# Sorting Algorithm Analysis

Ryan Davis

Ryan.Davis3@Marist.edu

October 5, 2024

# Contents

# 1 Introduction

Sorting algorithms are a good tool to study time complexity in computer science.For assignment 1 we looked at the following things :

- Creating stacks and queues using a node class.
- Identifying palindromes using these data structures.
- Shuffling data using the Knuth Shuffle algorithm.
- Sorting data using Selection Sort, Insertion Sort, Merge Sort, and Quick Sort.

# 2 Creating Stacks and Queues with Node Class

## 2.1 Node Class Template

```
template <typename T>
class Node
{
public:
    T data;     // Stores the value of the node, of generic type T
    Node *next; // Pointer to the next node

    Node(T value) : data(value), next(nullptr) {} // Constructor
};
```

Listing 1: Node Class Template

The `Node` class template serves as the foundational building block for both the `Stack` and `Queue` classes. It is a generic container that can store any data type specified by `T`.

**Explanation:**

- **Template Declaration**: Allows the `Node` class to handle any data type.
- **Data Members**:
  - `data`: Holds the value of the node.
  - `next`: A pointer to the next node in the data structure.
- **Constructor**: Initializes the `data` with the provided value and sets `next` to `nullptr`, indicating the end of the structure or an uninitialized next node.

This simple design lets us create linked structures where each node points to the next, enabling the dynamic creation of stacks and queues.

## 2.2 Stack Class Template

```
template <typename T>
class Stack
{
private:
    Node<T> *top; // Pointer to the top node

public:
    Stack() : top(nullptr) {} // Constructor that initializes an empty stack

    void push(T value)
    {
        Node<T> *newNode = new Node<T>(value); // Create a new node with the given value
        newNode->next = top; // Next pointer points to current top
```

```
        top = newNode;       // New node becomes the top of the stack
    }

    T pop()
    {
        if (isEmpty())
        {
            throw std::out_of_range("Stack is empty"); // Error handling
        }
        Node<T> *temp = top;       // Temporary storage of current top
        T poppedValue = top->data; // Store the data to return
        top = top->next;           // Update the top pointer to the next node
        delete temp;               // Delete the old top node
        return poppedValue;        // Return the popped value
    }

    bool isEmpty()
    {
        return top == nullptr; // Stack is empty if top is nullptr
    }

    ~Stack()
    {
        while (!isEmpty())
        {
            pop(); // Clean up all nodes
        }
    }
};
```

Listing 2: Stack Class Template

**Explanation:**

- **Private Member**:
  - `top`: Points to the top node of the stack.
- **Constructor**: Initializes an empty stack by setting `top` to `nullptr`.
- **Push Method**:
  - Creates a new node with the given value.
  - Sets the `next` pointer of the new node to the current `top`.
  - Updates `top` to point to the new node.
  - This effectively places the new node on top of the stack.
- **Pop Method**:
  - Checks if the stack is empty and throws an exception if it is.
  - Stores the current `top` node in a temporary variable.
  - Retrieves the data to return.
  - Updates `top` to the next node in the stack.
  - Deletes the old top node to free memory.
  - Returns the popped value.
- **isEmpty Method**: Returns `true` if `top` is `nullptr`, indicating the stack is empty.
- **Destructor**: Cleans up any remaining nodes when the stack is destroyed to prevent memory leaks.

The stack operates on a Last-In-First-Out (LIFO) principle, where elements are added and removed from the top of the stack.

## 2.3 Queue Class Template

```cpp
template <typename T>
class Queue
{
private:
    Node<T> *front; // Pointer to the front node
    Node<T> *rear;  // Pointer to the rear node

public:
    Queue() : front(nullptr), rear(nullptr) {} // Constructor initializes an empty queue

    void enqueue(T value)
    {
        Node<T> *newNode = new Node<T>(value); // Create a new node with the given value
        if (rear != nullptr)
        {
            rear->next = newNode; // Link the new node after the current rear
        }
        rear = newNode; // New node becomes the rear of the queue
        if (front == nullptr)
        { // If the queue was empty, front is also the new node
            front = rear;
        }
    }

    T dequeue()
    {
        if (isEmpty())
        {
            throw std::out_of_range("Queue is empty"); // Error handling
        }
        Node<T> *temp = front;          // Temporary storage of current front
        T dequeuedValue = front->data; // Store the data to return
        front = front->next;           // Update front to the next node
        if (front == nullptr)
        { // If the queue is now empty, rear should also be nullptr
            rear = nullptr;
        }
        delete temp;                   // Delete the old front node
        return dequeuedValue;          // Return the dequeued value
    }

    bool isEmpty()
    {
        return front == nullptr; // Queue is empty if front is nullptr
    }

    ~Queue()
    {
        while (!isEmpty())
        {
            dequeue(); // Clean up all nodes
        }
    }
};
```

