Experiment #11		Student ID	
Date		Student Name	

Experiment Title: Implementation of Programs on Branch and Bound Problems – TSP and 0/1 Knapsack.

Aim/Objective: : To understand the concept and implementation of Basic programs on Branch and Bound Technique Problems.

Description: The students will understand and able to implement programs on Branch and Bound Technique Problems.

Pre-Requisites:

Knowledge: Branch and Bound Technique in C/C++/Python

Tools: CodeBlocks/EclipseIDE/Python IDLE

Pre-Lab:

1. XYZ Logistics is a medium-sized logistics company that provides delivery services across multiple cities. The company aims to minimize travel costs and delivery times to improve efficiency and customer satisfaction. They face the challenge of determining the optimal route for their delivery trucks that need to visit several cities and return to the starting point. The company decides to implement the Branch and Bound method to solve their Traveling Salesman Problem (TSP).

Problem Formulation:

- Objective: Minimize the total distance travelled by a delivery truck.
- Constraints: Each city must be visited exactly once, and the truck must return to the starting city.

Let's consider a scenario where there are 4 cities (A, B, C, and D) and the distances between them are as follows:

- Distance from A to B: 10
- Distance from A to C: 15
- Distance from A to D: 20
- Distance from B to C: 35
- Distance from B to D: 25
- Distance from C to D: 30

The goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#define NUM_CITIES 4
int distance[NUM_CITIES][NUM_CITIES] = {
    {0, 10, 15, 20},
    {10, 0, 35, 25},
    {15, 35, 0, 30},
    {20, 25, 30, 0},
};
int visited[NUM_CITIES] = {0};
int tsp(int pos, int n, int count, int cost,
                                    int ans){
if(count == n && distance[pos][0]){
return ans < cost + distance[pos][0] ? ans :
        cost + distance[pos][0];
}
for(int i = 0; i < n; i++){
  if(!visited[i] && distance[pos][i])
      visited[i] = 1;
ans = tsp(i, n, count + 1, cost + distance
        [pos][i], ans);
      visited[i] = 0;
  }
}
```

```
    return ans;
}

int main(){
visited[0] = 1;
    int mincost = tsp(0, NUM_CITIES,
    1, 0, INT_MAX);

printf("Minimum travel cost: %d\n",
                        mincost);

    return 0;

}
```

- **Data and Results:**

~~Data~~ Traveling salseman Problem solved using Branch and Bound; minimum cost found

- **Analysis and Inferences:**

Branch and Bound effectively minimized travel cost, optimizing delivery route

2. Consider the following 0/1 Knapsack Problem:

You are a thief planning to rob a store. You have a backpack with a maximum weight capacity of 15 kilograms. The store contains the following items:

Item	Weight (kg)	Value ($)
A	2	10
B	4	25
C	6	15
D	3	20
E	5	30

You can only take whole items (either you take an item completely or you don't take it at all). Determine the maximum value of items you can steal without exceeding the weight capacity of your backpack using the branch and bound algorithm.

Solution Approach:

- **Formulate the problem:** Define the objective function and constraints.
- **Apply the branch and bound algorithm:** Break down the problem into subproblems and use bounding techniques to narrow down the search space.
- **Find the optimal solution:** Determine the maximum value of items that can be stolen without exceeding the weight capacity.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int weight, value;
    double ratio;
} Item;
```

```
int compare(const void *a, const void *b) {
return ((Item*)b)->ratio - ((Item*)a)->ratio;
}
double bound(Item items[], int n, int capacity, int
                currentweight, int currenvalue,
            int index) {
if (currentweight >= capacity) return 0;
  double valueBound = currentvalue;
while (index < n && currentweight + items[index].
                weight <= capacity) {
currentweight += items[index].weight;
    valueBound += items[index].value;
      index++;
  }
  if(index<n) valueBound += (capacity - currentweight)*
                items[index].ratio;
  return valueBound;
}
int knapsack(Item items[], int n, int capacity) {
 qsort(items, n, sizeof(Item), compare);
    int maxvalue = 0
 struct Node { int level, Profit, weight; };
                queue[1000];
 int front = 0,    rear = 0;

 queue[rear++] = (struct Node) {-1, 0, 0};
```

```
int main(){
Item items[ ]={{12, 10, 5}, {14, 25, 6.25},
              {6, 15, 2.5}, {13, 20, 6.67},
                            {5, 30, 6}};
int n = sizeof (items) / sizeof (items[0]);
    int maxValue = knapsack (items, n, 15);
Printf ("Maximum value in knapsack =
                        %.d\n", maxvalue);
return 0;
}
```

- **Data and Results:**

Thief maximizes stolen value to ss without exceeding weight limit

- **Analysis and Inferences:**

Branch and Bound effectively optimizes item selection for maximum value.

