EAST WEST UNIVERSITY

Course Code : CSE246

Course Tittle : ALGORITHM

Section : 04

Assignment

Restaurant Order Optimization System

Submitted By

Name	ID
Ali Haidar	2022-1-60-193
Md.Tanvir Hossain	2022-1-60-196
Nuzath Tabassum Arthi	2022-1-60-185
Ishmat Zaman	2022-1-60-102
Nusaiba Rahman	2022-1-60-298

Submitted To

Redwan Ahmed Rizvee

Adjunct Lecturer

Department of Computer Science and Engineering
East West university

❖ Scenario :

Let’s suppose that there is a very busy restaurant under your management which usually has a high flow of customer orders. Your restaurant is multicultural, and there are many issues in order management and optimization that you have to overcome because offers are numerous. You aim to enhance the clients’ satisfaction respectively as well as increase the productivity with reference to these aspects:

Choice of Resize Order Placement and Calculation: The customer is presented with a price menu which applies to specific items that are in most cases selected by the customers. You are required to give a total amount for their order, including and subtracting the total amount of all the coupons which are applicable, and also tips.

Menu Creation for Profit Maximization: The goal is to build a menu that has the highest profitability in a limited period of time. In this case, it requires choosing two or more combinations of each dish based on how profitable it is and how long it takes to make, so that the menu is profitable and can be made with ease.

Table Reservation Management: Your restaurant has many tables to reserve for different occasions. You want to optimize the reservation system to maximize the number of non-overlapping reservations, and ensure that as many customers as possible can dine in your restaurant. algorithmic approach to the problem

Order selection and pricing:

Binary search: You use binary search to quickly search for an item in a menu, helping you find items more clearly and calculate their overall value.

Sorting and aggregating: Once products are selected, sorting by price helps to organize orders and can also be used for various analyses, such as product cost or price easiest to find.

The menu system to get the most out of it:

Fractional Knapsack Problem: You use the fractional knapsack algorithm to determine which dishes to include in the menu based on their profit-to-time ratio. This helps in maximizing the total profit given the time constraints for preparation. Table Reservation Management: Activity Selection Problem: To manage table reservations efficiently, you use the activity selection algorithm. This helps in scheduling the maximum number of non-overlapping reservations, optimizing the usage of tables and accommodating as many guests as possible. Implementation Outline Order Selection and Cost Calculation: Implement a binary search to find items by name. Calculate the total cost of selected items.

Handle tip calculations using the coin change problem. Menu Design for Maximum Profit: Use the fractional knapsack algorithm to select dishes that maximize profit within the available preparation time. Table Reservation Management: Apply the activity selection algorithm to determine the maximum number of non-overlapping reservations.

❖ Solution Idea :

1. Binary Search for Ingredient Lookup:

Inventory:

[{ "Beef Bhuna---", 26 }, { "Biryani------", 30 }, { "Chicken Tikka", 28 }, { "Hilsa Fish---", 27 }, { "Kacchi Birani", 33 }]

Searching for "Chicken Tikka":

Steps:

Start with the entire list: low = 0, high = 4
Calculate mid: (0 + 4) / 2 = 2
Compare "Chicken Tikka" with the middle element "Chicken Tikka"
Match found! Return index 2
Searching for "Hilsa Fish---":

Steps:

Start with the entire list: low = 0, high = 4
Calculate mid: (0 + 4) / 2 = 2
Compare "Hilsa Fish---" with "Chicken Tikka"
"Hilsa Fish---" > "Chicken Tikka", so search right half: low = 3, high = 4
Calculate new mid: (3 + 4) / 2 = 3
Compare "Hilsa Fish---" with "Hilsa Fish---"
Match found! Return index 3

2. Merge Sort for Order Sorting:

Orders: [{ "Biryani", 30 }, { "Beef Bhuna", 26 }, { "Chicken Tikka", 28 }]

Sorting Process:

Steps:

Divide the list into two halves:

Left: [{ "Biryani", 30 }]

Right: [{ "Beef Bhuna", 26 }, { "Chicken Tikka", 28 }]

Recursively sort the right half:

