

Amazon Customer Table

by Karl Apolonio

Description of the Problem

The Amazon Customer Table problem aims to enhance order management by incorporating an efficient sorting algorithm, for swiftly processing customer orders. It requires a system capable of quick searches through extensive data fields like client ID and order status while ensuring compatibility with Excel, CSV, and JSON for effective document management.



Algorithm

```
// Insertion sort algorithm
for (let i = 1; i < rows.length; i++) {
  let j = i - 1;
  let current = rows[i];
  let currentValue = getCellValue(current, column);

  // Special handling for currency values
  let isCurrency = currentValue.includes('$');
  if (isCurrency) {
    currentValue = parseFloat(currentValue.replace(/^[^0-9.-]+/g, ""));
  } else {
    // Determine if the column contains numerical data
    let isNumeric = !isNaN(currentValue) && !isNaN(parseFloat(currentValue));
    if (isNumeric) {
      currentValue = parseFloat(currentValue);
    }
  }
}
```

The decrease and conquer algorithm is a problem-solving strategy that progressively reduces a complex problem to simpler instances until reaching base cases, solving them, and thereby conquering the original problem.

Code Implementation

```
const search = document.querySelector('.input-group input'),  
  table_rows = document.querySelectorAll('tbody tr'),  
  table_headings = document.querySelectorAll('thead th');
```

```
// 1. Searching for specific data of HTML table  
search.addEventListener('input', searchTable);  
  
function searchTable() {  
  table_rows.forEach((row, i) => {  
    let table_data = row.textContent.toLowerCase(),  
        search_data = search.value.toLowerCase();  
  
    row.classList.toggle('hide',  
      table_data.indexOf(search_data) < 0);  
    row.style.setProperty('--delay', i / 25 + 's');  
  })  
  
  document.querySelectorAll('tbody tr:not(.hide)').forEach(  
    (visible_row, i) => {  
      visible_row.style.backgroundColor = (i % 2 == 0) ?  
        'transparent' : '#0000000b';  
    });  
}
```

```
table_headings.forEach((head, i) => {  
  let sort_asc = true;  
  head.onclick = () => {  
    table_headings.forEach(head => head.classList.remove('active'));  
    head.classList.add('active');  
  
    document.querySelectorAll('td').forEach(td => td.classList.remove('active'));  
    table_rows.forEach(row => {  
      row.querySelectorAll('td')[i].classList.add('active');  
    })  
  
    head.classList.toggle('asc', sort_asc);  
    sort_asc = head.classList.contains('asc') ? false : true;  
  
    sortTable(i, sort_asc);  
  })  
})
```

Code Implementation

```
function sortTable(column, ascending) {
  const tbody = document.querySelector('tbody');
  let rows = Array.from(tbody.querySelectorAll('tr'));

  // Insertion sort algorithm
  for (let i = 1; i < rows.length; i++) {
    let j = i - 1;
    let current = rows[i];
    let currentValue = getCellValue(current, column);

    // Special handling for currency values
    let isCurrency = currentValue.includes('$');
    if (isCurrency) {
      currentValue = parseFloat(currentValue.replace(/^[^0-9.-]+/g, ""));
    } else {
      // Determine if the column contains numerical data
      let isNumeric = !isNaN(currentValue) && !isNaN(parseFloat(currentValue));
      if (isNumeric) {
        currentValue = parseFloat(currentValue);
      }
    }
  }
}
```

Code Implementation

```
while (j >= 0) {
  let comparisonValue = getCellValue(rows[j], column);
  if (isCurrency || (!isNaN(currentValue) && !isNaN(parseFloat(comparisonValue)))) {
    comparisonValue = parseFloat(comparisonValue.replace(/^[^0-9.-]+/g, ""));
  }
  let condition;
  if (!isNaN(currentValue) && !isNaN(comparisonValue)) {
    // Compare as numbers
    condition = ascending ? currentValue < comparisonValue : currentValue > comparisonValue;
  } else {
    // Compare as lowercase strings for textual data
    condition = ascending ? currentValue.toString().toLowerCase()
      < comparisonValue.toString().toLowerCase() : currentValue.toString().toLowerCase()
      > comparisonValue.toString().toLowerCase();
  }
  if (condition) {
    rows[j + 1] = rows[j];
    j--;
  } else {
    break;
  }
}
```