In-Lab:

Task1: Traveling Salesman Problem with Least Cost Branch and Bound

Problem: Imagine you're a delivery driver tasked with visiting all customers in a city but minimizing the total distance traveled. You need to find the most efficient route that ensures you visit each customer and return to the starting point.

Objective:

Implement the Traveling Salesman Problem (TSP) using the Least Cost Branch and Bound algorithm to find the shortest route a salesperson can take to visit all cities exactly once and return to the starting point.

Approach: You decide to use the Lower Cost Branch and Bound (LCBB) algorithm to efficiently search the solution space.

Input:

- **Distance Matrix:** A 2D array of size (n x n), where n is the number of cities. Each entry

distanceMatrix[i][j] represents the cost/distance between city i and city j. The matrix should be symmetric (distance from i to j is the same as j to i).

Output:

- **Minimum Tour Cost:** The total cost of the shortest route visiting all cities exactly once and returning to the starting point.
- **Tour Path (Optional):** An ordered list of city indices representing the optimal route for the salesman.

Additional Considerations:

- You can decide whether to accept the distance matrix directly as input or allow the user to enter the distances for each pair of cities.
- The program should handle invalid inputs like non-symmetric distance matrices or negative distances.
- Error messages or appropriate handling should be implemented for such cases.
- The output format for the minimum tour cost can be a simple numeric value.
- The optional tour path output can be an array or list of city indices in the optimal visiting order.

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define N 4
int tsp(int distanceMatrix[N][N]){
    int visited[N] = {0};
    int minCost = INT_MAX;
void dfs(int currentCity, int currentCost,
    int count, int startCity){
```

```
if (count == N && distanceMatrix[currentCity]
                    [startCity] > 0) {

int totalCost = currentCost + distanceMatrix[curren
            city] [startCity];

if (totalCost < minCost) {
    minCost = totalCost;
  }
  return;
}

for(int nextCity = 0; nextCity < N; nextCity++) {
if (!visited[nextCity] && distanceMatrix[currenCity]
                              [nextCity] > 0) {

  visited[nextCity] = 1;
tsp(nextCity, currentCost + distanceMatrix
          [currentCity][nextCity], count+1,
                              startCity);

  visited[nextCity] = 0;
  }
 }
}
  visited[0] = 1;
  tsp(0, 0, 1, 0);
    return minCost;
}
int main() {
  int distanceMatrix[N][N] = {
{ 0, 10, 15, 20},
```