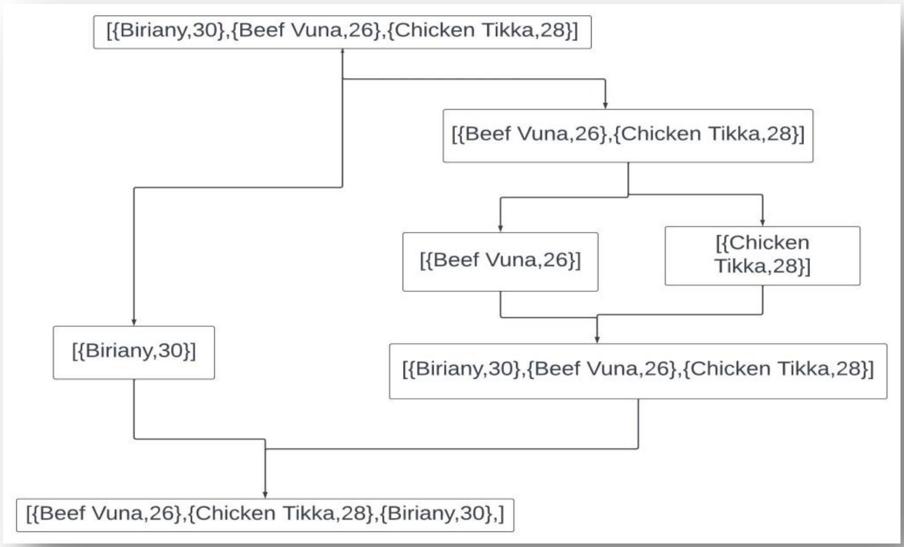
Divide: [{ "Beef Bhuna", 26 }] and [{ "Chicken Tikka", 28 }]

Merge: [{ "Beef Bhuna", 26 }, { "Chicken Tikka", 28 }]

Merge the sorted halves:

Compare "Biryani" (30) with "Beef Bhuna" (26)

Result: [{ "Beef Bhuna", 26 }, { "Chicken Tikka", 28 }, { "Biryani", 30 }]


[{Biriany,30},{Beef Vuna,26},{Chicken Tikka,28}]
[{Beef Vuna,26},{Chicken Tikka,28}]
[{Beef Vuna,26}]
[{Chicken Tikka,28}]
[{Biriany,30}]
[{Biriany,30},{Beef Vuna,26},{Chicken Tikka,28}]
[{Beef Vuna,26},{Chicken Tikka,28},{Biriany,30},]

Simulation – 1

3. Coin Change for Tip Calculation:

Coins: [1, 5, 10]

Tip Amount: 18

Steps:

Initialize dp array: [0, 19, 19, ..., 19]

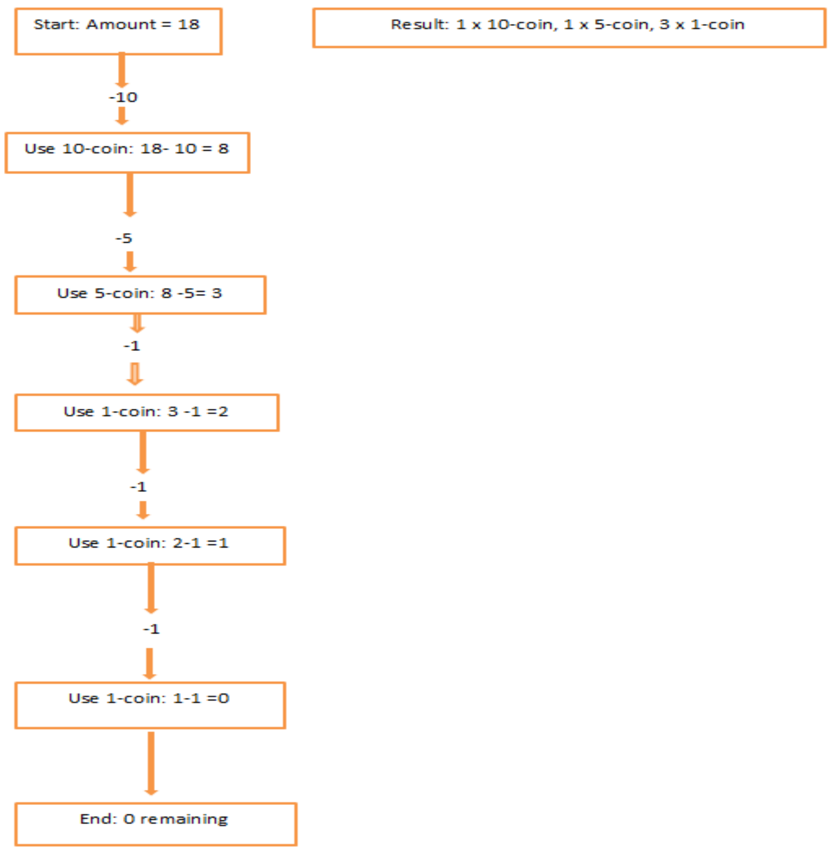
For each coin, update the dp array by checking if using the current coin minimizes the coin count:

Using coin 1: Fill dp[i] = dp[i-1] + 1 for all i = 1 to 18

Using coin 5: Update dp[5], dp[10], dp[15]

Using coin 10: Update dp[10], dp[15]

Result: dp[18] = 5 (One 10-coin, one 5-coin, and three 1-coins)



Simulation - 2

4. Fractional Knapsack for Menu Design:

Dishes (profit, time): [(60, 10), (100, 20), (120, 30)]

Max Time: 50

Steps:

Calculate profit/time ratios:

Dish 1: 60 / 10 = 6,

Dish 2: 100 / 20 = 5,

Dish 3: 120 / 30 = 4

Sort based on ratios: [{60, 10}, {100, 20}, {120, 30}]

