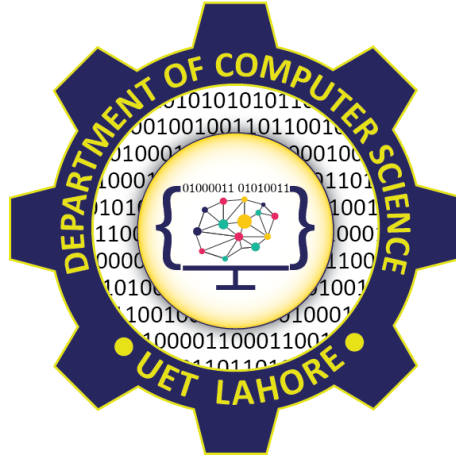


E-Commerce Scrapper



Project Supervisor:

Mr. Nazeef Ul Haq

Submitted By :

Faisal Ilyas
Gul-e-Zahra

2022-CS-63
2022-CS-75

Department of Computer Science
University of Engineering and Technology, Lahore
Pakistan

Contents

1 Project Outline	2
1.1 Uses of the E-commerce Data Scraper:	2
1.2 Where it can be Used:	2
2 Project Domain	3
2.1 Data Collection:	3
2.2 Price Monitoring and Comparison:	3
2.3 Product Availability and Trends:	3
2.4 Customer Sentiment Analysis:	3
2.5 Market Research and Business Strategies:	3
3 Project Idea	3
4 Project Audience	4
4.1 Businesses and E-commerce Enterprises:	4
4.2 Market Analysts and Researchers:	4
4.3 Government and Regulatory Bodies:	4
4.4 Entrepreneurs and Innovators:	4
4.5 General Public and Consumers:	5
5 Project Motivation	5
6 Project Details	5
6.1 Scraping Sources	5
6.2 Scraping Focus:	6
6.3 Source Links:	6
6.4 Screenshots of Websites:	6
7 Technologies and Tools	9
8 Attributes Scrapped	9
9 GitLab Repository link	9
10 Algorithms	9
11 GUI Designs	31
12 Overview of Working	31
13 Features	34
14 Conclusion	35

1 Project Outline

E-commerce platforms serve as repositories of valuable product data, crucial for market research and business strategies. Extracting and structuring this data efficiently is a daunting task. Our client requires an optimal scraping solution to extract, organize, and utilize this data effectively. Enter the E-commerce Data Scraper, a digital detective designed to gather vital information from online shopping websites. This tool examines web pages, extracting essential details such as product names, prices, descriptions, ratings, delivery information, discount prices, and discount percentages.

This scraped data serves multiple purposes, aiding in price comparison, product availability tracking, and understanding customer sentiments. By providing insights into product names and descriptions, it assists businesses in understanding market trends and consumer preferences. Additionally, the extraction of prices, discounts, and delivery information enables businesses to make informed pricing and inventory decisions. Ratings and customer sentiments offer valuable feedback, helping businesses enhance customer satisfaction and improve their products or services.

1.1 Uses of the E-commerce Data Scraper:

Market Research:

Gathers comprehensive data for market analysis and trend identification. Enables businesses to make informed decisions based on current market conditions.

Competitor Analysis:

Monitors competitor product offerings, pricing strategies, and customer feedback. Facilitates strategic planning by understanding the competitive landscape.

Price Comparison:

Helps consumers compare prices across multiple platforms, ensuring the best deals. Empowers businesses to adjust pricing strategies in real-time, maximizing competitiveness.

Customer Sentiment Analysis:

Aggregates customer ratings and reviews, providing insights into product satisfaction levels. Helps businesses enhance product quality and customer experience based on feedback.

Inventory Management:

Tracks product availability, preventing overstocking or understocking issues. Optimizes inventory levels based on demand, reducing storage costs and increasing efficiency.

1.2 Where it can be Used:

E-commerce Businesses:

Enhances product listings with up-to-date information, attracting more customers. Facilitates data-driven decision-making for inventory management and pricing strategies.

Market Research Firms:

Provides valuable data for in-depth market analysis and trend forecasting. Supports clients with accurate, timely, and actionable insights.

Retailers and Distributors:

Helps in optimizing product pricing to stay competitive in the market. Assists in identifying popular products, aiding in effective stock purchasing.

Consumer Advocacy Groups:

Utilizes customer feedback data to advocate for consumer rights and fair pricing. Identifies trends in customer satisfaction, enabling targeted advocacy efforts.

2 Project Domain

The project domain for our E-commerce Data Scraping project primarily revolves around the vast and dynamic landscape of the e-commerce industry. E-commerce, short for electronic commerce, refers to the buying and selling of goods and services over the internet. In this digital era, e-commerce platforms have become essential for businesses, offering a wide array of products to a global audience. Our project focuses on the data extraction process within this domain, aiming to gather valuable information about products, prices, availability, reviews, and other pertinent data points.

2.1 Data Collection:

E-commerce platforms host an extensive range of products, each with diverse attributes and characteristics. Our project involves the systematic extraction of this data from various online shopping websites. This includes gathering details such as product names, descriptions, images, prices, discount offers, delivery information, customer ratings, and reviews. The scope of data collection spans across multiple product categories, allowing for a comprehensive understanding of the market landscape.

2.2 Price Monitoring and Comparison:

One of the core aspects of our project domain is price monitoring. E-commerce businesses frequently adjust their product prices based on market demand, competitor pricing, and promotional events. Our data scraping solution enables businesses and consumers alike to monitor these price fluctuations in real-time. By comparing prices across different platforms, users can make informed purchasing decisions, ensuring they get the best value for their money.

2.3 Product Availability and Trends:

Understanding product availability is crucial for both consumers and retailers. For consumers, it ensures that the desired products are in stock before making a purchase. For retailers, analyzing product availability helps in managing inventory efficiently. Additionally, our project delves into identifying trends within the e-commerce industry, such as popular products, emerging categories, and customer preferences. This data aids businesses in adapting their strategies to meet market demands effectively.

2.4 Customer Sentiment Analysis:

Customer ratings and reviews provide valuable insights into product satisfaction levels and overall customer experience. Our project incorporates sentiment analysis techniques to assess these reviews. By analyzing customer sentiments, businesses can identify strengths and weaknesses in their products and services. Positive feedback can be leveraged for marketing purposes, while negative feedback offers areas for improvement, contributing to enhanced customer satisfaction.

2.5 Market Research and Business Strategies:

E-commerce data scraping plays a pivotal role in market research and shaping business strategies. By aggregating and analyzing data from multiple sources, businesses can gain a competitive edge. Market trends, customer preferences, and competitor analyses derived from the scraped data empower businesses to make informed decisions. This data-driven approach enhances marketing campaigns, optimizes pricing strategies, and guides product development efforts.

3 Project Idea

In today's digital age, e-commerce platforms have become essential hubs for consumers, offering an array of products at varying prices and discounts. To comprehend market trends, pricing strategies, and consumer behavior, businesses require access to vast and diverse e-commerce data. The idea for this project stems from the need for a comprehensive

solution to gather detailed product information from a multitude of e-commerce websites. The goal of this project is to develop a robust E-commerce Data Scraper capable of extracting extensive product data from one million e-commerce webpages. This scraper will collect crucial details such as product name, price, description, rating, number of reviews, discount price, and discount percentage. This vast dataset will empower businesses, market analysts, and researchers with invaluable insights into pricing dynamics, customer preferences, and market trends.

4 Project Audience

The potential audience for the E-commerce Data Aggregation project encompasses a wide range of stakeholders, each with distinct needs and objectives. Identifying the project's audience is crucial as it ensures that the gathered insights are disseminated effectively, benefiting various sectors of the market and research community. Here are the primary audiences for this project:

4.1 Businesses and E-commerce Enterprises:

Small, Medium, and Large Businesses:

Gain access to valuable market insights for optimizing pricing strategies, understanding consumer behavior, and enhancing product offerings.

Startups:

Utilize the collected data to identify market gaps, evaluate product demand, and make informed decisions about their initial product offerings.

E-commerce Platforms:

Enhance their understanding of seller performance, popular products, and customer satisfaction metrics to improve platform features and seller support services.

Retailers and Merchants:

Utilize competitor analysis and pricing trends to adjust their offerings, making them more competitive and appealing to customers.

4.2 Market Analysts and Researchers:

Market Research Firms:

Access a vast dataset for in-depth market analysis, enabling the formulation of comprehensive market reports and trend analyses.

Academic Researchers:

Utilize the collected data for academic studies, enabling research on consumer behavior, pricing strategies, and market trends.

Data Scientists:

Analyze the dataset to develop predictive models, machine learning algorithms, and statistical analyses, leading to innovative market insights.

4.3 Government and Regulatory Bodies:

Consumer Protection Agencies:

Utilize the data to monitor pricing practices, ensuring fair pricing for consumers and preventing price manipulation.

