndex] = temp;

            int tempWt = wt[i];
            wt[i] = wt[maxIndex];
            wt[maxIndex] = tempWt;

            int tempVal = val[i];
            val[i] = val[maxIndex];
```

```
        val[maxIndex] = tempVal;
    }

    // Add items to knapsack in descending order of value-to-weight ratio
    for (int i = 0; i < n; i++) {
        if (W >= wt[i]) {
            W -= wt[i];
            maxVal += val[i];
        } else {
            maxVal += (int) (ratio[i] * W);
            break;
        }
    }

    return maxVal;
}

// Dynamic programming algorithm
public static int knapsackDP(int W, int[] wt, int[] val) {
    int n = wt.length;
    int[][] dp = new int[n+1][W+1];

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (wt[i-1] <= w) {
                dp[i][w] = Math.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];
}
```

```
    public static void main(String[] args) {
        int W = 50;
        int[] wt = {10, 20, 30};
        int[] val = {60, 100, 120};

        // Solve using greedy algorithm
        int maxValGreedy = knapsackGreedy(W, wt, val);
        System.out.println("Greedy algorithm: \nMaximum value = " + 
maxValGreedy);
        System.out.println("  ");

        // Solve using dynamic programming algorithm
        int maxValDP = knapsackDP(W, wt, val);
        System.out.println("Dynamic programming algorithm: \nMaximum value = " 
+ maxValDP);
    }
}
```

```
java -cp /tmp/RsjLkaoZkT Knapsack
Greedy algorithm:
Maximum value = 240

Dynamic programming algorithm:
Maximum value = 220
```

CSE3004 (DAA) Lab-6

KHAN MOHD OWAIS RAZA
20BCD7138

Solve 0/1 knapsack problem using Backtracking & Branch and Bound.

```
import java.util.*;
public class Main {
    static int maxProfit = 0;
    public static void main(String[] args) {
        int[] weights = {10, 20, 30};
        int[] profits = {60, 100, 120};
        int capacity = 50;
        System.out.println("Using Backtracking:");
        knapsackBacktracking(weights, profits, capacity, 0, 0, 0);
        System.out.println("Maximum profit using Backtracking: " + maxProfit);
        System.out.println("Using Branch and Bound:");
        maxProfit = knapsackBranchAndBound(weights, profits, capacity);
        System.out.println("Maximum profit using Branch and Bound: " +
maxProfit);
    }
    public static void knapsackBacktracking(int[] weights, int[] profits, int capacity,
int currentWeight, int currentProfit, int index) {
        if (index >= weights.length || currentWeight == capacity) {
            if (currentProfit > maxProfit) {
                maxProfit = currentProfit;
            }
            return;
        }
        if (currentWeight + weights[index] <= capacity) {
            knapsackBacktracking(weights, profits, capacity, currentWeight +
weights[index], currentProfit + profits[index], index + 1);
        }
        knapsackBacktracking(weights, profits, capacity, currentWeight,
currentProfit, index + 1);
    }
```

```
    public static int knapsackBranchAndBound(int[] weights, int[] profits, int
capacity) {
        int n = weights.length;
        // Create list of items and sort by decreasing profit/weight ratio
        List<Item> items = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            items.add(new Item(weights[i], profits[i], i));
        }
        items.sort((a, b) -> Double.compare(b.profitWeightRatio,
a.profitWeightRatio));
        // Initialize root node of search tree
        Node root = new Node(-1, 0, 0, 0);
        // Initialize priority queue with root node
        PriorityQueue<Node> pq = new PriorityQueue<>();
        pq.offer(root);
        while (!pq.isEmpty()) {
            Node node = pq.poll();
            // If we have reached a leaf node, update maxProfit and continue
            if (node.level == n - 1) {
                if (node.profit > maxProfit) {
                    maxProfit = node.profit;
                }
                continue;
            }
            // Create left child node by including the next item in the knapsack
            int nextIndex = node.level + 1;
            if (node.weight + items.get(nextIndex).weight <= capacity) {
                Node leftChild = new Node(nextIndex, node.weight +
items.get(nextIndex).weight, node.profit + items.get(nextIndex).profit,
node.bound);
                pq.offer(leftChild);
            }
            // Create right child node by excluding the next item from the knapsack
            Node rightChild = new Node(nextIndex, node.weight, node.profit,
bound(items, capacity, node.level + 1, node.weight, node.profit));
            if (rightChild.bound > maxProfit) {
                pq.offer(rightChild);
```

```
            }
        }
        return maxProfit;
    }
    public static int bound(List<Item> items, int capacity, int index, int
currentWeight, int currentProfit) {
        int bound = currentProfit;
        // Add as much of the next item as possible
        while (index < items.size() && currentWeight + items.get(index).weight <=
capacity) {
            bound += items.get(index).profit;
            currentWeight += items.get(index).weight;
            index++;
        }
        // Add a fraction of the next item if it doesn't fit entirely
        if (index < items.size()) {
            int remainingCapacity = capacity - currentWeight;
            double fraction = (double) remainingCapacity / items.get(index).weight;
            bound += fraction * items.get(index).profit;
        }
        return bound;
    }
    static class Item {
        int weight;
        int profit;
        double profitWeightRatio;
        public Item(int weight, int profit, int index) {
            this.weight = weight;
            this.profit = profit;
            this.profitWeightRatio = (double) profit / weight;
        }
    }
    static class Node implements Comparable<Node> {
        int level;
        int weight;
        int profit;
        int bound;
```

```
        public Node(int level, int weight, int profit, int bound) {
            this.level = level;
            this.weight = weight;
            this.profit = profit;
            this.bound = bound;
        }

        public int compareTo(Node other) {
            return Integer.compare(other.bound, bound);
        }
    }
}
```