Add dishes:

Add full Dish 1 and Dish 2: Total profit = 160

Add fraction of Dish 3: Profit = 80

Total profit: 240

Box Simulation Table

Step	Dish	Time Used	Profit Added	Time Remaining	Total Profit
1	Dish 1 (Full)	10	60	40	60
2	Dish 2 (Full)	20	100	20	160
3	Dish 3 (Fraction)	20	80	0	240

Calculate Profit-to-Time Ratios

5. Activity Selection for Table Reservations:

Reservations (start, end): [(1, 4), (2, 5), (6, ⍰, (3, 6)]

Steps:

Sort by end time: [(1, 4), (2, 5), (3, 6), (6, 8)]

Select first reservation (1, 4): Count = 1

Skip overlapping reservations (2, 5) and (3, 6)

Select non-overlapping reservation (6, 8): Count = 2

Result: 2 non-overlapping reservations

❖ Pseudo Code

```
BEGIN

  FUNCTION toLowerCase(str):
    SET lowerStr TO str
    FOR each character in lowerStr:
      CONVERT character to lowercase
    RETURN lowerStr

  FUNCTION binarySearchIngredientsByName(inventory, ingredient):
    SET low TO 0
    SET high TO length of inventory - 1
    CONVERT ingredient to lowercase
    WHILE low <= high:
      SET mid TO (low + high) / 2
```

```
        CONVERT inventory[mid] name to lowercase
        IF inventory[mid] name equals ingredient:
            RETURN mid
        ELSE IF inventory[mid] name < ingredient:
            SET low TO mid + 1
        ELSE:
            SET high TO mid - 1
    RETURN -1

FUNCTION merge(orders, l, m, r):
    SET n1 TO m - l + 1
    SET n2 TO r - m
    CREATE left array L of size n1
    CREATE right array R of size n2
    COPY elements from orders[l...m] to L
    COPY elements from orders[m+1...r] to R
    SET i, j, k TO 0
    WHILE i < n1 AND j < n2:
        IF L[i].price <= R[j].price:
            orders[k] TO L[i]
            INCREMENT i
        ELSE:
            orders[k] TO R[j]
            INCREMENT j
        INCREMENT k
    COPY remaining elements from L to orders
    COPY remaining elements from R to orders

FUNCTION mergeSort(orders, l, r):
    IF l < r:
        SET m TO (l + r) / 2
        CALL mergeSort(orders, l, m)
        CALL mergeSort(orders, m + 1, r)
        CALL merge(orders, l, m, r)

FUNCTION coinChange(coins, tipAmount):
    SET n TO length of coins
    CREATE dp array of size tipAmount + 1 with value tipAmount + 1
    SET dp[0] TO 0
    FOR i FROM 1 TO tipAmount:
        FOR each coin in coins:
            IF coin <= i:
                SET dp[i] TO minimum of dp[i] and dp[i - coin] + 1
    IF dp[tipAmount] > tipAmount:
        RETURN -1
    RETURN dp[tipAmount]

FUNCTION fractionalKnapsack(dishes, maxTime):
    SORT dishes by profit/time ratio in descending order
    SET totalProfit TO 0
    FOR each dish in dishes:
        IF maxTime >= dish.time:
            SUBTRACT dish.time from maxTime
            ADD dish.profit to totalProfit
        ELSE:
            ADD (dish.profit / dish.time) * maxTime to totalProfit
            BREAK
    RETURN totalProfit

FUNCTION activitySelection(reservations):
    SORT reservations by end time in ascending order
    SET count TO 1
    SET lastEnd TO reservations[0].end
    FOR each reservation from 1 to length of reservations:
        IF reservation.start >= lastEnd:
            INCREMENT count
            SET lastEnd TO reservation.end
    RETURN count

FUNCTION displayIngredientDetails(inventory):
    FOR each ingredient in inventory:
        PRINT ingredient name, price, and expiry day

FUNCTION displayIngredientNames(inventory):
    FOR each ingredient in inventory:
        PRINT ingredient name
```

```
FUNCTION itemSelectionAndDisplay(inventory):
  CALL displayIngredientDetails(inventory)
  READ numItems
  CREATE selectedIndices array of size numItems
  FOR i FROM 0 TO numItems - 1:
    READ serialNumber
    SET selectedIndices[i] TO serialNumber - 102
  SET totalCost TO 0
  FOR each index in selectedIndices:
    IF index is valid:
      PRINT ingredient details
      ADD price to totalCost
    ELSE:
      PRINT "Invalid item number"
  PRINT total cost
  READ orderChoice
  IF orderChoice is 'y' or 'Y':
    PRINT "Order placed successfully"
    CREATE selectedItems array
    FOR each index in selectedIndices:
      IF index is valid:
        ADD item to selectedItems
    CALL mergeSort(selectedItems)
    PRINT sorted selected items
  ELSE:
    PRINT "Order not placed"

FUNCTION tipCalculation():
  READ numCoins
  CREATE coins array of size numCoins
  FOR i FROM 0 TO numCoins - 1:
    READ coin denomination
  READ tipAmount
  CALL coinChange(coins, tipAmount) TO result
  IF result != -1:
    PRINT "Minimum coins required for tip" and result
  ELSE:
    PRINT "It's not possible to provide the exact tip amount"
  FUNCTION menuDesign():
    READ numDishes
    CREATE dishes array of size numDishes
    FOR i FROM 0 TO numDishes - 1:
      READ profit and time for each dish
    READ maxTime
    CALL fractionalKnapsack(dishes, maxTime) TO maxProfit
    PRINT maximum profit

  FUNCTION tableReservation():
    READ numReservations
    CREATE reservations array of size numReservations
    FOR i FROM 0 TO numReservations - 1:
      READ start and end time for each reservation
    CALL activitySelection(reservations) TO maxActivities
    PRINT maximum number of non-overlapping reservations

  MAIN:
    CREATE inventory with dish names and prices
    SET choice TO 0
    WHILE choice != 0:
      PRINT menu options
      READ choice
      SWITCH choice:
        CASE 1:
          CALL displayIngredientDetails(inventory)
        CASE 2:
          CALL itemSelectionAndDisplay(inventory)
        CASE 3:
          CALL menuDesign()
        CASE 4:
          CALL tableReservation()
        CASE 0:
          PRINT "Exiting the program"
        DEFAULT:
          PRINT "Invalid choice. Please try again."
END
```