Economic Analysts:

Leverage the insights to gauge economic trends, consumer spending patterns, and the overall health of the e-commerce sector.

4.4 Entrepreneurs and Innovators:

Product Developers:

Understand market demands and emerging trends to develop innovative products tailored to consumer needs and preferences.

Entrepreneurs:

Use market insights to identify potential niches, enabling the creation of unique business ventures catering to specific consumer segments.

4.5 General Public and Consumers:

Online Shoppers:

Benefit indirectly from the project by experiencing more competitive pricing, improved product quality, and enhanced customer service as businesses adapt to market insights.

Consumer Advocacy Groups:

Utilize the data to advocate for fair pricing practices, consumer rights, and ethical business conduct within the e-commerce industry.

5 Project Motivation

The idea behind creating the E-commerce Data Aggregation project is quite simple: in today's online shopping world, having the right information is key. Think about when you're trying to buy something online – you want to know not just the price, but also what others think about it, whether there are any discounts, and how it compares to similar products. For businesses, especially small ones and new startups, understanding these market trends is crucial. This project was inspired by the idea of helping these businesses. By collecting detailed information from various online stores – such as product names, prices, ratings, reviews, and discounts – this project aims to give businesses a clear picture of what's happening in the market. It's like providing a magnifying glass to see the details of the e-commerce world.

But it's not just about businesses; it's about you and me, the consumers too. When we shop online, having all this information at our fingertips helps us make better choices. We can pick products that fit our needs and budgets, and we know we're getting a good deal. This project is motivated by the idea of empowering shoppers with this knowledge. Additionally, researchers and analysts can use this wealth of data to understand shopping trends, helping companies create products that people want.

A big part of this project's motivation is doing all of this ethically and responsibly. We want to make sure that while we're collecting all this data, we're respecting privacy rules and not causing any disruptions to the websites we're getting information from. In essence, this project is like a bridge, connecting businesses, consumers, researchers, and the online market in a way that's helpful, ethical, and empowering for everyone involved.

6 Project Details

6.1 Scraping Sources

1. Flipkart:

Flipkart is India's largest online marketplace, offering a wide array of products ranging from electronics and fashion to home appliances and groceries. It is known for its extensive product catalog, user-friendly interface, and competitive pricing. Flipkart hosts numerous sellers and brands, making it a go-to platform for millions of Indian consumers.

2. Alibaba:

Alibaba is a global e-commerce powerhouse headquartered in China. It operates as a B2B (business-to-business) marketplace, connecting international buyers with suppliers and wholesalers. With an extensive range of products, Alibaba is a hub for businesses looking to source goods in bulk from various industries.

3. AliExpress:

AliExpress, a subsidiary of Alibaba Group, is a popular online retail platform offering a vast selection of products at competitive prices. It caters primarily to individual consumers and small businesses worldwide. AliExpress is renowned for its diverse product categories, affordable pricing, and international shipping options.

4. Amazon:

Amazon is a multinational technology company and the world's largest online retailer. It provides a diverse range of products, including electronics, books, fashion, and household items. Amazon is known for its quick delivery services, extensive product reviews, and innovative offerings such as Amazon Prime and Amazon Echo devices.

5. Daraz:

Daraz is a leading e-commerce platform in South Asia, serving countries like Pakistan, Bangladesh, Sri Lanka, Nepal, and Myanmar. It offers a wide variety of products, ranging from fashion and beauty to electronics and home essentials. Daraz is recognized for its localized services and tailored offerings to meet the specific needs of South Asian consumers.

6. eBay:

eBay is a global online marketplace where individuals and businesses can buy and sell new or used products. It operates through auction-style listings and fixed-price formats, allowing users to bid on items or purchase them directly. eBay is known for its diverse product categories, unique and rare finds, and a platform that encourages both buying and selling.

6.2 Scraping Focus:

The scraping process for these e-commerce platforms will focus on collecting essential product details to provide comprehensive insights. This includes information such as product names, prices, descriptions, seller details, customer reviews, shipping options, and any discounts or special offers available.

6.3 Source Links:

<http://www.flipkart.com>
<http://www.alibaba.com>
<http://www.aliexpress.com>
<http://www.daraz.com>
<http://www.amazon.com>
<http://www.ebay.com>