Listing 3: Queue Class Template

**Explanation:**

- **Private Members**:

  - `front`: Points to the front node of the queue.
  - `rear`: Points to the rear node of the queue.
- **Constructor**: Initializes an empty queue by setting both `front` and `rear` to `nullptr`.
- **Enqueue Method**:
  - Creates a new node with the given value.
  - If the queue is not empty, links the new node after the current `rear`.
  - Updates `rear` to the new node.
  - If the queue was empty, sets `front` to the new node.
  - This adds the new element to the end of the queue.
- **Dequeue Method**:
  - Checks if the queue is empty and throws an exception if it is.
  - Stores the current `front` node in a temporary variable.
  - Retrieves the data to return.
  - Updates `front` to the next node in the queue.
  - If the queue becomes empty, sets `rear` to `nullptr`.
  - Deletes the old front node to free memory.
  - Returns the dequeued value.
- **isEmpty Method**: Returns `true` if `front` is `nullptr`, indicating the queue is empty.
- **Destructor**: Cleans up any remaining nodes when the queue is destroyed to prevent memory leaks.

The queue operates on a First-In-First-Out (FIFO) principle, where elements are added to the rear and removed from the front.

# 3 Finding Palindromes

```
bool isPalindrome(const std::string &str)
{
    Stack<char> stack;
    Queue<char> queue;

    // Load stack and queue ignoring non-alphabetic characters
    for (char ch : str)
    {
        if (std::isalpha(ch))
        {
            char lowerCh = std::tolower(ch);
            stack.push(lowerCh);
            queue.enqueue(lowerCh);
        }
    }

    // Compare characters from stack and queue
    while (!stack.isEmpty() && !queue.isEmpty())
    {
        if (stack.pop() != queue.dequeue())
        {
            return false;
        }
    }
```

```
25
26      return true; // The string is a palindrome
27  }
```

Listing 4: Palindrome Check Function

**Explanation:**

The `isPalindrome` function determines whether a given string is a palindrome, considering only alphabetic characters and ignoring case.

- **Initialization**:
  - Creates an instance of `Stack<char>` and `Queue<char>` to store characters.
- **Loading Data Structures**:
  - Iterates over each character in the input string.
  - Checks if the character is alphabetic using `std::isalpha`.
  - Converts the character to lowercase for case-insensitive comparison.
  - Pushes the character onto the stack and enqueues it into the queue.
- **Comparison**:
  - Continues looping as long as neither the stack nor the queue is empty.
  - Pops a character from the stack and dequeues a character from the queue, then compares them.
  - If the characters are not the same, the function returns `false`, indicating the string is not a palindrome.
  - If all characters match, the function returns `true` at the end.
- **Why It Works**:
  - A stack reverses the order of insertion due to its LIFO nature.
  - A queue maintains the original order due to its FIFO nature.
  - By comparing the characters from both data structures, we effectively compare the string to its reverse.

# 4 Making the Shuffle (Knuth Shuffle)

```
1  void knuthShuffle(std::vector<std::string> &arr)
2  {
3      std::srand(static_cast<unsigned int>(std::time(nullptr))); // Seed RNG with current time
4
5      for (int i = arr.size() - 1; i > 0; --i)
6      {
7          int j = std::rand() % (i + 1); // Generate random index between 0 and i
8          std::swap(arr[i], arr[j]);     // Swap current element with element at random index
9      }
10 }
```

Listing 5: Knuth Shuffle Function

**Explanation:**

The `knuthShuffle` function randomizes the order of elements in a vector using the Fisher-Yates shuffle algorithm.