❖ Time Complexity

1) Binary Search for Ingredients by Name

Binary Search Process:

In binary search, the array is divided into two halves at each step. Each step reduces the problem size from n to n / 2, and this halving continues until we have a single element left.

Steps Calculation:

Initially, the array size is n. After the first step, the size becomes n / 2. After the second step, the size becomes n / 4. After the third step, the size becomes n / 8, and so on. The size of the array at step k is n / 2^k.

When does it stop?:

The algorithm stops when n / 2^k = 1.

Solving for k gives:

n 2 k = 1
⟹ n = 2 k
⟹ k = log 2 n 2 k n =1
⟹ n=2 k
⟹ k=log 2 n

Hence, the binary search performs k = \log_2 n steps in the worst case.

Time Complexity: $O(\log n)$

2) Merge Sort for Sorting Orders

Complexity Analysis:

The complexity is often represented as : $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Time Complexity: O(n log n)
where n is the number of orders. Merge sort divides the array into halves and sorts each half recursively, which takes O(log n) time for the divisions and O(n) time for merging, resulting in O(n log n) overall.

Space Complexity: O(n) due to the auxiliary space required for the temporary arrays used in merging.

3. Coin Change Problem

Complexity Analysis:

Time Complexity: O(n * m)
where n is the number of coin types and m is the tip amount. The nested loops run through all coin types for each amount from 1 to tipAmount, resulting in O(n * m) time complexity.

Space Complexity: O(m) due to the dp array used to store intermediate results.

4. Fractional Knapsack for Menu Design

Complexity Analysis:

Time Complexity: O(n log n)
where n is the number of dishes. Sorting the dishes based on profit-to-time ratio takes O(n log n), and iterating through the sorted list takes O(n). Thus, the overall time complexity is dominated by the sorting step.

Space Complexity: O(1) as the algorithm uses a constant amount of extra space beyond the input data.

5. Activity Selection for Table Reservations

Complexity Analysis:

Time Complexity: O(n log n)
where n is the number of reservations. Sorting the reservations based on end times takes O(n log n), and iterating through the sorted list takes O(n). Thus, the overall time complexity is dominated by the sorting step.

Space Complexity: O(1) as the algorithm uses a constant amount of extra space beyond the input data.

Summary of Time Complexity Calculations

Function	Time Complexity
Binary Search Ingredients By Name	0(log n)
Merge Sort (for orders)	0(n log n)
Coin Change	0(n * k)
Fractional Knapsack	0(n log n)
Activity Selection	0(n log n)
Item Selection And Display	0(n + m log m)

❖ Code :

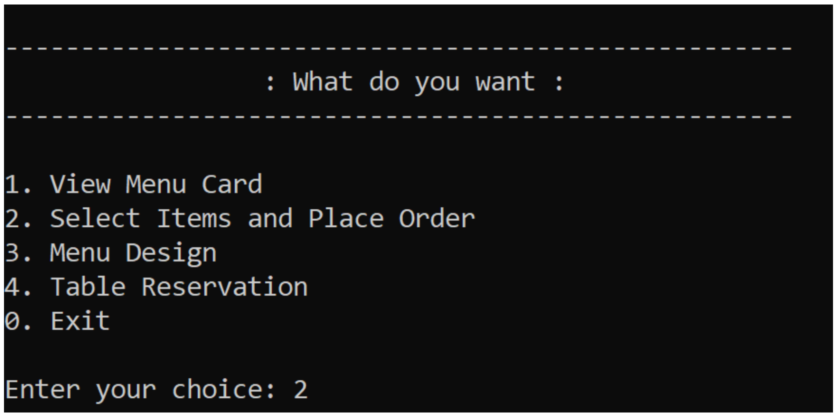
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cctype>
#include <limits>
#include <iomanip>  // For formatting output

using namespace std;