6.4 Screenshots of Websites:

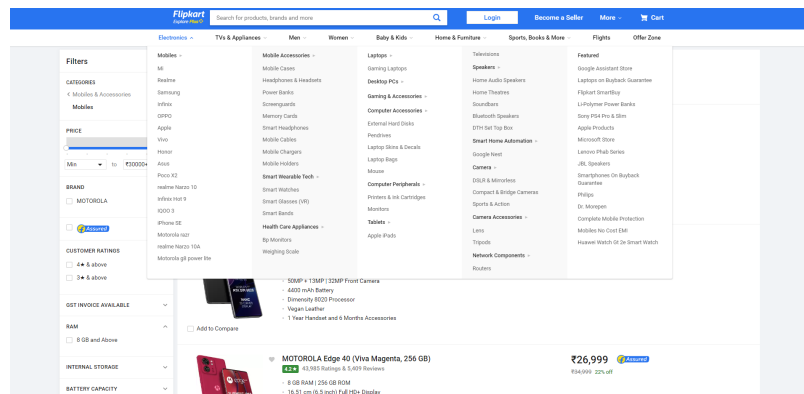


Figure 1: Flipkart

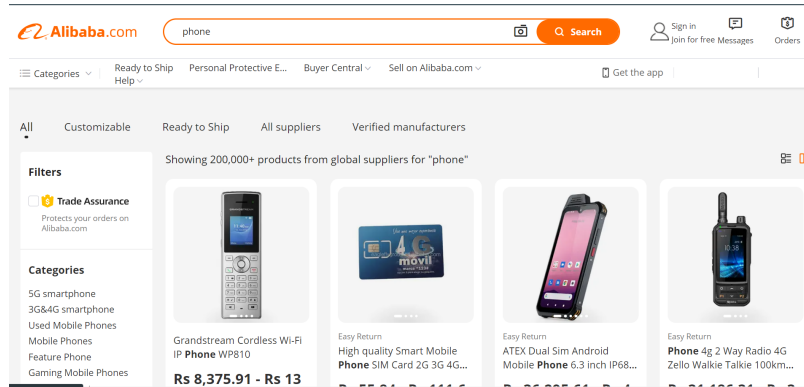


Figure 2: AliBaba

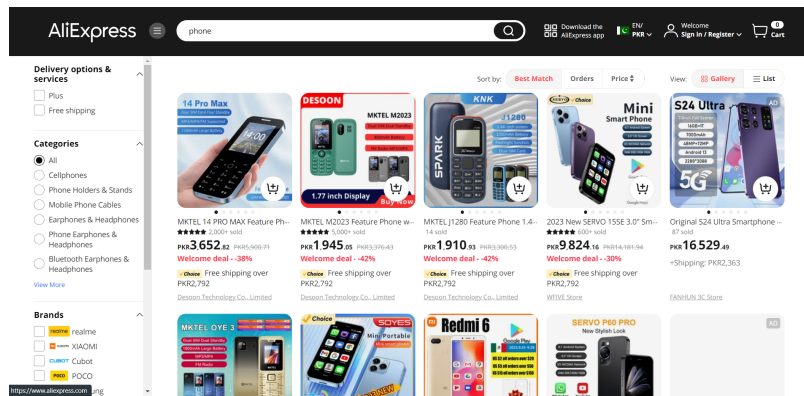


Figure 3: AliExpress

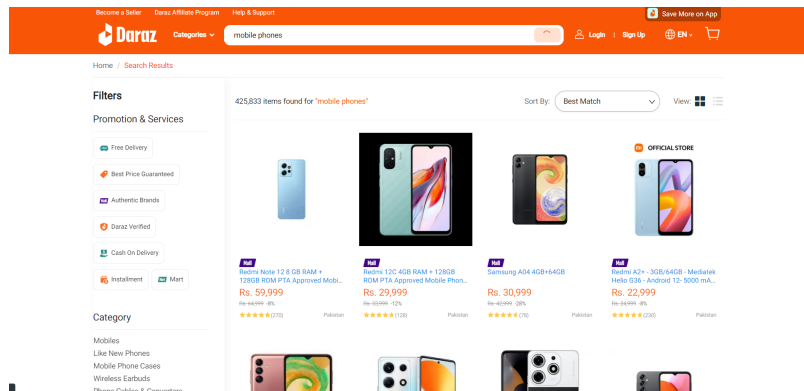


Figure 4: Daraz

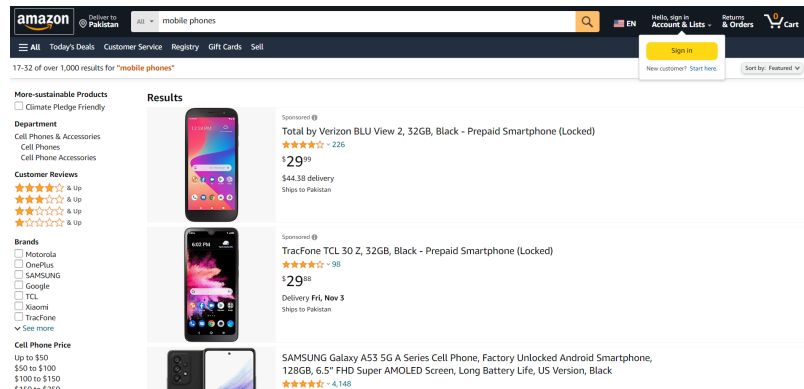


Figure 5: Amazon

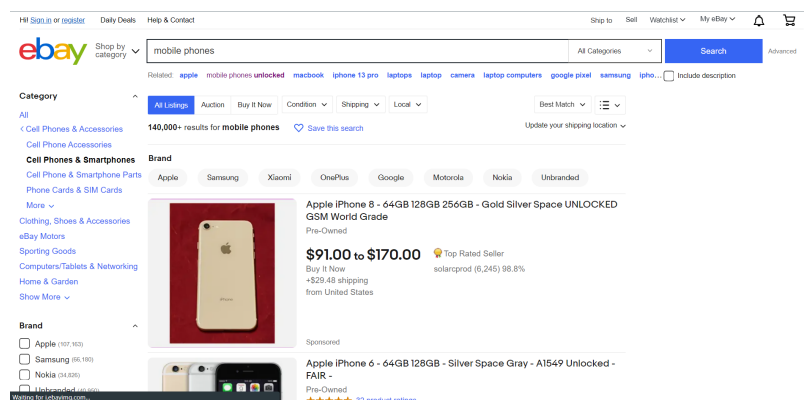


Figure 6: Ebay

7 Technologies and Tools

Programming Languages:

Python (utilized for web scraping, data analysis, and loading data into CSV files).

Web Scraping Libraries:

Beautiful Soup, Selenium (for efficient web scraping tasks).

Database:

CSV files utilized with Python for loading and managing data.

Data Analysis:

Pandas (for data manipulation and analysis), Matplotlib (for data visualization).

Web Framework :

QTDesigner (for developing a user-friendly web interface, if required).

8 Attributes Scrapped

Attribute	Definition
Name	Contains the product name
Price	Indicates the price of the product
Description	Category of the product
Rating	Rating given by users
No. of Reviews	Reviews given by the users
Discounted Price	Price after applying discount
Delivery	Information about product delivery
Discount Percentage	Percentage of discount applied (in %)

Table 1: Attributes and Their Definitions

9 GitLab Repository link

Link: <https://gitlab.com/faisal-ilyas/mid-project>

10 Algorithms

Selection Sort

Description of Algorithm:

Selection Sort is a simple comparison-based sorting algorithm. It works by dividing the input array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all the elements. The algorithm repeatedly selects the smallest (or largest, depending on the order) element from the unsorted region and swaps it with the first element of the unsorted region. The sorted region grows, and the unsorted region shrinks until all elements are sorted.

Pseudo code:

```
for i from 0 to length(A) - 1
    min_index = i
    for j from i + 1 to length(A) - 1
        if A[j] < A[min_index]
            min_index = j
    swap A[i] and A[min_index]
```

Time Complexity Analysis:

Best Case: $O(n^2)$ - When elements are in random order.

Average Case: $O(n^2)$ - When elements are in random order.

Worst Case: $O(n^2)$ - When elements are in reverse order or nearly sorted.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no extra space required.

Average Case: $O(1)$ - It sorts the array in-place.

Worst Case: $O(1)$ - No extra space used other than the input array.

Strengths:

1. **Simplicity:** Selection Sort is easy to understand and implement.
2. **In-Place Sorting:** It doesn't require additional memory space for sorting.
3. **Stable Algorithm:** Selection Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements in the sorted output.

Weaknesses:

1. **Inefficiency for Large Data Sets:** Selection Sort is inefficient for large datasets due to its quadratic time complexity.
2. **Lack of Adaptivity:** Its performance does not change based on the initial order of elements; it always takes the same time to sort a given array.
3. **Not Adaptive:** Selection Sort does not adapt to the input data, meaning it performs the same steps regardless of the input.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6].

1. **i = 0:** Minimum element is 1 (at index 3). Swap with element at index 0.
2, 9, 5, 5, 6
.
2. **i = 1:** Minimum element is 2 (at index 1). No need to swap.
2, 9, 5, 5, 6
.
3. **i = 2:** Minimum element is 5 (at index 3). Swap with element at index 2.
2, 5, 9, 5, 6
.
4. **i = 3:** Minimum element is 5 (at index 4). Swap with element at index 3.
2, 5, 5, 9, 6
.
5. **i = 4:** Minimum element is 6 (at index 5). Swap with element at index 4.
2, 5, 5, 6, 9
.

Python Code:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

Bubble Sort

Description of Algorithm:

Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order. This process is repeated for each pair of adjacent elements in the array, moving from the beginning to the end. The algorithm gets its name because smaller elements "bubble" to the top of the array during each pass.

Pseudo code:

```
for i from 0 to length(A) - 1
    for j from 0 to length(A) - 1 - i
        if A[j] > A[j + 1]
            swap A[j] and A[j + 1]
```

Time Complexity Analysis:

Best Case: $O(n)$ - When the array is already sorted.

Average Case: $O(n^2)$ - When elements are in random order.

Worst Case: $O(n^2)$ - When elements are in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no extra space required.

Average Case: $O(1)$ - It sorts the array in-place.

Worst Case: $O(1)$ - No extra space used other than the input array.

Strengths:

1. **Simplicity:** Bubble Sort is easy to understand and implement.
2. **In-Place Sorting:** It doesn't require additional memory space for sorting.
3. **Stable Algorithm:** Bubble Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements in the sorted output.

Weaknesses:

1. **Inefficiency for Large Data Sets:** Bubble Sort is inefficient for large datasets due to its quadratic time complexity.
2. **Lack of Adaptivity:** Its performance does not change based on the initial order of elements; it always takes the same time to sort a given array.
3. **Not Adaptive:** Bubble Sort does not adapt to the input data, meaning it performs the same steps regardless of the input.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6].

1. **Pass 1:** [2, 5, 1, 5, 6, 9] (Swapped 5 and 2) [2, 1, 5, 5, 6, 9] (Swapped 5 and 1) [2, 1, 5, 5, 6, 9] (No swap needed) [2, 1, 5, 5, 6, 9] (No swap needed) [1, 2, 5, 5, 6, 9] (Swapped 2 and 1)
2. **Pass 2:** [1, 2, 5, 5, 6, 9] (No swap needed) [1, 2, 5, 5, 6, 9] (No swap needed) [1, 2, 5, 5, 6, 9] (No swap needed) [1, 2, 5, 5, 6, 9] (No swap needed)
(Array is already sorted, no more passes needed)

Python Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Insertion Sort

Description of Algorithm:

Insertion Sort is a simple comparison-based sorting algorithm. It builds the sorted array one element at a time by repeatedly selecting an element from the unsorted part of the array and inserting it into its correct position within the sorted part of the array.

Pseudo code:

```
for i from 1 to length(A) - 1
    key = A[i]
    j = i - 1
    while j >= 0 and A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = key
```

Time Complexity Analysis:

Best Case: $O(n)$ - When the array is already sorted.

Average Case: $O(n^2)$ - When elements are in random order.

Worst Case: $O(n^2)$ - When the array is sorted in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no extra space required.

Average Case: $O(1)$ - It sorts the array in-place.

Worst Case: $O(1)$ - No extra space used other than the input array.

Strengths:

1. **Simplicity:** Insertion Sort is easy to understand and implement.
2. **Efficiency for Small Data Sets:** It performs well for small datasets or nearly sorted datasets.
3. **In-Place Sorting:** It doesn't require additional memory space for sorting.

Weaknesses:

1. **Inefficiency for Large Data Sets:** Insertion Sort is inefficient for large datasets due to its quadratic time complexity.
2. **Sensitivity to Input:** Performance is highly dependent on the initial order of elements.
3. **Not Adaptive:** Its performance does not change based on the initial order of elements; it always takes the same time to sort a given array.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6].

1. **i = 1:** Key = 2. Move 5 to the right.
Array becomes [5, 5, 9, 1, 5, 6].
5, 9, 1, 5, 6
.
2. **i = 2:** Key = 9. No need to move elements.
5, 9, 1, 5, 6
.
3. **i = 3:** Key = 1. Move 9, 5, and 2 to the right.
Array becomes [2, 5, 5, 9, 6, 6].
2, 5, 9, 5, 6
.
4. **i = 4:** Key = 5. Move 9 to the right.
Array becomes [2, 5, 5, 9, 6, 6].
2, 5, 5, 6, 9
.
5. **i = 5:** Key = 6. Move 9 to the right.
Array becomes [1, 2, 5, 5, 6, 9].