Code Implementation

```
        rows[j + 1] = current;
    }

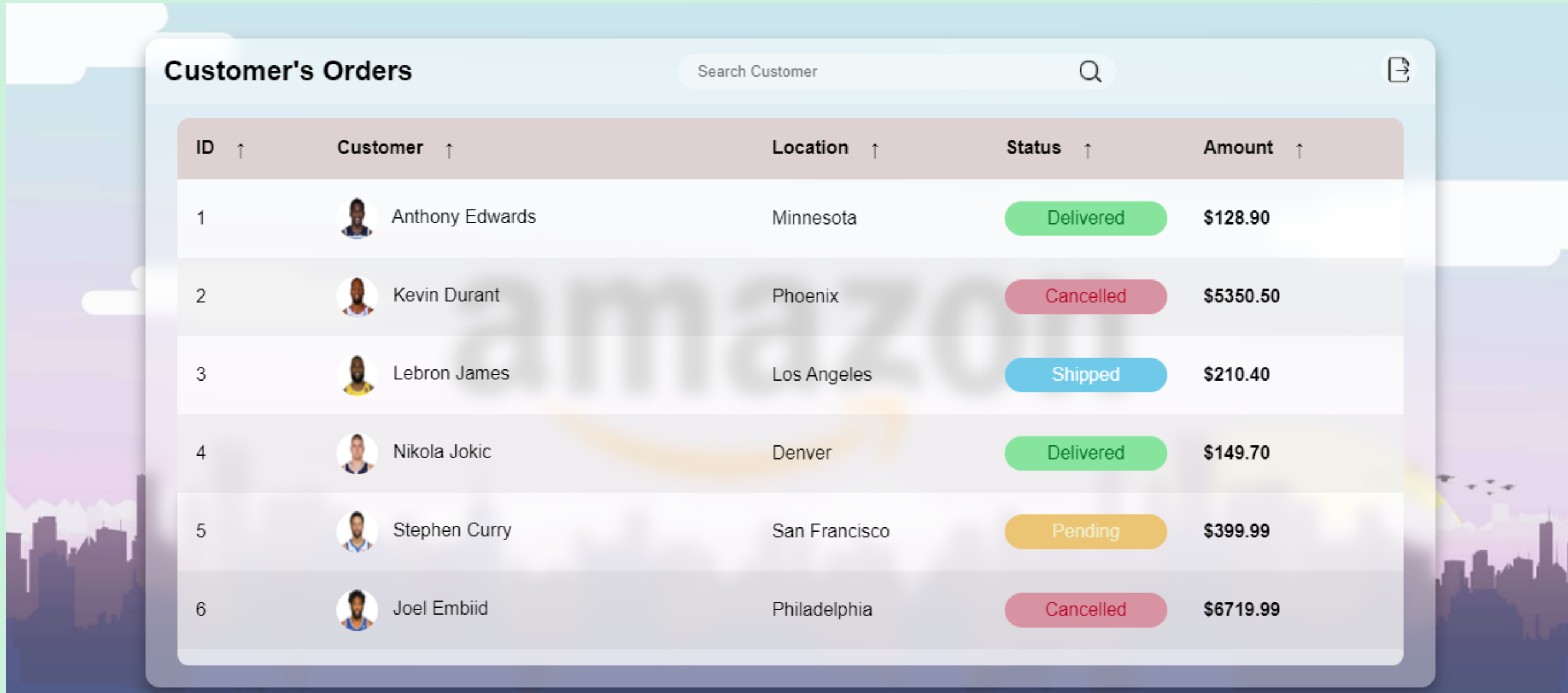
    // Re-appending rows to the tbody in their sorted order
    rows.forEach(row => tbody.appendChild(row));
}

function getCellValue(row, column) {
    return row.querySelectorAll('td')[column].textContent.trim();
}
```







Code Explanation

This code contains functions for managing and interacting with HTML tables. It has a search tool that dynamically filters and hides rows based on user input, a sorting system that allows users to sort the table data by clicking on column headers, and utilities for converting the HTML table to JSON, CSV, and Excel formats. Furthermore, the code includes insertion sort for efficient sorting, special handling for currency values, and compatibility with a variety of file formats, allowing for seamless document management and download capabilities for an enhanced user experience.