// Helper function to convert a string to lowercase
string toLowerCase(const string& str) {
   string lowerStr = str;
   transform(lowerStr.begin(), lowerStr.end(), lowerStr.begin(), ::tolower);
   return lowerStr;
}

// Function to perform binary search for ingredients based on name (case-
insensitive)
int binarySearchIngredientsByName(const vector<pair<string, int>>& inventory,
const string& ingredient) {
   int low = 0, high = inventory.size() - 1;
   string ingredientLower = toLowerCase(ingredient);  // Convert the input to
lowercase
```

```
    while (low <= high) {
        int mid = low + (high - low) / 2;
        string midIngredientLower = toLowerCase(inventory[mid].first);  // Compare
in lowercase
        if (midIngredientLower == ingredientLower)
            return mid;
        else if (midIngredientLower < ingredientLower)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

// Merge function for merge sort
void merge(vector<pair<string, int>>& orders, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    vector<pair<string, int>> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = orders[l + i];
    for (int i = 0; i < n2; i++) R[i] = orders[m + 1 + i];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i].second <= R[j].second)
            orders[k++] = L[i++];
        else
            orders[k++] = R[j++];
    }

    while (i < n1)
        orders[k++] = L[i++];
    while (j < n2)
        orders[k++] = R[j++];
```

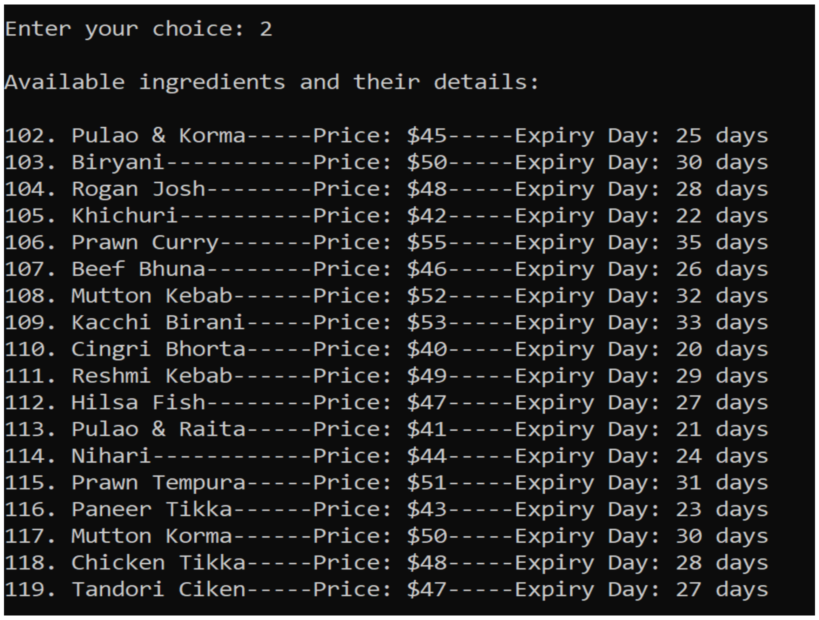
```
}

// Merge sort for sorting orders
void mergeSort(vector<pair<string, int>>& orders, int l, int r) {
   if (l < r) {
      int m = l + (r - l) / 2;
      mergeSort(orders, l, m);
      mergeSort(orders, m + 1, r);
      merge(orders, l, m, r);
   }
}

// Coin change for calculating tips
int coinChangeHelper(const vector<int>& coins, int amount,vector<int>& memo)
{
   // Base cases
   if (amount == 0) return 0;
   if (amount < 0) return -1;

   if (memo[amount] != -1) return memo[amount];

   int minCoins = INT_MAX;
   for (int coin : coins) {
      int res = coinChangeHelper(coins, amount - coin, memo);
      if (res != -1 && res + 1 < minCoins) {
         minCoins = res + 1;
      }
   }

   memo[amount] = (minCoins == INT_MAX) ? -1 : minCoins;
   return memo[amount];
}

int coinChange(const vector<int>& coins, int amount) {
   vector<int> memo(amount + 1, -1);
```

```
    return coinChangeHelper(coins, amount, memo);
}

// Fractional knapsack for menu design (profit maximization)
double fractionalKnapsack(vector<pair<double, double>>& dishes, double
maxTime) {
    sort(dishes.begin(), dishes.end(), [](const pair<double, double>& a, const
pair<double, double>& b) {
        return (a.first / a.second) > (b.first / b.second);
    });

    double totalProfit = 0.0;
    for (auto& dish : dishes) {
        if (maxTime >= dish.second) {
            maxTime -= dish.second;
            totalProfit += dish.first;
        } else {
            totalProfit += (dish.first / dish.second) * maxTime;
            break;
        }
    }
    return totalProfit;
}

