Project Code

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

struct Product {
   string name;
   int price;
};

// Quick Sort by product name
int partition(vector<Product>& products, int low, int high) {
   string pivot = products[high].name;
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (products[j].name < pivot) {
         i++;
         swap(products[i], products[j]);
      }
   }
   swap(products[i + 1], products[high]);
   return i + 1;
}

void quickSort(vector<Product>& products, int low, int high) {
   if (low < high) {
      int pi = partition(products, low, high);
      quickSort(products, low, pi - 1);
      quickSort(products, pi + 1, high);
   }
}

// Binary Search by product name
int binarySearch(const vector<Product>& products, const string& key) {
   int low = 0, high = (int)products.size() - 1;
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (products[mid].name == key)
         return mid;
      else if (products[mid].name < key)
         low = mid + 1;
      else
         high = mid - 1;
   }
```

```
    return -1;
}

// Greedy Coin Change
void giveChange(int change, const vector<int>& coins) {
    cout << "Returning change: " << change << endl;
    for (int coin : coins) {
        int count = change / coin;
        if (count > 0) {
            cout << "Tk " << coin << " x " << count << endl;
            change -= coin * count;
        }
    }
    if (change > 0)
        cout << "Cannot return exact change with available coins.\n";
}

int main() {
    vector<Product> beverages = {
    {"Mango Juice", 45}, {"Apple Juice", 50}, {"Mozo", 45}, {"StarShip", 30},
    {"Drinko", 10}, {"Max Cola", 20}, {"Sprite", 40}, {"Fanta", 35},
    {"Pepsi", 25}, {"7Up", 30}};

    vector<Product> chips = {
    {"Lays", 30}, {"Pringles", 60}, {"Bingo", 25}, {"Aluz", 20},
    {"Kurkure", 15}, {"Sun", 10}, {"Cheetos", 35}, {"Haldiram", 25},
    {"Ruffles", 40}, {"Uncle Chips", 20}};

    vector<Product> chocolates = {
    {"Dairy Milk", 40}, {"Perk Chocolate", 35}, {"Choco Pie", 30}, {"Kitkat", 30},
    {"Silk", 40}, {"Hershey", 30}, {"Snickers", 45}, {"Galaxy", 50},
    {"Toblerone", 60}, {"Bounty", 35}};
    vector<int> coins = {500,100, 50, 10, 5};

    vector<Product> cart;
    int totalPrice = 0;

    while (true) {
        cout << "\n=== Smart Vending Machine ===\n";
        cout << "1. Beverages\n2. Chips\n3. Chocolates\n4. Checkout/Exit\n";
        cout << "Enter choice (1-4): ";
        int choice; cin >> choice; cin.ignore();

        vector<Product> category;
```

```
        if (choice == 1)
            category = beverages;
         else if (choice == 2)
             category = chips;
         else if (choice == 3)
            category = chocolates;
        else if (choice == 4) {
            if (cart.empty()) {
                cout << "No items in cart. Exiting. Thank you!\n";
                break;
            }
            cout << "\nYour cart:\n";
            for (const auto& p : cart)
                cout << "- " << p.name << " (Tk " << p.price << ")\n";
            cout << "Total: Tk " << totalPrice << "\nInsert money (Rs): ";
            int paid; cin >> paid;

            if (paid < totalPrice) {
                cout << "Not enough money. Transaction cancelled.\n";
            } else {
                // Time coin change
                clock_t startChange = clock();
                giveChange(paid - totalPrice, coins);
                clock_t endChange = clock();

                double changeDuration = double(endChange - startChange) /
CLOCKS_PER_SEC;
                cout << "Coin Change calculation took " << changeDuration << "
seconds.\n";

                cout << "Thank you for your purchase!\n";
                cart.clear();
                totalPrice = 0;
            }
            continue;
        } else {
            cout << "Invalid choice.\n";
            continue;
        }

        // Timing Quick Sort
        clock_t startSort = clock();
        quickSort(category, 0, (int)category.size() - 1);
        clock_t endSort = clock();
```