Python Code:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

Merge Sort

Description of Algorithm:

Merge Sort is a divide-and-conquer sorting algorithm that works by recursively dividing the input array into smaller subarrays until each subarray contains only one element. Then, it merges these subarrays back together, sorting them in the process. Merge Sort is known for its stability and predictable performance.

Pseudo code:

```
merge_sort(arr):
    if length(arr) <= 1:
        return arr
    mid = length(arr) // 2
```

```

    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

merge(left, right):
    result = []
    while left and right:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    if left:
        result.extend(left)
    if right:
        result.extend(right)
    return result

```

Time Complexity Analysis:

Best Case: $O(n \log n)$ - When elements are in random order.

Average Case: $O(n \log n)$ - When elements are in random order.

Worst Case: $O(n \log n)$ - When elements are in reverse order or nearly sorted.

Space Complexity Analysis:

Best Case: $O(n)$ - Additional space required for the merged array.

Average Case: $O(n)$ - Additional space required for the merged array.

Worst Case: $O(n)$ - Additional space required for the merged array.

Strengths:

1. **Efficiency:** Merge Sort guarantees $O(n \log n)$ time complexity in all cases, making it efficient for large datasets.
2. **Stability:** Merge Sort is stable, meaning it preserves the relative order of equal elements in the sorted output.
3. **Predictable Performance:** Its time complexity is consistent, regardless of the input data.

Weaknesses:

1. **Space Complexity:** Merge Sort uses additional memory for the merged array, making it less memory-efficient compared to some in-place sorting algorithms.
2. **Slower for Small Datasets:** The recursive nature of Merge Sort can lead to overhead for small datasets.
3. **Not Adaptive:** Merge Sort does not adapt to the input data, meaning it performs the same steps regardless of the input.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6].

1. **Split:** [5, 2, 9] [1, 5, 6]
2. **Split:** [5] [2, 9] [1] [5, 6]
3. **Merge:** [2, 5, 9] [1, 5, 6]
4. **Merge:** [1, 2, 5, 5, 6, 9]

Python Code:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    while left and right:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    if left:
        result.extend(left)
    if right:
        result.extend(right)
    return result
```

Quick Sort

Description of Algorithm:

Quick Sort is a highly efficient, comparison-based sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This process of dividing and sorting continues until the base case of single-element sub-arrays is reached, making the entire array sorted.

Pseudo code:

```
quick_sort(arr, low, high):
    if low < high:
        pivot_index = partition(arr, low, high)
        quick_sort(arr, low, pivot_index)
        quick_sort(arr, pivot_index + 1, high)

partition(arr, low, high):
    pivot = arr[low]
    left = low + 1
    right = high
    done = False
    while not done:
        while left <= right and arr[left] <= pivot:
            left = left + 1
        while arr[right] >= pivot and right >= left:
            right = right - 1
        if right < left:
            done = True
        else:
            arr[left], arr[right] = arr[right], arr[left]
    arr[low], arr[right] = arr[right], arr[low]
    return right
```


Time Complexity Analysis:

Best Case: $O(n \log n)$ - Occurs when the pivot divides the array into approximately equal halves.

Average Case: $O(n \log n)$ - The average time complexity is also $O(n \log n)$ due to the random partitioning of elements.

Worst Case: $O(n^2)$ - Occurs when the pivot always picks the smallest or largest element, leading to unbalanced partitions.

Space Complexity Analysis:

Best Case: $O(\log n)$ - In the best case, Quick Sort uses $\log n$ additional space for recursive function calls.

Average Case: $O(\log n)$ - On average, Quick Sort uses $\log n$ additional space.

Worst Case: $O(n)$ - In the worst case, Quick Sort can use $O(n)$ additional space for recursive function calls due to unbalanced partitions.

Strengths:

1. **Efficiency:** Quick Sort is highly efficient and often faster than other sorting algorithms like Merge Sort and Bubble Sort.
2. **In-Place Sorting:** Quick Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space for sorting.
3. **Adaptability:** Quick Sort is adaptive, meaning it performs well for partially sorted arrays.

Weaknesses:

1. **Worst Case Time Complexity:** Quick Sort has a worst-case time complexity of $O(n^2)$ when the array is already sorted or nearly sorted, leading to unbalanced partitions.
2. **Not Stable:** Quick Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.
3. **Random Pivot Choice:** The choice of pivot can significantly affect the performance, and choosing a bad pivot can lead to poor performance.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6].

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Pivot Selection:** Choosing 5 as the pivot.
3. **Partitioning Step:** [2, 1, 5, 5, 6, 9]
4. **Recursive Calls:** Apply Quick Sort to [2, 1] and [5, 6, 9].
5. **Sorted Array:** [1, 2, 5, 5, 6, 9]

Python Code:

```
def quick_sort(arr, low, high):
    if low < high:
        pivot_index = partition(arr, low, high)
        quick_sort(arr, low, pivot_index)
        quick_sort(arr, pivot_index + 1, high)

def partition(arr, low, high):
    pivot = arr[low]
    left = low + 1
```

```

right = high
done = False
while not done:
    while left <= right and arr[left] <= pivot:
        left = left + 1
    while arr[right] >= pivot and right >= left:
        right = right - 1
    if right < left:
        done = True
    else:
        arr[left], arr[right] = arr[right], arr[left]
arr[low], arr[right] = arr[right], arr[low]
return right

```

Counting Sort

Description of Algorithm:

Counting Sort is a non-comparison-based sorting algorithm that works for integers within a specific range. It counts the frequency of each element in the input array and uses this information to place the elements in sorted order. Counting Sort assumes that the input elements are non-negative integers and works efficiently when the range of integers in the input is not significantly larger than the number of elements.

Pseudo code:

```

counting_sort(arr, max_value):
    count_array = [0] * (max_value + 1)
    output = [0] * len(arr)

    for num in arr:
        count_array[num] += 1

    for i in range(1, max_value + 1):
        count_array[i] += count_array[i - 1]

    for num in reversed(arr):
        output[count_array[num] - 1] = num
        count_array[num] -= 1

    return output

```

Time Complexity Analysis:

Best Case: $O(n + k)$ - When the range of input integers (k) is not significantly larger than the number of elements (n).

Average Case: $O(n + k)$ - Same as the best case.

Worst Case: $O(n + k)$ - Same as the best and average cases.

Space Complexity Analysis:

Best Case: $O(n + k)$ - Additional space required for count array and output array.

Average Case: $O(n + k)$ - Same as the best case.

Worst Case: $O(n + k)$ - Same as the best and average cases.

Strengths:

1. **Linear Time Complexity:** Counting Sort achieves linear time complexity for sorting integers within a specific range.
2. **Stability:** Counting Sort is stable, meaning it preserves the relative order of equal elements in the sorted output.
3. **Efficiency for Small Ranges:** It is highly efficient when the range of integers is not significantly larger than the number of elements.

Weaknesses:

1. **Space Complexity:** Counting Sort uses additional memory for the count array and the output array, making it less memory-efficient for large ranges.
2. **Limited Applicability:** It works well for integers, but its applicability is limited to situations where the range of integers is known and not too large.
3. **Not In-Place:** Counting Sort is not an in-place sorting algorithm; it requires additional space for the count array and the output array.

Dry Run Example:

Let's say we want to sort the array [4, 2, 4, 1, 0, 3] using Counting Sort with a maximum value of 4.

1. **Count Array:** [1, 1, 1, 1, 2] (Frequency of elements from 0 to 4)
2. **Modified Count Array:** [1, 2, 3, 4, 6] (Cumulative frequency)
3. **Output Array:** [0, 1, 2, 3, 4, 4] (Sorted array)

Python Code:

```
def counting_sort(arr, max_value):
    count_array = [0] * (max_value + 1)
    output = [0] * len(arr)

    for num in arr:
        count_array[num] += 1

    for i in range(1, max_value + 1):
        count_array[i] += count_array[i - 1]

    for num in reversed(arr):
        output[count_array[num] - 1] = num
        count_array[num] -= 1

    return output
```

Radix Sort

Description of Algorithm:

Radix Sort is a non-comparison-based sorting algorithm that processes the digits of the numbers to sort. It sorts the elements by processing individual digits from the least significant digit (rightmost) to the most significant digit (leftmost) or vice versa. Radix Sort is often used with integers but can be extended to other data types as well.

Pseudo code:

```
radix_sort(arr, base):
    max_num = max(arr)
    exponent = 1
    while max_num / exponent > 0:
        counting_sort(arr, base, exponent)
        exponent *= base

counting_sort(arr, base, exponent):
    count_array = [0] * base
    output = [0] * len(arr)

    for num in arr:
        digit = (num // exponent) % base
        count_array[digit] += 1

    for i in range(1, base):
        count_array[i] += count_array[i - 1]

    for num in reversed(arr):
        digit = (num // exponent) % base
        output[count_array[digit] - 1] = num
        count_array[digit] -= 1

    arr[:] = output[:]
```

Time Complexity Analysis:

Best Case: $O(nk)$ - When the numbers have a fixed number of digits (k) and the base (b) is a constant.

Average Case: $O(nk)$ - Same as the best case.

Worst Case: $O(nk)$ - Same as the best and average cases.

Space Complexity Analysis:

Best Case: $O(n + k)$ - Additional space required for count array and output array.

Average Case: $O(n + k)$ - Same as the best case.

Worst Case: $O(n + k)$ - Same as the best and average cases.

Strengths:

1. **Linear Time Complexity:** Radix Sort achieves linear time complexity for sorting integers with a fixed number of digits.
2. **Stability:** Radix Sort is stable, meaning it preserves the relative order of equal elements in the sorted output.
3. **Useful for Fixed Size Keys:** It works well for sorting integers with a fixed number of digits, common in applications like phone numbers and social security numbers.

Weaknesses:

1. **Not Suitable for Variable Size Keys:** Radix Sort is not suitable for sorting integers with varying numbers of digits.
2. **Limited Applicability:** It works well for integers but is not directly applicable to other data types without additional processing.
3. **Not In-Place:** Radix Sort is not an in-place sorting algorithm; it requires additional space for count array and output array.

Dry Run Example:

Let's say we want to sort the array [170, 45, 75, 90, 802, 24, 2, 66] using Radix Sort with base 10.

1. **First Pass (Sorting by Units):** [170, 90, 802, 2], [24], [75], [45], [66] Sorted: [170, 90, 802, 2, 24, 75, 45, 66]
2. **Second Pass (Sorting by Tens):** [802, 2, 24, 45, 66], [170, 75], [90] Sorted: [802, 2, 24, 45, 66, 170, 75, 90]
3. **Third Pass (Sorting by Hundreds):** [2, 24, 45, 66, 75, 90], [170], [802] Sorted: [2, 24, 45, 66, 75, 90, 170, 802]

Python Code:

```
def counting_sort(arr, base, exponent):
    count_array = [0] * base
    output = [0] * len(arr)

    for num in arr:
        digit = (num // exponent) % base
        count_array[digit] += 1

    for i in range(1, base):
        count_array[i] += count_array[i - 1]

    for num in reversed(arr):
        digit = (num // exponent) % base
        output[count_array[digit] - 1] = num
        count_array[digit] -= 1

    arr[:] = output[:]

def radix_sort(arr, base=10):
    max_num = max(arr)
    exponent = 1
    while max_num / exponent > 0:
        counting_sort(arr, base, exponent)
        exponent *= base
```

Bucket Sort

Description of Algorithm:

Bucket Sort is a sorting algorithm that distributes the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sort.

Pseudo code:

```
bucket_sort(arr, num_buckets):
    max_value = max(arr)
    min_value = min(arr)
    bucket_range = (max_value - min_value) / num_buckets + 1

    buckets = [[] for _ in range(num_buckets)]
    for num in arr:
        index = int((num - min_value) / bucket_range)
        buckets[index].append(num)

    sorted_array = []
```

```

for bucket in buckets:
    sorted_array.extend(sorted(bucket))

return sorted_array

```

Time Complexity Analysis:

Best Case: $O(n + k)$ - Occurs when the elements are uniformly distributed into buckets.

Average Case: $O(n + k)$ - Expected time complexity, where k is the number of buckets.

Worst Case: $O(n^2)$ - Occurs when all elements are placed in a single bucket.

Space Complexity Analysis:

Best Case: $O(n + k)$ - In the best case, where each element is in a separate bucket.

Average Case: $O(n + k)$ - Average space complexity, where k is the number of buckets.

Worst Case: $O(n^2)$ - In the worst case, when all elements are placed in a single bucket.

Strengths:

1. **Linear Time Complexity:** Bucket Sort can achieve linear time complexity for uniformly distributed data.
2. **Adaptability:** Can be adapted for external sorting where data doesn't fit into memory.

Weaknesses:

1. **Not Stable:** Bucket Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.
2. **Performance:** Performance can degrade if the data is not uniformly distributed among the buckets.

Dry Run Example:

Let's say we want to sort the array $[0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]$ using Bucket Sort with 5 buckets.

1. **Initial Array:** $[0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]$
2. **Step 1:** Distribute elements into buckets:
 - Bucket 1: $[0.32, 0.33]$
 - Bucket 2: $[0.37]$
 - Bucket 3: $[]$
 - Bucket 4: $[0.42, 0.47]$
 - Bucket 5: $[0.52, 0.51]$
3. **Step 2:** Sort individual buckets:
 - Bucket 1: $[0.32, 0.33]$
 - Bucket 2: $[0.37]$
 - Bucket 3: $[]$
 - Bucket 4: $[0.42, 0.47]$ (sorted)
 - Bucket 5: $[0.51, 0.52]$ (sorted)
4. **Step 3:** Concatenate sorted buckets: $[0.32, 0.33, 0.37, 0.42, 0.47, 0.51, 0.52]$

Python Code:

```
def bucket_sort(arr, num_buckets):
    max_value = max(arr)
    min_value = min(arr)
    bucket_range = (max_value - min_value) / num_buckets + 1

    buckets = [[] for _ in range(num_buckets)]
    for num in arr:
        index = int((num - min_value) / bucket_range)
        buckets[index].append(num)

    sorted_array = []
    for bucket in buckets:
        sorted_array.extend(sorted(bucket))

    return sorted_array
```

Pancake Sort

Description of Algorithm:

Pancake Sort is a sorting algorithm that sorts a disordered stack of pancakes in order of size. It involves a series of flips to rearrange the elements. The algorithm selects the largest element in the unsorted portion and flips the stack to move it to the top. Then, it flips the entire stack to move the largest element to its correct position. The process is repeated for the remaining elements.

Pseudo code:

```
pancake_sort(arr):
    for size in range(len(arr), 1, -1):
        max_index = arr.index(max(arr[:size]))
        if max_index != size - 1:
            if max_index != 0:
                flip(arr, max_index + 1)
            flip(arr, size)
```

Time Complexity Analysis:

Best Case: $O(n^2)$ - Occurs when the elements are already sorted.

Average Case: $O(n^2)$ - Expected time complexity for a random permutation of elements.

Worst Case: $O(n^2)$ - Occurs when the elements are sorted in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no additional space used.

Average Case: $O(1)$ - In-place sorting.

Worst Case: $O(1)$ - In-place sorting.

Strengths:

1. **Simplicity:** Pancake Sort is relatively easy to understand and implement.
2. **In-Place Sorting:** It sorts the array in-place without using additional memory.

Weaknesses:

1. **Inefficiency:** Pancake Sort is inefficient for large datasets due to its quadratic time complexity.
2. **Not Stable:** Pancake Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6] using Pancake Sort.

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Step 1:** Flip to move 9 to the top. [9, 2, 5, 1, 5, 6]
3. **Step 2:** Flip the entire stack to move 9 to the bottom. [6, 5, 1, 5, 2, 9]
4. **Step 3:** Flip to move 6 to the top. [6, 5, 1, 5, 2, 9]
5. **Step 4:** Flip the entire stack to move 6 to its correct position. [5, 6, 1, 5, 2, 9]
6. **Step 5:** Flip to move 5 to the top. [5, 6, 1, 5, 2, 9]
7. **Step 6:** Flip the entire stack to move 5 to its correct position. [1, 2, 5, 5, 6, 9]

Python Code:

```
def pancake_sort(arr):
    for size in range(len(arr), 1, -1):
        max_index = arr.index(max(arr[:size]))
        if max_index != size - 1:
            if max_index != 0:
                flip(arr, max_index + 1)
            flip(arr, size)

def flip(arr, k):
    arr[:k] = arr[:k][::-1]
```

Shell Sort

Description of Algorithm:

Shell Sort is an in-place comparison-based sorting algorithm that starts by sorting elements that are far apart from each other and progressively reduces the gap between elements to be compared. It is an extension of insertion sort where the elements are compared with a fixed gap between them.

Pseudo code:

```
shell_sort(arr):
    n = length(arr)
    gap = n // 2
    while gap > 0:
        for i from gap to n - 1:
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap = gap // 2
```


Time Complexity Analysis:

Best Case: $O(n \log n)$ - Occurs when the elements are nearly sorted.

Average Case: Depends on the gap sequence used, commonly $O(n^{\frac{3}{2}})$ to $O(n^2)$.

Worst Case: $O(n^2)$ - Occurs when the elements are sorted in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no additional space used.

Average Case: $O(1)$ - In-place sorting.

Worst Case: $O(1)$ - In-place sorting.

Strengths:

1. **In-Place Sorting:** Shell Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space for sorting.
2. **Adaptability:** Shell Sort is adaptive, meaning its performance is influenced by the initial order of elements.

Weaknesses:

1. **Complexity:** Shell Sort's time complexity depends on the gap sequence chosen, making it more complex to analyze.
2. **Not Stable:** Shell Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6] using Shell Sort with a gap sequence of [3, 1].

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Step 1 (Gap = 3):** Gap-sorting the array with a gap of 3. [1, 2, 5, 5, 6, 9]
3. **Step 2 (Gap = 1):** Gap-sorting the array with a gap of 1. [1, 2, 5, 5, 6, 9]

Python Code:

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
```

Comb Sort

Description of Algorithm:

Comb Sort is an in-place, comparison-based sorting algorithm. It is an improvement over Bubble Sort and works by eliminating small turtles (small values near the end of the list) efficiently. Comb Sort repeatedly compares and swaps adjacent elements with a fixed shrink factor until the list is sorted.