// Activity scheduling for table reservations
int activitySelection(vector<pair<int, int>>& reservations) {
    sort(reservations.begin(), reservations.end(), [](const pair<int, int>& a, const
pair<int, int>& b) {
        return a.second < b.second;
    });

    int count = 1, lastEnd = reservations[0].second;
    for (int i = 1; i < reservations.size(); ++i) {
        if (reservations[i].first >= lastEnd) {
            count++;
            lastEnd = reservations[i].second;
        }
    }
    return count;
}

// Function to display the names, prices, and expiry dates of the ingredients
void displayIngredientDetails(const vector<pair<string, int>>& inventory) {
    cout << "\nAvailable ingredients and their details:\n\n";
    for (size_t i = 0; i < inventory.size(); ++i) {
        cout << setw(3) << setfill('0') << (102 + i) << ". " << inventory[i].first
             << "-----Price: $" << (inventory[i].second + 20)
             << "-----Expiry Day: " << inventory[i].second << " days"<< endl;
    }
}

// Function to display the names of the ingredients
void displayIngredientNames(const vector<pair<string, int>>& inventory) {
    cout << "\nAvailable ingredients:\n";
    for (size_t i = 0; i < inventory.size(); ++i) {
        cout << setw(3) << setfill('0') << (102 + i) << ". " << inventory[i].first << endl;
    }
}

void itemSelectionAndDisplay(vector<pair<string, int>>& inventory) {
    displayIngredientDetails(inventory);

    int numItems;
    cout << "\nHow many food item you want :";
    cin >> numItems;

    vector<int> selectedIndices(numItems);
    cout << "Enter the serial number from menu card : ";
    for (int i = 0; i < numItems; ++i) {
        int serialNumber;
        cin >> serialNumber;
        selectedIndices[i] = serialNumber - 102; // Convert to 0-based index
    }

    double totalCost = 0.0;
    cout << "\nSelected items and their details:\n";
    for (int index : selectedIndices) {
        if (index >= 0 && index < inventory.size()) {
            cout << inventory[index].first << "-----Price: $" << (inventory[index].second
+ 20) << "-----Expiry Day: " << inventory[index].second << endl;
            totalCost += inventory[index].second + 20;
        } else {
            cout << "Invalid item number.\n";
        }
    }

    cout << "\nTotal cost of selected items: $" << totalCost << endl;

    char orderChoice;
    cout << "\nDo you want to place an order for these items? (y/n): ";
    cin >> orderChoice;

    if (orderChoice == 'y' || orderChoice == 'Y') {
        cout << "\nOrder placed successfully!\n";

        // Now handle the tip calculation
        //tipCalculation();

        // Sort selected items based on their price for ordering
        vector<pair<string, int>> selectedItems;
        for (int index : selectedIndices) {
            if (index >= 0 && index < inventory.size()) {
                selectedItems.emplace_back(inventory[index].first,
inventory[index].second);
            }
        }
        mergeSort(selectedItems, 0, selectedItems.size() - 1);
        cout << "\nSorted selected items based on price:\n";
        for (const auto& item : selectedItems) {
            cout << item.first << ": $" << (item.second + 20) << endl;
        }
    } else {
        cout << "Order not placed.\n";
    }
}

void tipCalculation() {
    int numCoins;
    cout << "\nEnter the number of coin types for paying the bill: ";
    cin >> numCoins;
    vector<int> coins(numCoins);
    cout << "Enter the coin denominations:" << endl;
    for (int i = 0; i < numCoins; ++i) {
        cin >> coins[i];
    }
    int tipAmount;
    cout << "Enter the tip amount: ";
    cin >> tipAmount;
    int result = coinChange(coins, tipAmount);
    if (result != -1)
        cout << "\nMinimum coins required for tip: " << result << endl;
    else
        cout << "\nIt's not possible to provide the exact tip amount with given
coins.\n";
}

void menuDesign() {
    int numDishes;
    cout << "Enter the number of dishes: ";
    cin >> numDishes;
    };

    vector<pair<double, double>> dishes(numDishes);
    cout << "Enter the profit and time for each dish:" << endl;
    for (int i = 0; i < numDishes; ++i) {
        cin >> dishes[i].first >> dishes[i].second;
    }
    double maxTime;
    cout << "Enter the maximum available time: ";
    cin >> maxTime;
    double maxProfit = fractionalKnapsack(dishes, maxTime);
    cout << "\nMaximum profit for menu design: " << maxProfit << endl;
}