- **Seeding the Random Number Generator**:
  - Uses `std::srand` with the current time to seed the random number generator, ensuring different results on each run.
- **Shuffling Logic**:
  - Starts from the end of the array and moves backward.
  - Generates a random index `j` between 0 and `i`.
  - Swaps the element at index `i` with the element at index `j`.
  - Repeats this process for each element, effectively shuffling the array uniformly.
- **Why It Works**:
  - The algorithm ensures that each possible permutation of the array is equally likely.
  - By swapping each element with a randomly selected element, we avoid bias in the shuffle.

# 5 Sorting Algorithms

In this section, we discuss four different sorting algorithms, their implementations, and explanations of how the code works, referencing line numbers for clarity.

## 5.1 Selection Sort

```
1  void selectionSort(std::vector<std::string> &arr, int &comparisonCount)
2  {
3      int n = arr.size();
4
5      for (int i = 0; i < n - 1; ++i)
6      {
7          int min = i; // Assume the current position holds the minimum
8
9          // Find the minimum element in the unsorted part
10         for (int j = i + 1; j < n; ++j)
11         {
12             comparisonCount++; // Increment comparison counter
13             if (arr[j] < arr[min])
14             {
15                 min = j; // Update index of minimum element
16             }
17         }
18
19         // Swap the found minimum element with the first element of the unsorted part
20         if (min != i)
21         {
22             std::swap(arr[i], arr[min]);
23         }
24     }
25 }
```

Listing 6: Selection Sort Implementation

**Explanation:**

Selection Sort repeatedly selects the minimum element from the unsorted portion of the array and moves it to the beginning.

Here's how the code works:

- **Line 3**: Initialize `n` to the size of the array.

- **Lines 5–17**: The outer loop iterates over each position in the array from `i = 0` to `n - 2`.
- **Line 6**: Assume the element at index `i` is the minimum.
- **Lines 9–14**: The inner loop searches for the smallest element in the unsorted portion of the array.
  - **Line 10**: Increment the comparison counter.
  - **Lines 11–13**: If a smaller element is found, update `min` with its index.
- **Lines 17–19**: After finding the minimum, swap it with the element at index `i` if necessary.

This process is repeated until the entire array is sorted.

## 5.2 Insertion Sort

```
1  void insertionSort(std::vector<std::string> &arr, int &comparisonCount)
2  {
3      int n = arr.size();
4      for (int i = 1; i < n; ++i)
5      {
6          std::string key = arr[i]; // Element to be inserted
7          int j = i - 1;
8
9          // Move elements greater than key one position ahead
10         while (j >= 0 && arr[j] > key)
11         {
12             comparisonCount++;    // Increment comparison counter
13             arr[j + 1] = arr[j];  // Shift element to the right
14             j = j - 1;
15         }
16         comparisonCount++;        // For the last comparison when the while loop condition fails
17         arr[j + 1] = key;         // Insert the key at the correct position
18     }
19 }
```

Listing 7: Insertion Sort Implementation

**Explanation:**

Insertion Sort builds the final sorted array one item at a time.

Here's how the code operates:

- **Line 3**: Initialize `n` as the size of the array.
- **Lines 4–16**: The outer loop iterates from `i = 1` to `n - 1`.
- **Line 5**: The `key` variable holds the current element to be inserted.
- **Line 6**: Initialize `j` to the index before `i`.
- **Lines 9–13**: The inner loop shifts elements greater than `key` to the right.
  - **Line 10**: Increment the comparison counter.
  - **Line 11**: Move the element at `arr[j]` one position ahead.
  - **Line 12**: Decrement `j` to move backward through the array.
- **Line 14**: Increment the comparison counter for the last failed comparison.
- **Line 15**: Place the `key` in its correct sorted position.

This process inserts each element into its proper place within the sorted portion of the array.