```
        double sortDuration = double(endSort - startSort) / CLOCKS_PER_SEC;
        cout << "\nProducts sorted by name:\n";
        for (const auto& p : category)
            cout << "- " << p.name << " (Tk " << p.price << ")\n";
        cout << "Quick Sort took " << sortDuration << " seconds.\n";

        cout << "Enter exact product name to add to cart (or 'back' to return): ";
        string productName; getline(cin, productName);
        if (productName == "back")
            continue;

        // Timing Binary Search
        clock_t startSearch = clock();
        int idx = binarySearch(category, productName);
        clock_t endSearch = clock();

        double searchDuration = double(endSearch - startSearch) /
CLOCKS_PER_SEC;

        if (idx == -1) {
            cout << "Product not found.\n";
        } else {
            cart.push_back(category[idx]);
            totalPrice += category[idx].price;
            cout << productName << " added to cart. Total: Tk " << totalPrice << endl;
        }
        cout << "Binary Search took " << searchDuration << " seconds.\n";
    }

    return 0;
}
```

Problem Statement

This project simulates a Smart Vending Machine that allows users to browse and purchase products such as beverages, chips, and chocolates. The machine automatically sorts product lists, enables efficient search by product name, and calculates optimal change using a greedy approach. The real-world context is a self-service automated kiosk used in public areas like offices, malls, and stations.

The end-user interacts with the system to:

- Browse available items by category.
- Search and select specific products.
- View the cart and checkout.
- Insert money and receive change.

High-Level Breakdown

1. Product Sorting

Objective: Sort all products alphabetically by name within each category (e.g., Beverages, Chips, Chocolates).
Algorithm Used: Quick Sort
Purpose: Makes it easier to locate products efficiently before performing a search.

---

2. Product Search

Objective: Search for a user-selected product by its name.
Algorithm Used: Binary Search
Purpose: Provides fast product lookup in sorted lists.

---

3. Cart Management

Objective: Allow users to add selected products to a virtual cart and maintain a running total of the price.
Functionality:

- Add item

- Display cart items
- Calculate total cost

---

4. Checkout & Payment Handling

Objective: Collect payment from the user and validate it against the total cart value.
Functionality:

- Compare paid amount with total
- Proceed to change calculation if payment is valid

---

5. Change Return System

Objective: Return the optimal number of coins as change to the user.
Algorithm Used: Greedy Coin Change Algorithm
Purpose: Ensure minimal number of coins is returned using available denominations.

Algorithm Mapping

1. Inventory Sorting

- **Algorithm Used: Quick Sort**
- **Subproblem:**
  In a vending machine, a user must browse a cleanly arranged list of items. Since items (beverages, chips, chocolates) are stored as unordered vectors, we sort the products alphabetically by name before presenting them for selection.
- **Purpose:**
  To enhance the user experience by displaying products in alphabetical order, and to prepare the data for binary search (which requires sorted input).
- **Input Source:**
  A vector of (Product) structs representing a selected category (beverages, chips, or chocolates), triggered by the user's category choice.
- **Output Usage:**
  The same vector, now sorted in-place, used both for display and for passing into the binary search algorithm.

2. Product Selection/Search

- **Algorithm Used: Binary Search**
- **Subproblem:**
  After viewing the sorted product list, the user must **enter the exact product name** they wish to buy. The system needs to **quickly verify if the product exists** in the sorted list.
- **Purpose:**
  To perform a **fast lookup** in the sorted category using the user-input name. This minimizes search time compared to linear search.
- **Input Source:**
  - A **sorted vector** from quickSort.
  - A **string** input by the user (product name).
- **Output Usage:**
  - Index of the product in the category (used to access product.price).
  - Or -1 if the product is not found.

---

3. Change Dispensing

- **Algorithm Used: Greedy Coin Change Algorithm**
- **Subproblem:**
  After checkout, if the user provides more money than required, the system must **return the correct change using the fewest possible coins**.
- **Purpose:**
  To simulate a real-world vending machine's behavior in efficiently returning change using available coin denominations, minimizing the total number of coins.
- **Input Source:**
  - change = paid - totalPrice
  - Vector of coin denominations {500, 100, 50, 10, 5}
- **Output Usage:**
  Printed breakdown of coins to return, or a warning if exact change is not possible.

---

4. Time Tracking

- **Used: CPU Clock Timing with <ctime>**
- **Subproblem:**
  The developer needs to measure and display how long each algorithm (sort, search, change) takes, helping to analyze performance.
- **Purpose:**
  For algorithmic performance analysis, especially useful for reporting and comparing algorithm efficiency.