```
{10, 0, 35, 25},
{15, 35, 0, 30},
{20, 25, 30, 0}
};
int result = tsp(distanceMatrix);
printf("Minimum Tour cost : %d\n", result);
return 0;
}
```

- **Data and Results:**

The distance matrix represents city distances; minimum tour cost computed

- **Analysis and Inferences:**

The algorithm efficiently finds optimal route minimizing travel distance.

Task2: 0/1 knapsack problem with Least Cost Branch and Bound

Problem: Imagine you're a thief planning a heist and need to choose the most valuable items from a safe that has a weight limit. Each item has a value and a weight. You can only take an item entirely (not a fraction) and aim to maximize the total value of stolen items without exceeding the safe's weight capacity.

Objective:

Implement the 0/1 Knapsack Problem using the Branch and Bound algorithm to find the optimal selection of items with maximum total value that fits within a limited knapsack capacity.

Approach: You decide to use the Lower Cost Branch and Bound (LCBB) algorithm to efficiently search the solution space.

Input:

- **Item Values:** An array of size n where values[i] represents the value of item i.
- **Item Weights:** An array of size n where weights[i] represents the weight of item i.
- **Knapsack Capacity:** The weight limit of the knapsack.

Output:

- **Maximum Total Value:** The total value of the optimal selection of items that fits within the knapsack capacity.
- **Optimal Selection (Optional):** An ordered list of item indices representing the items included in the optimal solution.
- **Additional Considerations:**
- You can decide whether to accept the item values, weights, and capacity directly as input or allow the user to enter them individually.
- The program should handle invalid inputs like negative values or a capacity that cannot accommodate any items.
- Error messages or appropriate handling should be implemented for such cases.
- The output format for the maximum total value can be a simple numeric value.
- The optional optimal selection output can be an array or list of item indices in the knapsack for the optimal solution.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int index, value, weight;
} Item;

int cmp(const void *a, const void *b) {
return ((Item *)b)->value * ((Item *)a)->
    weight - ((Item *)a)->value * (Item *)b)
        ->weight;
}
```

```
int knapsack(int n, int w, Item items[]){
qsort(items, n, sizeof(Item), cmp);
    int max_value = 0;
for(int i = 0; i < (1<<n); i++){
  int total_weight = 0, total_value = 0;
  for(int j = 0; j < n; j++){
    if(i & (1 << j)){
  total_weight += items[j].weight;
  total_value += items[j].value;
    }
  }
if (total_weight <= w && total_value >
                          max_value){
  max_value = total_value;
  }
}
return max_value;
}
int main(){
Item items[] = {{0, 60, 2}, {1, 100, 3},
                {2, 120, 4}};
  int n = sizeof(items) / sizeof(items[0]),
                                  w=5;
```

```
Printf ("Max value: %.d\n", knapsack
                (n, w, items));

    return 0;
}
```

- **Data and Results:**

Input values, weights, and knapsack capacity lead to maximum value

- **Analysis and Inferences:**

Optimizing item selection maximizes value while respecting weight constraints

Post Lab:

1. Problem Statement: There are N cities. The time it takes to travel from City i to City j is $T_{i,j}$. Among those paths that start at City 1, visit all other cities exactly once, and then go back to City 1, how many paths take the total time of exactly K to travel along?

Constraints:

$2 \leq N \leq 8$
if $i \neq j, 1 \leq T_{i,j} \leq 10^8$
$T_{i,i} = 0$
$T_{i,j} = T_{j,i}$
$1 \leq K \leq 10^9$
All the Input values are integers

Input

Input is given from Standard Input in the following format:

$N\ K$
$T_{1,1} \cdots\cdots T_{1,N}$
$\vdots$
$T_{1,N} \cdots\cdots T_{N,N}$

Output

Print the answer as an integer.

Sample Input 1

```
4 330
0 1 10 100
1 0 20 200
10 20 0 300
100 200 300 0
```

Sample Output 1

2

There are six paths that start at City 1, visit all other cities exactly once, and then go back to City 1:

1→2→3→4→1
1→2→4→3→1
1→3→2→4→1
1→3→4→2→1
1→4→2→3→1
1→4→3→2→1

The times it takes to travel along these paths are 421, 511, 330, 511, 330, and 421, respectively, among which two are exactly 330.

Sample Input 2

```
5 5
0 1 1 1 1
1 0 1 1 1
1 1 0 1 1
1 1 1 0 1
1 1 1 1 0
```

Sample Output 2

24

In whatever order we visit the cities, it will take the total time of 5 to travel.

- **Program:**

```
#include <stdio.h>
#define MAX_CITIES 10
#define INF 999999
int N;
int K;
int T[MAX_CITIES][MAX_CITIES];
```

```
int count = 0;
int visited[MAX_CITIES];

void dfs(int current, int total_time, int visited_
                                    count){
if(visited_count == N && current == 0){
  if(total_time == K){
    count++;
  }
  return;
}
for(int next=0; next<N; next++){
if(next != current && !visited[next]){
  visited[next] = 1;
  dfs(next, total_time + T[current][next],
                visited_count + 1);
  visited[next] = 0;
}
}
}

int main(){
scanf("%d %d", &N, &K);
for(int i=0; i<N; i++){
  for(int j=0; j<N; j++){
    scanf("%d", &T[i][j]);
}}
```

```
visited [0] = 1;
dfs(0, 0, 1);
printf("%d\n", count);
return 0;
}
```