void tableReservation() {
    int numReservations;
    cout << "Enter the number of reservations: ";
    cin >> numReservations;
    vector<pair<int, int>> reservations(numReservations);
    cout << "Enter the start and end time for each reservation:" << endl;
    for (int i = 0; i < numReservations; ++i) {
        cin >> reservations[i].first >> reservations[i].second;
    }
    int maxActivities = activitySelection(reservations);
    cout << "\nMaximum number of non-overlapping reservations: " <<
maxActivities << endl;
}

int main() {
    vector<pair<string, int>> inventory = {
        {"Pulao & Korma", 25}, {"Biryani------", 30}, {"Rogan Josh---", 28},
        {"Khichuri-----", 22}, {"Prawn Curry--", 35}, {"Beef Bhuna---", 26},
        {"Mutton Kebab-", 32}, {"Kacchi Birani", 33}, {"Cingri Bhorta", 20},
        {"Reshmi Kebab-", 29}, {"Hilsa Fish---", 27}, {"Pulao & Raita", 21},
        {"Nihari-------", 24}, {"Prawn Tempura", 31}, {"Paneer Tikka-", 23},
        {"Mutton Korma-", 30}, {"Chicken Tikka", 28}, {"Tandori Ciken", 27}
    };
    int choice;
    do {
        cout << "\n---------------------------------------------------" << endl ;
        cout << "                : What do you want : \n"                  ;
        cout << "---------------------------------------------------\n" <<endl;
        cout << "1. View Menu Card \n";
        cout << "2. Select Items and Place Order\n";
        cout << "3. Menu Design\n";
        cout << "4. Table Reservation\n";
        cout << "0. Exit\n";
        cout << "\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                displayIngredientDetails(inventory);
                break;
            case 2:
                itemSelectionAndDisplay(inventory);
                break;
            case 3:
                menuDesign();
                break;
            case 4:
                tableReservation();
                break;
            case 0:
                cout << "Exiting the program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != 0);
    return 0;
}
```

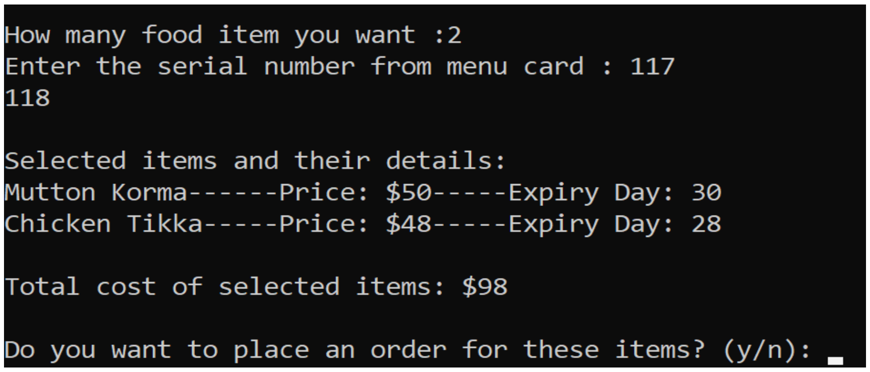
❖ OutPut :



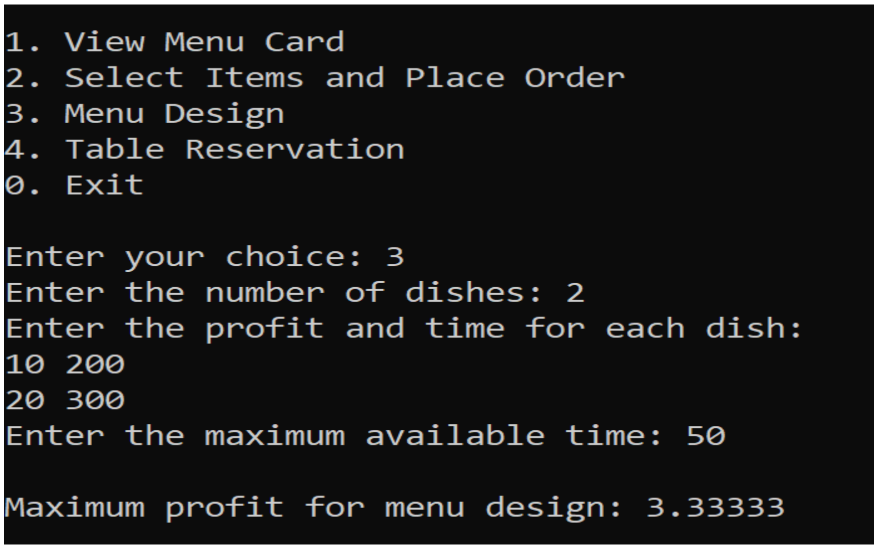
```
----------------------------------------------------
                  : What do you want :
----------------------------------------------------

