Extra assign 3

1) **KNAPSACK**

```
#include <iostream>
#include <vector>
using namespace std;

int knapsack(int weights[], int values[], int n, int capacity) {
   // Backtracking function to explore all subsets
   if (n == 0 || capacity == 0)
      return 0;

   // If the weight of the nth item is more than the remaining capacity
   if (weights[n - 1] > capacity)
      return knapsack(weights, values, n - 1, capacity);

   // Case 1: Exclude the current item
   int exclude = knapsack(weights, values, n - 1, capacity);

   // Case 2: Include the current item
   int include = values[n - 1] + knapsack(weights, values, n - 1, capacity - weights[n - 1]);

   // Return the maximum of including or excluding the item
   return max(exclude, include);
}

int main() {
   int values[] = {60, 100, 120};
   int weights[] = {10, 20, 30};
   int capacity = 50;
   int n = sizeof(values) / sizeof(values[0]);

   cout << "Maximum value that can be carried: " << knapsack(weights, values, n, capacity) <<
endl;

   return 0;
}
```

```
Maximum value that can be carried: 220
```

2) TRAVELLING SALESMAN

```
#include <bits/stdc++.h>
using namespace std;

// Function to find the minimum cost using backtracking
void totalCost(vector<vector<int>>& cost, vector<bool>& visited,
               int currPos, int n, int count, int costSoFar, int& ans) {

    // If all nodes are visited and there's an edge to the start node
    if (count == n && cost[currPos][0]) {
        // Update the minimum cost
        ans = min(ans, costSoFar + cost[currPos][0]);
        return;
    }

    // Try visiting each node from the current position
    for (int i = 0; i < n; i++) {
        if (!visited[i] && cost[currPos][i]) {
            // If node is not visited and has an edge
            visited[i] = true;
            totalCost(cost, visited, i, n, count + 1, costSoFar + cost[currPos][i], ans);
            visited[i] = false;
        }
    }
}

// Function to solve the TSP problem
int tsp(vector<vector<int>>& cost) {
    int n = cost.size();
    vector<bool> visited(n, false);
    visited[0] = true;  // Start from the first city

    int ans = INT_MAX;  // Initialize the answer to a large value
    totalCost(cost, visited, 0, n, 1, 0, ans);  // Start the backtracking process
```

```
    return ans;
}

int main() {
    // Distance matrix representing the cost between cities
    vector<vector<int>> cost = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    // Call the TSP function and get the result
    int res = tsp(cost);

    // Output the minimum cost of the tour
    cout << "Minimum cost of the tour: " << res << endl;

    return 0;
}
```

3) PARTITION INTO K SUBSET OF EQUAL SUM

```
#include <bits/stdc++.h>
using namespace std;

bool isKPartitionPossible(vector<int> &arr, vector<int> &subsetSum,
                          vector<bool> &taken, int target, int k,
                          int n, int currIdx, int limitIdx) {

    // If the current subset sum matches the target
    if (subsetSum[currIdx] == target) {

        // If all but one subset are filled, the
        // last subset is guaranteed to work
        if (currIdx == k - 2)
            return true;
        return isKPartitionPossible(arr, subsetSum, taken,
                                    target, k, n, currIdx + 1, n - 1);
```

```
    }

    // Try including each element in the current subset
    for (int i = limitIdx; i >= 0; i--) {

        // Skip if the element is already used
        if (taken[i])
            continue;
        int temp = subsetSum[currIdx] + arr[i];
        if (temp <= target) {

            // Only proceed if it doesn't exceed the target
            taken[i] = true;
            subsetSum[currIdx] += arr[i];
            if (isKPartitionPossible(arr, subsetSum, taken,
                                target, k, n, currIdx, i - 1))
                return true;

            // Backtrack
            taken[i] = false;
            subsetSum[currIdx] -= arr[i];
        }
    }
    return false;
}

bool isKPartitionPossible(vector<int> &arr, int k) {

    int n = arr.size(), sum = accumulate(arr.begin(), arr.end(), 0);

    // If only one subset is needed, it's always possible
    if (k == 1)
        return true;

    // Check if partition is impossible
    if (n < k || sum % k != 0)
        return false;
```

```
    int target = sum / k;
    vector<int> subsetSum(k, 0);
    vector<bool> taken(n, false);

    // Initialize first subset with the last element
    subsetSum[0] = arr[n - 1];
    taken[n - 1] = true;

    // Recursively check for partitions
    return isKPartitionPossible(arr, subsetSum, taken,
                                target, k, n, 0, n - 1);
}

int main() {
    vector<int> arr = {2, 1, 4, 5, 3, 3};
    int k = 3;

    if (isKPartitionPossible(arr, k))
        cout << "true";
    else
        cout << "false";

    return 0;
}
```

4) PALINDROMIC PARTITION

```
#include <iostream>
#include <vector>
using namespace std;

bool isPalindrome(string& str, int start, int end) {
    while (start < end) {
        if (str[start] != str[end]) return false;
        start++;
        end--;
    }
    return true;
```

```
}

void palindromicPartitionsHelper(string& str, int index, vector<string>& current) {
    if (index == str.size()) {
        for (string& s : current) {
            cout << s << " ";
        }
        cout << endl;
        return;
    }

    for (int i = index; i < str.size(); ++i) {
        if (isPalindrome(str, index, i)) {
            current.push_back(str.substr(index, i - index + 1));
            palindromicPartitionsHelper(str, i + 1, current);
            current.pop_back();
        }
    }
}

void printPalindromicPartitions(string& str) {
    vector<string> current;
    palindromicPartitionsHelper(str, 0, current);
}

int main() {
    string str = "aab";
    printPalindromicPartitions(str);
    return 0;
}
```