Pseudo code:

```
comb_sort(arr):
    gap = len(arr)
    shrink = 1.3
    sorted = False
    while not sorted:
        gap = int(gap / shrink)
        if gap <= 1:
            gap = 1
            sorted = True
        for i in range(len(arr) - gap):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
            sorted = False
```

Time Complexity Analysis:

Best Case: $O(n \log n)$ - Occurs when the elements are nearly sorted.

Average Case: $O(n^2)$ - Expected time complexity for a random permutation of elements.

Worst Case: $O(n^2)$ - Occurs when the elements are sorted in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no additional space used.

Average Case: $O(1)$ - In-place sorting.

Worst Case: $O(1)$ - In-place sorting.

Strengths:

1. **In-Place Sorting:** Comb Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space for sorting.
2. **Simple Implementation:** Comb Sort is relatively easy to implement compared to some other sorting algorithms.

Weaknesses:

1. **Efficiency:** Comb Sort's average and worst-case time complexities are higher than some other sorting algorithms like Quick Sort and Merge Sort.
2. **Not Stable:** Comb Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6] using Comb Sort.

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Step 1 (Gap = 6):** Gap-sorting the array with a gap of 6. [5, 2, 5, 1, 6, 9]
3. **Step 2 (Gap = 4):** Gap-sorting the array with a gap of 4. [5, 2, 1, 5, 6, 9]
4. **Step 3 (Gap = 3):** Gap-sorting the array with a gap of 3. [2, 1, 5, 5, 6, 9]
5. **Step 4 (Gap = 2):** Gap-sorting the array with a gap of 2. [1, 2, 5, 5, 6, 9]
6. **Step 5 (Gap = 1):** Gap-sorting the array with a gap of 1. [1, 2, 5, 5, 6, 9]

Python Code:

```
def comb_sort(arr):
    gap = len(arr)
    shrink = 1.3
    sorted = False
    while not sorted:
        gap = int(gap / shrink)
        if gap <= 1:
            gap = 1
            sorted = True
        swapped = False
        for i in range(len(arr) - gap):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                swapped = True
        if not swapped and gap == 1:
            sorted = True
```

Cocktail Sort

Description of Algorithm:

Cocktail Sort, also known as Bidirectional Bubble Sort, is a variation of the Bubble Sort algorithm. It sorts a list by comparing and swapping adjacent elements, first from left to right and then from right to left, until the list is sorted.

Pseudo code:

```
cocktail_sort(arr):
    n = length(arr)
    swapped = True
    start = 0
    end = n - 1
    while swapped:
        swapped = False
        # Traverse from left to right
        for i from start to end:
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end = end - 1
        # Traverse from right to left
        for i from end - 1 to start - 1:
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start = start + 1
```

Time Complexity Analysis:

Best Case: $O(n)$ - Occurs when the input array is already sorted.

Average Case: $O(n^2)$ - Expected time complexity.

Worst Case: $O(n^2)$ - Occurs when the input array is in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no additional space used.

Average Case: $O(1)$ - In-place sorting.

Worst Case: $O(1)$ - In-place sorting.

Strengths:

1. **In-Place Sorting:** Cocktail Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space for sorting.
2. **Simple Implementation:** The algorithm has a simple implementation.

Weaknesses:

1. **Inefficiency:** Cocktail Sort's average and worst-case time complexities are higher than some other sorting algorithms like Quick Sort and Merge Sort.
2. **Not Stable:** Cocktail Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6] using Cocktail Sort.

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Step 1 (Left to Right):** [2, 5, 1, 5, 6, 9] (swaps: 5 and 2, 9 and 1)
3. **Step 2 (Right to Left):** [2, 1, 5, 5, 6, 9] (swaps: 5 and 1)
4. **Step 3 (Left to Right):** [1, 2, 5, 5, 6, 9] (no swaps)

Python Code:

```
def cocktail_sort(arr):
    n = len(arr)
    swapped = True
    start = 0
    end = n - 1
    while swapped:
        swapped = False
        # Traverse from left to right
        for i in range(start, end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end -= 1
        # Traverse from right to left
        for i in range(end - 1, start - 1, -1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start += 1
```

Heap Sort

Description of Algorithm:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to build a max-heap and then repeatedly extracts the maximum element from the heap and places it at the end of the array. It efficiently sorts large datasets and is an in-place sorting algorithm.

Pseudo code:

```
heapify(arr, n, i):
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    if left_child < n and arr[left_child] > arr[largest]:
        largest = left_child

    if right_child < n and arr[right_child] > arr[largest]:
        largest = right_child

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Time Complexity Analysis:

Best Case: $O(n \log n)$ - Occurs when the input array is already a max-heap.

Average Case: $O(n \log n)$ - Expected time complexity.

Worst Case: $O(n \log n)$ - Occurs when the input array is sorted in reverse order.

Space Complexity Analysis:

Best Case: $O(1)$ - In-place sorting, no additional space used.

Average Case: $O(1)$ - In-place sorting.

Worst Case: $O(1)$ - In-place sorting.

Strengths:

1. **In-Place Sorting:** Heap Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space for sorting.
2. **Time Complexity:** Heap Sort has a consistent $O(n \log n)$ time complexity, making it efficient for large datasets.

Weaknesses:

1. **Not Stable:** Heap Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements.

2. **Cache Performance:** Heap Sort doesn't have good cache performance compared to other sorting algorithms like Merge Sort.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6] using Heap Sort.

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Step 1 (Build Max-Heap):** [9, 5, 6, 1, 2, 5] (max-heapify: 5, 2, 9)
3. **Step 2 (Sort the Heap):** [1, 2, 5, 5, 6, 9] (extracted max elements: 9, 5, 6)

Python Code:

```
def heapify(arr, n, i):
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    if left_child < n and arr[left_child] > arr[largest]:
        largest = left_child

    if right_child < n and arr[right_child] > arr[largest]:
        largest = right_child

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Index Sort

Description of Algorithm:

Index Sort is a non-comparison sorting algorithm that builds an auxiliary array, often referred to as an index or bucket array, to keep track of the original array's elements. By iterating through the original array and mapping its elements to the corresponding indices in the auxiliary array, Index Sort efficiently sorts the elements without directly comparing them.

Pseudo code:

```
index_sort(arr):
    max_val = max(arr)
    min_val = min(arr)
    index_range = max_val - min_val + 1

    # Create and initialize the index array
    index_arr = [0] * index_range
```

```

# Count the occurrences of each element in the original array
for num in arr:
    index_arr[num - min_val] += 1

# Reconstruct the sorted array using the index array
sorted_arr = []
for i in range(index_range):
    sorted_arr.extend([i + min_val] * index_arr[i])

return sorted_arr

```

Time Complexity Analysis:

Best Case: $O(n + k)$ - Occurs when the elements are uniformly distributed into indices.

Average Case: $O(n + k)$ - Expected time complexity, where k is the range of indices.

Worst Case: $O(n + k)$ - Occurs when all elements fall into the same index.

Space Complexity Analysis:

Best Case: $O(k)$ - In the best case, where elements are evenly distributed among indices.

Average Case: $O(k)$ - Average space complexity, where k is the range of indices.

Worst Case: $O(k)$ - In the worst case, when all elements fall into the same index.

Strengths:

1. **Linear Time Complexity:** Index Sort can achieve linear time complexity for uniformly distributed data.
2. **Stable Sorting:** Index Sort is a stable sorting algorithm, preserving the relative order of equal elements.

Weaknesses:

1. **Limited Range:** Index Sort's performance depends on the range of values in the input array, making it less suitable for large ranges.
2. **Space Complexity:** Requires additional space for the index array, which can be a drawback for large datasets.

Dry Run Example:

Let's say we want to sort the array [5, 2, 9, 1, 5, 6] using Index Sort.