- **Data and Results:**

Input specifies cities and travel times; output counts valid paths

- **Analysis and Inference:**

Pathfinding complexity increases with cities; multiple valid routes exist

2. Problem Statement

There are N items remaining in the company. The weight of the i-th item is 1<=i<=N and Wi
Takahashi will sell these items as D lucky bags.
He wants to minimize the variance of the total weights of the items in the lucky bags.
Here, the variance is defined as

$V = \frac{1}{D}\sum_{i=1}^{D}((x_i - \bar{x})^2)$

where x1, x2,..xD are the total weights of the items in the lucky bags, and

$\bar{x} = \frac{1}{D}(x_1 + x_2 + \ldots x_D)$

is the average of x1, x2,..,xD . Find the variance of the total weights of the items in the lucky bags when the items are divided to minimize this value.

It is acceptable to have empty lucky bags (in which case the total weight of the items in that bag is defined as *0*), but **each item must be in exactly one of the *D* lucky bags**.

Constraints

- $2 \leq D \leq N \leq 15$
- $1 \leq D \leq 10^8$
- **All input values are integers.**

Input

The input is given from Standard Input in the following format:

$N\ D$
$W_1, W_2 \ldots W_N$
Output

Print the variance of the total weights of the items in the lucky bags when the items are divided to minimize this value.
Your output will be considered correct if the absolute or relative error from the true value is at most 10^{-6}.

Sample Input 1

```
5 3
3 5 3 6 3
```

Sample Output 1

```
0.888888888888889
```

If you put the first and third items in the first lucky bag, the second and fifth items in the second lucky bag, and the fourth item in the third lucky bag, the total weight of the items in the bags are *6*, *8*, and *6*, respectively.

Then, the average weight is $\frac{1}{3}\left(6+8+6=\frac{20}{3}\right)$

and the variance is $\frac{1}{3}\left\{\left(6-\frac{20}{3}\right)^2+\left(8-\frac{20}{3}\right)^2+\left(6-\frac{20}{3}\right)^2\right\}=0.88888\ldots$ which is the minimum.

Note that multiple items may have the same weight, and that each item must be in one of the lucky bags.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_N 100
#define INF 1e18

double weights[MAX_N];
double dp[MAX_N][MAX_N];
double sum[MAX_N];

int main(){
    int N, D;
    scanf("%d %d", &N, &D);
for(int i=0; i<N; i++){
    scanf("%lf", &weights[i]);
}
```

```
    sum[0] = 0;
for(int i = 0;    i<N;  i++){
    sum[i+1] = sum[i] + weights[i];
    }
for(int  i = 0;   i<=N;  i++){
  for(int j=0;    j<=D;  j++){
      dP[i][j]  = INF;
  }
}
dP[0][0] = 0;
 for(int   j=1;    j<=D ;   j++){
    for(int   i=1;   i<=N;   i++){
  for(int   k=0;     k<i;      k++){
double   totalweight  = sum[i]- sum[k];
double   avgweight  = totalweight / (i-k);
double   variance  = (totalweight * totalweight)/
          (i-k)- avgweight  * avgweight;
dP[i][j] = fmin(dP[i][j], dP[k][j-1] +
                          variance);
    }
  }
}
```

```
printf("%.1f\n", dp[N][D]);
return 0;
}
```

- **Data/ Results:**

input consists of item weights and lucky bags, producing variance result

- **Analysis and Inference:**

minimizing variance improves distribution of weights across lucky bags.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the difference between backtracking and branch and bound?
2. List the applications of Brach and Bound approach
3. List the methods of Branch and bound
4. Define state space tree
5. Define E-Node and Live node
6. In what real-world scenarios might TSP be applicable?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) Differences: Backtracking explores all paths; Branch and Bound prunes

2) Applications: TSP, knapsack problem, job assignment, scheduling, integer programming

3) Methods: least-cost, depth-first, and breadth-first branch and bound

4) State space tree: Tree representing all possible solution paths

5) E-Node: Node being expanded; Live Node: yet-to-be-expanded node.

6) Real-world TSP scenario: Delivery routing, circuit design, manufacturing, city touring.