Screenshot of the Output






The screenshot displays a web application interface for managing customer orders. The title 'Customer's Orders' is at the top left, followed by a search bar labeled 'Search Customer' and a magnifying glass icon. A table lists the orders with columns for ID, Customer, Location, Status, and Amount. Each row includes a small profile picture of the customer. The status of each order is highlighted in a colored pill: green for 'Delivered', red for 'Cancelled', blue for 'Shipped', and orange for 'Pending'.



ID ↑	Customer ↑	Location ↑	Status ↑	Amount ↑
1	 Anthony Edwards	Minnesota	Delivered	\$128.90
2	 Kevin Durant	Phoenix	Cancelled	\$5350.50
3	 Lebron James	Los Angeles	Shipped	\$210.40
4	 Nikola Jokic	Denver	Delivered	\$149.70
5	 Stephen Curry	San Francisco	Pending	\$399.99
6	 Joel Embiid	Philadelphia	Cancelled	\$6719.99

Screenshot of the Output










Customer's Orders				
Search Customer				
ID ↑	Customer ↑	Location ↑	Status ↑	Amount ↑
1	 Anthony Edwards	Minnesota	Delivered	\$128.90
7	 Giannis Antetokounmpo	Milwaukee	Shipped	\$7199.99
9	 Jalen Brunson	New York	Delivered	\$809.99
6	 Joel Embiid	Philadelphia	Cancelled	\$6719.99
2	 Kevin Durant	Phoenix	Cancelled	\$5350.50
3	 Lebron James	Los Angeles	Shipped	\$210.40

Screenshot of the Output

Customer's Orders				
Delivered				
ID ↑	Customer ↑	Location ↑	Status ↑	Amount ↑
1	 Anthony Edwards	Minnesota	Delivered	\$128.90
4	 Nikola Jokic	Denver	Delivered	\$149.70
9	 Jalen Brunson	New York	Delivered	\$809.99

Customer's Orders				
Cancelled				
ID ↑	Customer ↑	Location ↑	Status ↑	Amount ↑
2	 Kevin Durant	Phoenix	Cancelled	\$5350.50
6	 Joel Embiid	Philadelphia	Cancelled	\$6719.99

Screenshot of the Output

Customer's Orders					Search Customer		
ID ↑	Customer ↑	Location ↑	Status ↑		Export As →		
10	 Luka Doncic	Dallas	Shipped		JSON		
4	 Nikola Jokic	Denver	Delivered		CSV		
8	 Tyrese Haliburton	Indiana	Pending	\$699.99	EXCEL		
3	 Lebron James	Los Angeles	Shipped	\$210.40			
7	 Giannis Antetokounmpo	Milwaukee	Shipped	\$7199.99			
1	 Anthony Edwards	Minnesota	Delivered	\$128.90			

Time Complexity

The searchTable function has a time complexity of $O(n)$, where n is the number of rows in the table, because it performs the search operation by iterating through each row once. The complexity of text matching within each row varies, but assuming a roughly equal distribution of text length across rows, the total effect on time complexity is linear in relation to the number of rows.

Time Complexity

The `sortTable` function's usage of the insertion sort method results in a worst-case time complexity of $O(n^2)$, making it inefficient for large datasets. This quadratic complexity emerges because, for each element in the array, it may compare and shift all preceding elements to determine the correct position for the element within the sorted portion of the array.

Learning Reflection

During the prelim period, I learned about the efficiency and application of algorithms such as the Decrease and Conquer Algorithm, Insertion Sort, Binary Search, and Topological Sort, each offering unique solutions to complex problems. I found Topological Sort challenging due to its reliance on understanding graph theory and directed acyclic graphs, which I addressed by seeking additional resources and practical exercises to strengthen my grasp. Key takeaways include the importance of algorithm selection based on the problem at hand, the efficiency of binary search in sorted lists, and the critical role of order in tasks with Topological Sort.