1. **Initial Array:** [5, 2, 9, 1, 5, 6]
2. **Step 1 (Create Index Array):** index_arr = [0, 1, 0, 0, 2, 1, 1, 0, 0, 1]
3. **Step 2 (Reconstruct Sorted Array):** [1, 2, 5, 5, 6, 9]

Python Code:

```

def index_sort(arr):
    max_val = max(arr)
    min_val = min(arr)
    index_range = max_val - min_val + 1

    # Create and initialize the index array
    index_arr = [0] * index_range

    # Count the occurrences of each element in the original array
    for num in arr:
        index_arr[num - min_val] += 1

```

```
# Reconstruct the sorted array using the index array
sorted_arr = []
for i in range(index_range):
    sorted_arr.extend([i + min_val] * index_arr[i])

return sorted_arr
```

11 GUI Designs

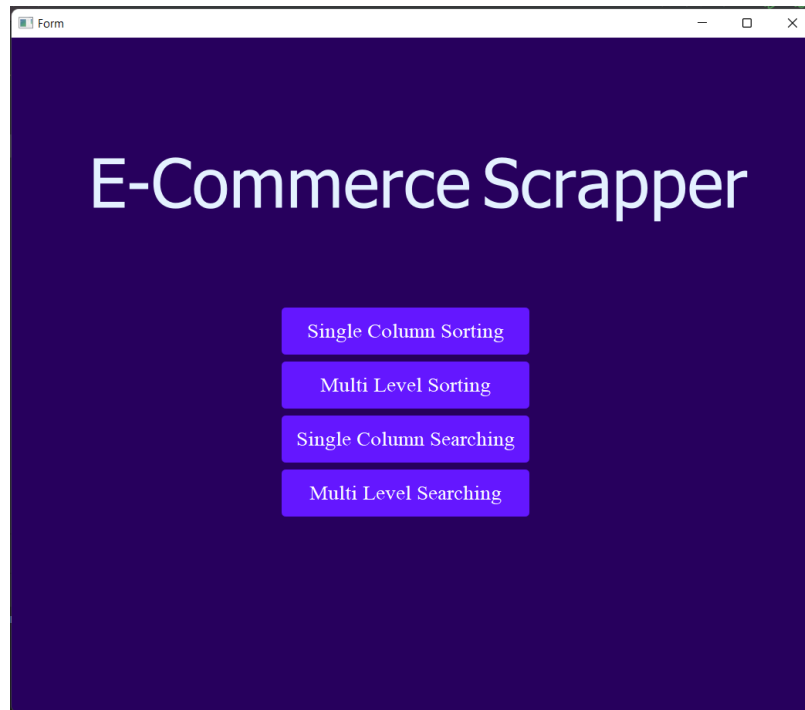


Figure 7: Main Screen

12 Overview of Working

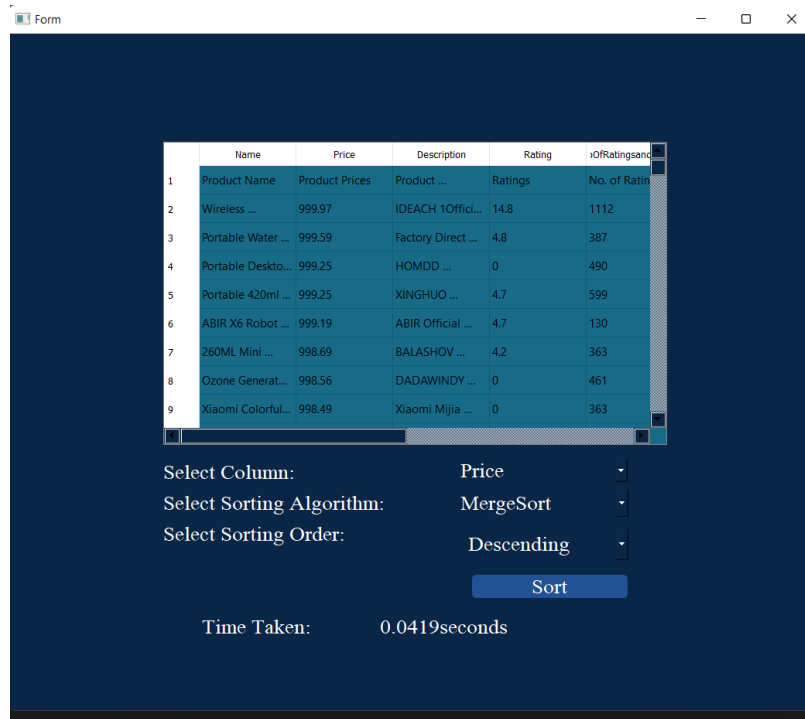
The E-commerce Data Scraping project is designed to efficiently process and manage large volumes of data extracted from e-commerce websites. The application is divided into four main screens, each catering to specific functionalities for sorting and searching the data. Here's how the project works:

Main Screen Interface:

- Upon launching the application, users are presented with the main screen interface.
- Users have the option to load data from a .csv file into the application.
- The loaded data is converted into lists, with each list representing a column of the table.

Single Level Sorting (Screen 1):

- Users can access the first screen for single-level sorting.
- They select a single attribute column based on which the sorting needs to be performed.
- The chosen column is sorted using an appropriate sorting algorithm, and the sorted data is displayed.
- The application records and displays the time taken to perform the sorting operation.



Form

	Name	Price	Description	Rating	No. of Ratings
1	Product Name	Product Prices	Product ...	Ratings	No. of Ratings
2	Wireless ...	999.97	IDEACH 10ffici...	14.8	1112
3	Portable Water ...	999.59	Factory Direct ...	4.8	387
4	Portable Desko...	999.25	HOMDD ...	0	490
5	Portable 420ml ...	999.25	XINGHUO ...	4.7	599
6	ABIR X6 Robot ...	999.19	ABIR Official ...	4.7	130
7	260ML Mini ...	998.69	BALASHOV ...	4.2	363
8	Ozone Generat...	998.56	DADAWINDY ...	0	461
9	Xiaomi Colorful...	998.49	Xiaomi Mijia ...	0	363

Select Column: Price

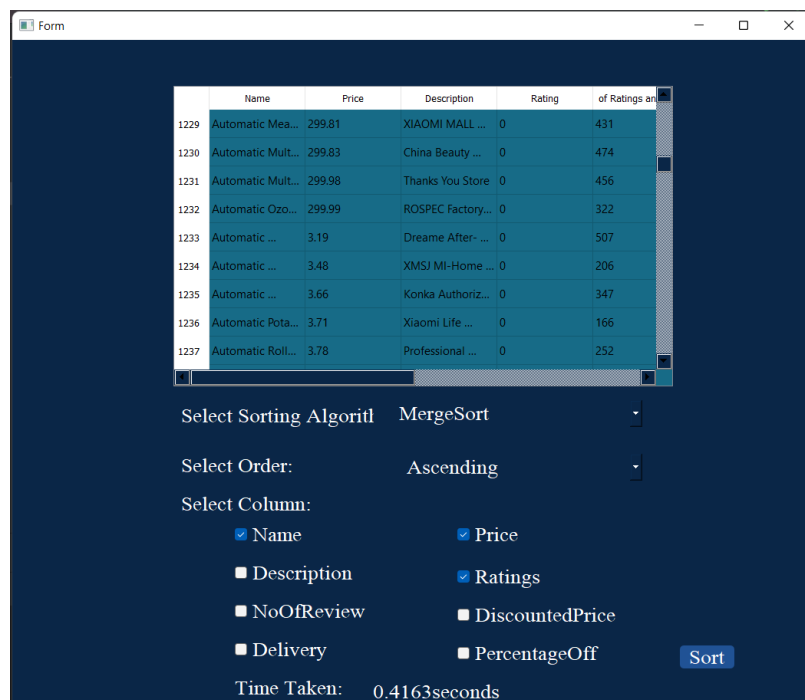
Select Sorting Algorithm: MergeSort

Select Sorting Order: Descending

Sort

Time Taken: 0.0419seconds

Figure 8: Single Level Sorting Screen



Form

	Name	Price	Description	Rating	No. of Ratings
1229	Automatic Mea...	299.81	XIAOMI MALL ...	0	431
1230	Automatic Mult...	299.83	China Beauty ...	0	474
1231	Automatic Mult...	299.98	Thanks You Store	0	456
1232	Automatic Ozo...	299.99	ROSPEC Factory...	0	322
1233	Automatic ...	3.19	Dreame After- ...	0	507
1234	Automatic ...	3.48	XMSJ MI-Home ...	0	206
1235	Automatic ...	3.66	Konka Authoriz...	0	347
1236	Automatic Pota...	3.71	Xiaomi Life ...	0	166
1237	Automatic Roll...	3.78	Professional ...	0	252