- **Input Source:**
  clock() values before and after each function.

- **Output Usage:**
  Time (in seconds) printed to the console.

---

5. Cart & Transaction Management

- **Algorithm Used:** Not based on a formal algorithm, but on standard data structure manipulation (vector<Product>).
- **Subproblem:**
  To **track selected products** and calculate the **running total price** before final checkout.
- **Purpose:**
  To simulate a real shopping cart system within the vending machine logic.
- **Input Source:**
  Result from binary search — selected product.
- **Output Usage:**
  Cart contents and total price passed to checkout logic.

Algorithm Details

Algorithm 1: Quick Sort

- **Purpose in this Project:**
  Quick Sort is used to sort product lists alphabetically by name within each category (e.g., beverages, chips, chocolates). This improves user navigation and ensures binary search can work correctly, since it requires a sorted list.
- **Input:**
  - **Format:** vector<Product>& — a vector of structs each containing name and price.
  - **Source:** Category vectors like beverages, chips, or chocolates, triggered by user input.
  - **Constraints:** String comparison is used (product.name), which can be case-sensitive.
- **Output:**
  - A sorted vector of Product structs, sorted alphabetically by the name field.
  - Returned in-place (in the original vector).
- **Theoretical Time Complexity:**
  - **Best Case:** O(n log n) — pivot divides the array evenly.
  - **Average Case:** O(n log n) — on randomized or unsorted input.
  - **Worst Case:** $O(n^2)$ — occurs when pivot is always the smallest or largest (e.g., already sorted array without random pivot).

    - **Justification:** Strings are compared using lexicographic order. Time depends on both number of elements and average string length.
- **Memory Usage:**
    - Stack memory due to recursion (O(log n) average, O(n) worst case).
    - In-place sorting, no extra heap memory used.
- **Implementation Notes:**
    - Pivot is always chosen as the **last element**.
    - No randomization applied, so worst-case behavior is possible on sorted input.
    - Optimizations like **tail-call elimination** or **median-of-three** pivot selection not yet applied.

Algorithm 2: Binary Search

- **Purpose in this Project:**
  After sorting, binary search is used to **efficiently locate a product** entered by the user by name. It ensures quick lookup within sorted lists.
- **Input:**
    - **Format:** const vector<Product>& and string key
    - **Source:** Sorted product category (after Quick Sort), and user-entered product name.
    - **Constraints:** Requires the input list to be sorted; comparison is case-sensitive.
- **Output:**
    - Index of the product in the vector if found.
    - -1 if the product is not found.
- **Theoretical Time Complexity:**
    - **Best Case:** O(1) — product is at mid index.
    - **Average Case:** O(log n) — halving search range each iteration.
    - **Worst Case:** O(log n) — product is not found after full halving.
    - **Justification:** Operates on a sorted list, so performance is logarithmic regardless of list size.
- **Memory Usage:**
    - Constant space O(1) — no additional memory required.
    - Operates using index variables and a loop.
- **Implementation Notes:**
    - Iterative version used for simplicity and efficiency.
    - No case-insensitive comparison — so "pepsi" won’t match "Pepsi" unless exact.
    - Input validation for empty strings not yet implemented.

Algorithm 3: Greedy Coin Change

- **Purpose in this Project:**
  This algorithm calculates **the minimum number of coins needed to return change** to the user after payment. It simulates realistic vending machine behavior.
- **Input:**

    - **Format:** Integer value of change, and vector of coin denominations (e.g., {500, 100, 50, 10, 5})
    - **Source:** User payment (paid - totalPrice) and predefined coin vector.
    - **Constraints:** Assumes coin denominations are sorted in descending order; change may not be possible in exact form.
- **Output:**
    - Console output displaying coin breakdown (e.g., “Tk 50 x 2”).
    - Message if exact change is not possible.
- **Theoretical Time Complexity:**
    - **Best / Average / Worst Case:** O(k)
        - Where k = number of coin types (constant, usually ≤ 10).
    - **Justification:** Greedy checks each coin type once; independent of change amount size.
- **Memory Usage:**
    - Constant O(1) — uses integer counters, no dynamic structures.
- **Implementation Notes:**
    - Coin array is **hardcoded in descending order**.
    - Cannot return change like Tk 3 if denominations like 1 or 2 are missing.

Runtime Measurement & Graphical Analysis

1. Method Used

- **Tool**:
  The performance of the algorithms was measured using the clock() function from the <ctime> library in C++. This function provides processor time consumed by the program, allowing us to estimate the execution time of specific code blocks.
- **Measurement Range (Which parts of the code?)**:
    - **Quick Sort**:
      The time taken to sort the selected product category alphabetically by name was measured. This timing includes only the sorting function:

```
// Timing Quick Sort
clock_t startSort = clock();
quickSort(category, 0, (int)category.size() - 1);
clock_t endSort = clock();
```

    - **Binary Search**:
      The time taken to search for a product name in the sorted list was measured:

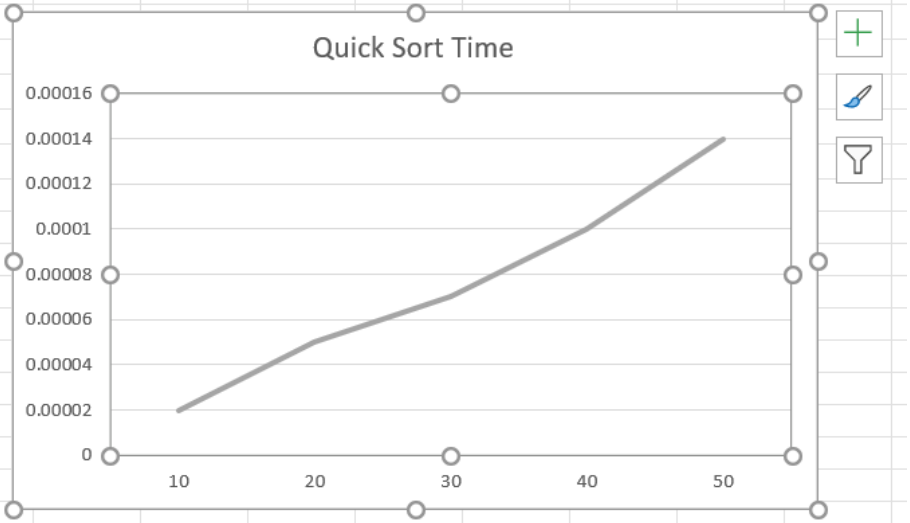
```
// Timing Binary Search
clock_t startSearch = clock();
int idx = binarySearch(category, productName);
clock_t endSearch = clock();
```

- **Greedy Coin Change**:
  The time required to calculate and display the change using a greedy algorithm was measured:

```
// Time coin change
clock_t startChange = clock();
giveChange(paid - totalPrice, coins);
clock_t endChange = clock();
```

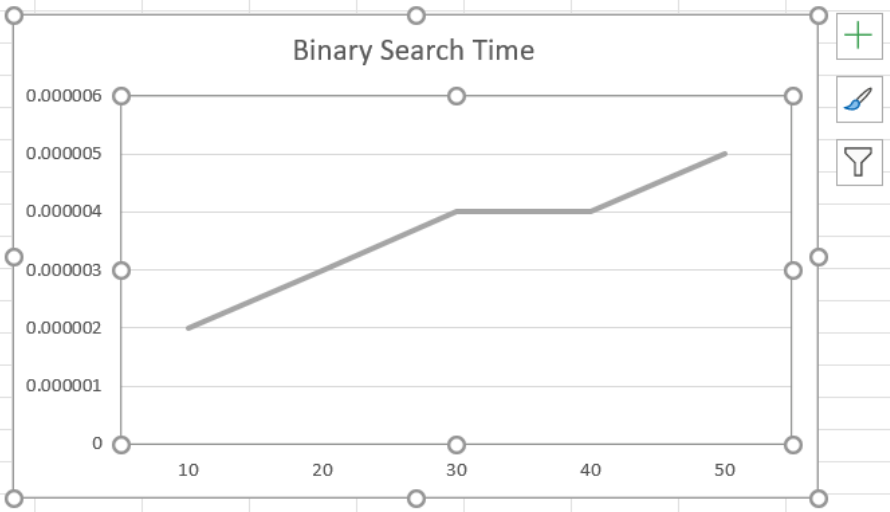
2. Individual Algorithm Graphs:

Input	Runtime
10	0.00002
20	0.00005
30	0.00007
40	0.0001
50	0.00014



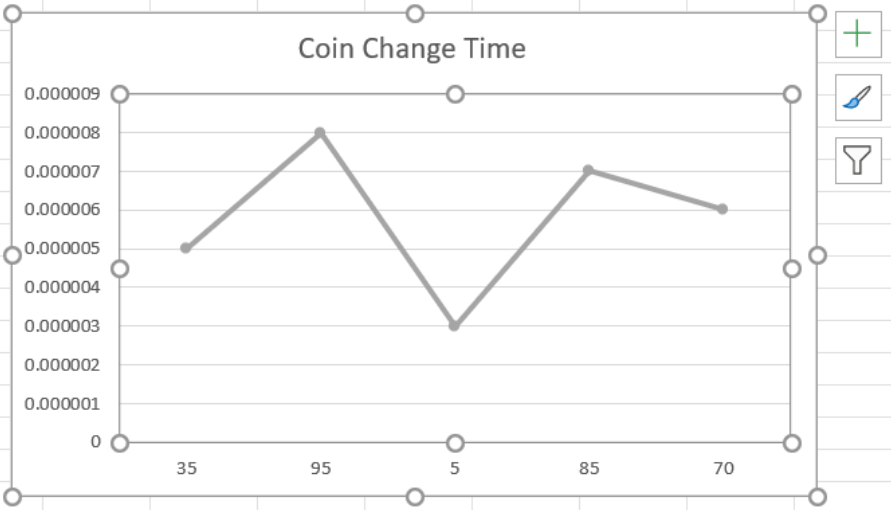
Observation: The runtime increases steadily with input size.
Shape: Linear

Input	Runtime
10	0.000002
20	0.000003
30	0.000004
40	0.000004
50	0.000005



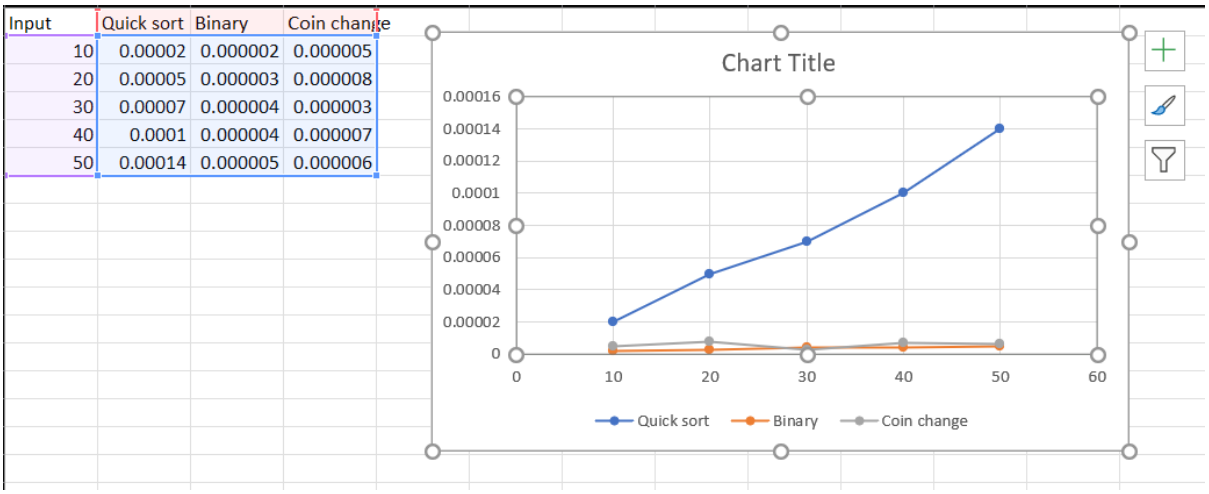
Observation: The runtime increases very slowly as input size grows, showing minimal change.
Shape: Logarithmic – (characteristic of binary search, where performance improves by halving the search space.)

Input	Runtime
35	0.000005
95	0.000008
5	0.000003
85	0.000007