1. View Menu Card
2. Select Items and Place Order
3. Menu Design
4. Table Reservation
0. Exit

Enter your choice: 2
```



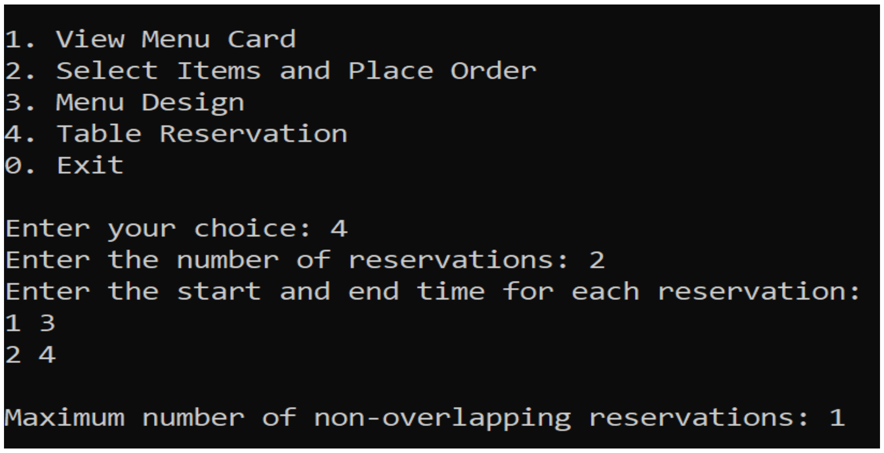
```
Enter your choice: 2

Available ingredients and their details:

102. Pulao & Korma-----Price: $45-----Expiry Day: 25 days
103. Biryani-----------Price: $50-----Expiry Day: 30 days
104. Rogan Josh--------Price: $48-----Expiry Day: 28 days
105. Khichuri----------Price: $42-----Expiry Day: 22 days
106. Prawn Curry-------Price: $55-----Expiry Day: 35 days
107. Beef Bhuna--------Price: $46-----Expiry Day: 26 days
108. Mutton Kebab------Price: $52-----Expiry Day: 32 days
109. Kacchi Birani-----Price: $53-----Expiry Day: 33 days
110. Cingri Bhorta-----Price: $40-----Expiry Day: 20 days
111. Reshmi Kebab------Price: $49-----Expiry Day: 29 days
112. Hilsa Fish--------Price: $47-----Expiry Day: 27 days
113. Pulao & Raita-----Price: $41-----Expiry Day: 21 days
114. Nihari------------Price: $44-----Expiry Day: 24 days
115. Prawn Tempura-----Price: $51-----Expiry Day: 31 days
116. Paneer Tikka------Price: $43-----Expiry Day: 23 days
117. Mutton Korma------Price: $50-----Expiry Day: 30 days
118. Chicken Tikka-----Price: $48-----Expiry Day: 28 days
119. Tandori Ciken-----Price: $47-----Expiry Day: 27 days
```



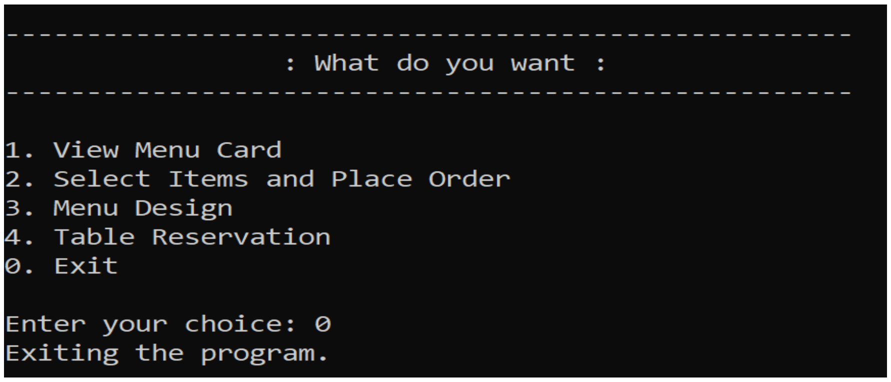
```
How many food item you want :2
Enter the serial number from menu card : 117
118

Selected items and their details:
Mutton Korma------Price: $50-----Expiry Day: 30
Chicken Tikka-----Price: $48-----Expiry Day: 28

Total cost of selected items: $98

Do you want to place an order for these items? (y/n): _
```



```
1. View Menu Card
2. Select Items and Place Order
3. Menu Design
4. Table Reservation
0. Exit

Enter your choice: 3
Enter the number of dishes: 2
Enter the profit and time for each dish:
10 200
20 300
Enter the maximum available time: 50

Maximum profit for menu design: 3.33333
```



```
1. View Menu Card
2. Select Items and Place Order
3. Menu Design
4. Table Reservation
0. Exit

Enter your choice: 4
Enter the number of reservations: 2
Enter the start and end time for each reservation:
1 3
2 4

Maximum number of non-overlapping reservations: 1
```



```
--------------------------------------------------
                : What do you want :
--------------------------------------------------

1. View Menu Card
2. Select Items and Place Order
3. Menu Design
4. Table Reservation
0. Exit

Enter your choice: 0
Exiting the program.
```

--- The End ---