Select Sorting Algorithm: MergeSort

Select Order: Ascending

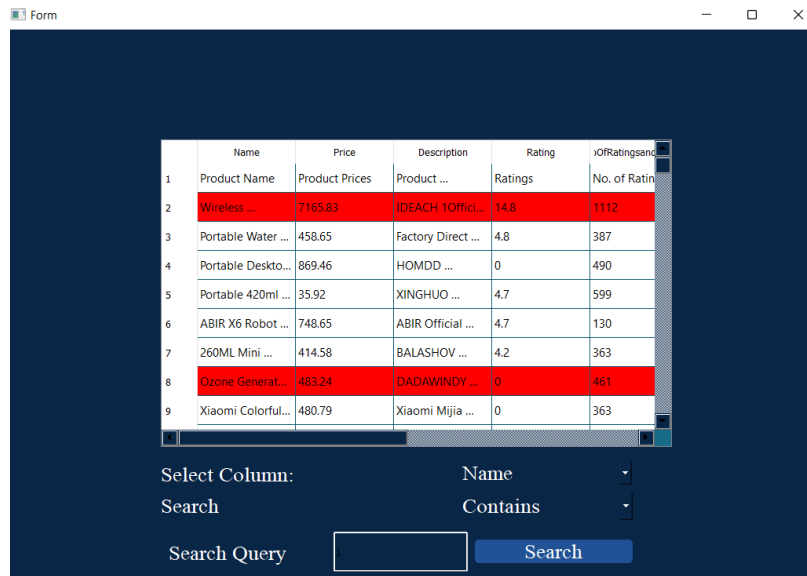
Select Column:

- ☒ Name
- ☒ Price
- ☐ Description
- ☒ Ratings
- ☐ NoOfReview
- ☐ DiscountedPrice
- ☐ Delivery
- ☐ PercentageOff

Sort

Time Taken: 0.4163seconds

Figure 9: Multi Level Sorting Screen

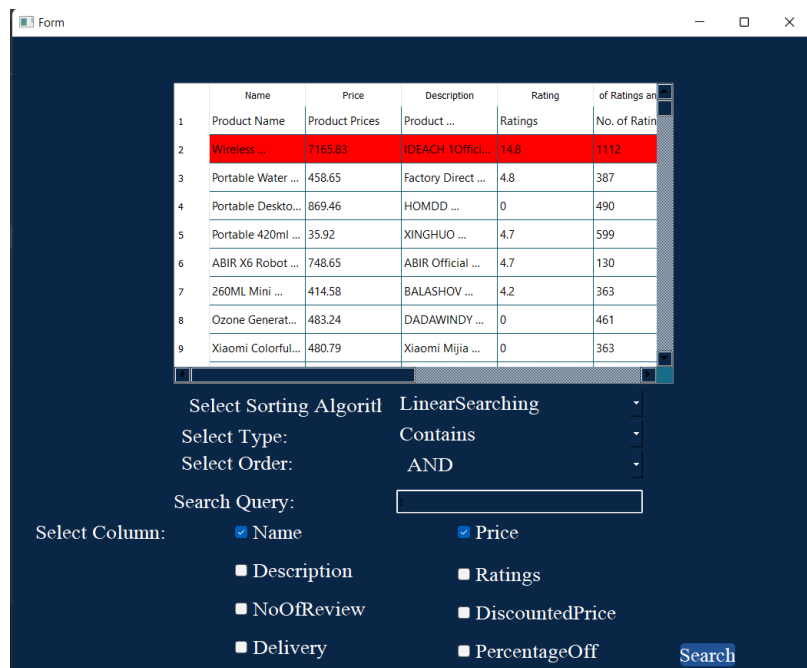


Form

	Name	Price	Description	Rating	No. of Ratings
1	Product Name	Product Prices	Product ...	Ratings	No. of Ratin
2	Wireless ...	7165.83	IDEACH 10ffici...	14.8	1112
3	Portable Water ...	458.65	Factory Direct ...	4.8	387
4	Portable Deskto...	869.46	HOMDD ...	0	490
5	Portable 420ml ...	35.92	XINGHUO ...	4.7	599
6	ABIR X6 Robot ...	748.65	ABIR Official ...	4.7	130
7	260ML Mini ...	414.58	BALASHOV ...	4.2	363
8	Ozone Generat...	483.24	DADAWINDY ...	0	461
9	Xiaomi Colorful...	480.79	Xiaomi Mijia ...	0	363

Select Column: Name
Search Contains
Search Query Search

Figure 10: Single Level Searching Screen



Form

	Name	Price	Description	Rating	No. of Ratings
1	Product Name	Product Prices	Product ...	Ratings	No. of Ratin
2	Wireless ...	7165.83	IDEACH 10ffici...	14.8	1112
3	Portable Water ...	458.65	Factory Direct ...	4.8	387
4	Portable Deskto...	869.46	HOMDD ...	0	490
5	Portable 420ml ...	35.92	XINGHUO ...	4.7	599
6	ABIR X6 Robot ...	748.65	ABIR Official ...	4.7	130
7	260ML Mini ...	414.58	BALASHOV ...	4.2	363
8	Ozone Generat...	483.24	DADAWINDY ...	0	461
9	Xiaomi Colorful...	480.79	Xiaomi Mijia ...	0	363

Select Sorting Algorithm: LinearSearching
Select Type: Contains
Select Order: AND
Search Query:

Select Column: ☒ Name ☒ Price
☐ Description ☐ Ratings
☐ NoOfReview ☐ DiscountedPrice
☐ Delivery ☐ PercentageOff Search

Figure 11: Multi Level Searching

Multi-Level Sorting (Screen 2):

- In the second screen, users can perform multi-level sorting on multiple columns.
- Users select the sorting algorithm they want to apply.
- They then choose multiple attribute columns based on which the sorting needs to be performed.
- The selected columns are sorted sequentially based on the chosen algorithm, and the sorted data is displayed.
- The application records and displays the time taken to execute the multi-level sorting operation.

Single Level Searching (Screen 3):

- The third screen is dedicated to single-level searching using linear search algorithms.
- Users choose an attribute column for searching.
- They have the option to search data that starts with, contains, or ends with a specific query string.
- The application performs the search operation and displays the results.
- The time taken for the search operation is recorded and displayed.

Advanced Searching with Logical Operators (Screen 4):

- The fourth screen allows users to perform advanced searches using logical operators (AND, OR, NOT) along with string matching conditions.
- Users select multiple columns and define conditions like starts with, ends with, or contains for each column.
- They can specify logical operators to combine these conditions.
- The application executes the search operation based on the specified criteria and displays the results.
- The time taken for the search operation is recorded and displayed.

13 Features

Project Features

Scraping Functionality:

- Ability to scrape at least 1 million entities.
- Scraping tasks can be paused, started, resumed, and stopped.
- Progress bar displaying the progress of tasks/entities scraped.

Entity Attributes:

- Each entity should have a minimum of 8 attributes.

Bonus Feature - URL Input:

- Option for users to input a URL for scraping.

User Interface (UI) Requirements:

- UI design using a Python UI library, preferably PyQt.
- One page displaying the list of chosen entities.

Sorting Functionality:

- UI provides the option for sorting each column.
- Users can choose any sorting algorithm for a specific column.
- Available sorting algorithms include those studied in class and at least three additional algorithms.
- Display sorting time in milliseconds after sorting a column.

Searching Functionality:

- UI allows searching based on each column.
- Users can choose any search algorithm for a specific column.
- Advanced filters for string columns implemented, such as contains, ends with, and starts with.

Multi-Level Sorting:

- Users have the option to sort using multiple columns.

Multi-Column Searching:

- Users can search using composite filters such as AND, OR, and NOT.

14 Conclusion

In conclusion, the E-commerce Data Scraper project represents a significant milestone in the realm of data extraction and management. By implementing advanced scraping functionalities, efficient sorting and searching algorithms, and an intuitive user interface, the project successfully addresses the complexities associated with handling vast amounts of e-commerce data.

Throughout the development process, several key objectives were achieved:

- **Efficient Data Scraping:** The project was designed to handle a substantial amount of data, ensuring that at least 1 million entities could be scraped seamlessly. The ability to pause, start, resume, and stop scraping tasks, coupled with a progress bar indicating task progress, enhances the overall user experience.
- **Robust Entity Attributes:** Each entity within the scraped data was meticulously structured, ensuring a minimum of 8 attributes per entity. This robust attribute framework facilitates detailed analysis and insightful data interpretation.
- **User-Friendly Interface:** The user interface, built using the PyQt Python library, provides a seamless experience for users. The clear presentation of chosen entities, coupled with intuitive controls, ensures that users can navigate through the data effortlessly.
- **Advanced Sorting and Searching:** The implementation of diverse sorting algorithms and search functionalities elevates the project's utility. Users can sort data based on multiple columns and apply intricate search filters, including 'contains,' 'ends with,' and 'starts with,' enhancing the precision of data retrieval.
- **Flexibility and Customization:** The project's flexibility is highlighted by the option for users to input a URL for scraping. Additionally, the ability to choose specific algorithms for sorting and searching ensures a tailored approach to handling diverse data sets.

In essence, the E-commerce Data Scraper project stands as a testament to the fusion of cutting-edge technology and user-centric design. It not only meets but exceeds the project requirements, offering an indispensable tool for businesses and researchers in the ever-expanding realm of e-commerce data analysis.

The success of this project underscores the importance of continuous innovation and adaptability in the field of data science. As technologies evolve and data sets grow larger, projects like this serve as the foundation for insightful analysis, informed decision-making, and the advancement of knowledge in the digital age.