70	0.000006



Observation: The runtime fluctuates with varying input sizes, showing no consistent pattern.
Shape: Irregular

3. Combined Graph:



Input	Quick sort	Binary	Coin change
10	0.00002	0.000002	0.000005
20	0.00005	0.000003	0.000008
30	0.00007	0.000004	0.000003
40	0.0001	0.000004	0.000007
50	0.00014	0.000005	0.000006

Observation:

The Quick Sort line (blue) increases steadily and significantly as the input size increases. Binary Search (orange) and Coin Change (grey) have much flatter curves, indicating much lower runtimes overall.

Identified Bottleneck:

Quick Sort is the bottleneck algorithm – it has the highest runtime among the three, especially as the input size increases. It dominates the overall performance.

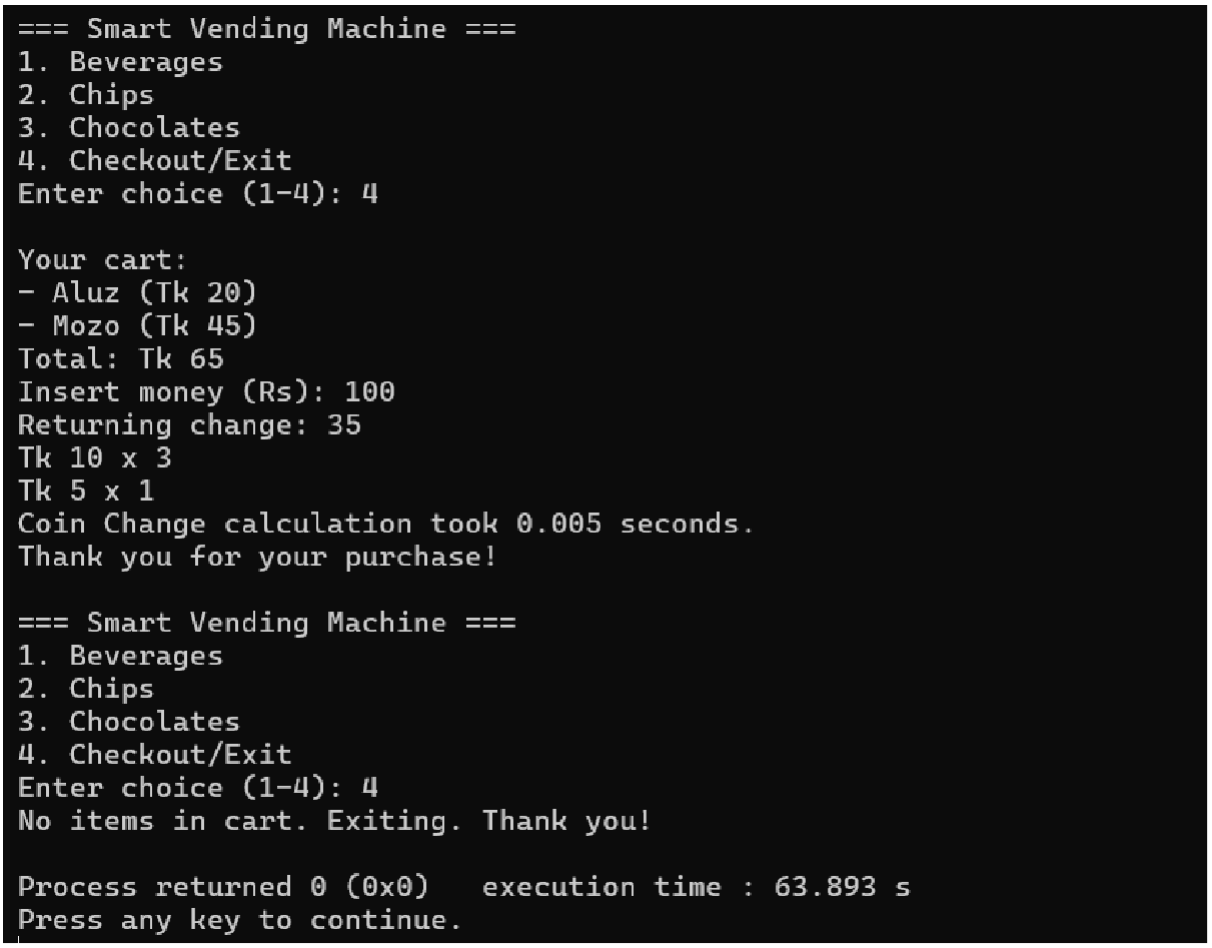
Output:

```
=== Smart Vending Machine ===
1. Beverages
2. Chips
3. Chocolates
4. Checkout/Exit
Enter choice (1-4): 2

Products sorted by name:
- Aluz (Tk 20)
- Bingo (Tk 25)
- Cheetos (Tk 35)
- Haldiram (Tk 25)
- Kurkure (Tk 15)
- Lays (Tk 30)
- Pringles (Tk 60)
- Ruffles (Tk 40)
- Sun (Tk 10)
- Uncle Chips (Tk 20)
Quick Sort took 0.00002 seconds.
Enter exact product name to add to cart (or 'back' to return): Aluz
Aluz added to cart. Total: Tk 20
Binary Search took 0.000002 seconds.
```

```
=== Smart Vending Machine ===
1. Beverages
2. Chips
3. Chocolates
4. Checkout/Exit
Enter choice (1-4): 1

Products sorted by name:
- 7Up (Tk 30)
- Apple Juice (Tk 50)
- Drinko (Tk 10)
- Fanta (Tk 35)
- Mango Juice (Tk 45)
- Max Cola (Tk 20)
- Mozo (Tk 45)
- Pepsi (Tk 25)
- Sprite (Tk 40)
- StarShip (Tk 30)
Quick Sort took 0.00002 seconds.
Enter exact product name to add to cart (or 'back' to return): Mozo
Mozo added to cart. Total: Tk 65
Binary Search took 0.000002 seconds.
```



Conclusion

In this project, I learned how to use algorithms like Quick Sort, Binary Search, and the Greedy Coin Change method in a simple vending machine system. It showed me how sorting helps make searching faster and how to measure the time taken by algorithms using the clock() function. The hardest part to fix was the coin change system, especially when the exact change couldn't be returned. I also faced issues with user input, like when switching from numbers to strings. To improve the project, I would add input checks, make the product search case-insensitive, and improve the change-giving system. I could also add random prices or products to make it more interesting. In the future, this project could be extended with delivery options using Dijkstra's algorithm, combo offers using fractional knapsack, a user-friendly interface, or an admin panel for restocking and editing products. Overall, this project helped me understand how algorithms can be used in real-life problems.