## 5.3 Merge Sort

```cpp
void mergeSort(std::vector<std::string> &arr, int leftIndex, int middleIndex, int rightIndex, int
     &comparisonCount)
{
    // Determine the sizes of the two subarrays
    int leftSubarraySize = middleIndex - leftIndex + 1;
    int rightSubarraySize = rightIndex - middleIndex;

    // Create temporary vectors
    std::vector<std::string> leftSubarray(leftSubarraySize);
    std::vector<std::string> rightSubarray(rightSubarraySize);

    // Copy data to temporary vectors
    for (int i = 0; i < leftSubarraySize; i++)
    {
        leftSubarray[i] = arr[leftIndex + i];
    }
    for (int j = 0; j < rightSubarraySize; j++)
    {
        rightSubarray[j] = arr[middleIndex + 1 + j];
    }

    // Merge the temporary vectors back into arr
    int leftPointer = 0, rightPointer = 0, mergedPointer = leftIndex;

    while (leftPointer < leftSubarraySize && rightPointer < rightSubarraySize)
    {
        comparisonCount++; // Increment comparison counter
        if (leftSubarray[leftPointer] <= rightSubarray[rightPointer])
        {
            arr[mergedPointer] = leftSubarray[leftPointer];
            leftPointer++;
        }
        else
        {
            arr[mergedPointer] = rightSubarray[rightPointer];
            rightPointer++;
        }
        mergedPointer++;
    }

    // Copy any remaining elements
    while (leftPointer < leftSubarraySize)
    {
        arr[mergedPointer] = leftSubarray[leftPointer];
        leftPointer++;
        mergedPointer++;
    }
    while (rightPointer < rightSubarraySize)
    {
        arr[mergedPointer] = rightSubarray[rightPointer];
        rightPointer++;
        mergedPointer++;
    }
}

// Helper function for recursive Merge Sort
void mergeSortHelper(std::vector<std::string> &arr, int left, int right, int &comparisonCount)
{
    if (left < right)
    {
        int middle = left + (right - left) / 2;

        // Recursively sort the left and right halves
        mergeSortHelper(arr, left, middle, comparisonCount);
        mergeSortHelper(arr, middle + 1, right, comparisonCount);

```

```
66          // Merge the sorted halves
67          mergeSort(arr, left, middle, right, comparisonCount);
68      }
69  }
```

Listing 8: Merge Sort Implementation

**Explanation:**

Merge Sort is a divide-and-conquer algorithm that splits the array into halves, sorts them, and then merges them back together.

Here's how the code functions:

- **Recursive Splitting** (in `mergeSortHelper`):
  - **Line 41**: Check if the array can be split further.
  - **Line 42**: Calculate the middle index.
  - **Lines 45–46**: Recursively call `mergeSortHelper` on the left and right halves.
  - **Line 49**: Merge the sorted halves using the `mergeSort` function.
- **Merging Process** (in `mergeSort`):
  - **Lines 3–6**: Determine the sizes of the left and right subarrays.
  - **Lines 9–14**: Copy data into temporary subarrays.
  - **Lines 17–34**: Merge the subarrays back into the main array.
    - **Line 20**: Increment the comparison counter.
    - **Lines 21–29**: Compare elements and insert the smaller one into the main array.
  - **Lines 37–43**: Copy any remaining elements from the subarrays.

This process ensures that at each level of recursion, the subarrays are sorted and merged correctly.

## 5.4 Quick Sort

```
1  int partitionArray(std::vector<std::string> &dataArray, int startIndex, int endIndex, int &
       comparisonCount)
2  {
3      // Pivot is chosen as the last element
4      std::string pivotElement = dataArray[endIndex];
5
6      // Index of smaller element
7      int smallerElementIndex = startIndex - 1;
8
9      for (int currentIndex = startIndex; currentIndex <= endIndex - 1; currentIndex++)
10     {
11         // Increment comparison count
12         comparisonCount++;
13
14         // If current element is smaller than pivot
15         if (dataArray[currentIndex] < pivotElement)
16         {
17             smallerElementIndex++;
18             std::swap(dataArray[smallerElementIndex], dataArray[currentIndex]);
19         }
20     }
21
22     // Place pivot in the correct position
23     std::swap(dataArray[smallerElementIndex + 1], dataArray[endIndex]);
24
```

```
25      return smallerElementIndex + 1; // Return pivot index
26  }
27
28  // Quick Sort Implementation
29  void quickSortArray(std::vector<std::string> &dataArray, int startIndex, int endIndex, int &
        comparisonCount)
30  {
31      // Base condition: at least two elements
32      if (startIndex < endIndex)
33      {
34          // Partition the array
35          int pivotIndex = partitionArray(dataArray, startIndex, endIndex, comparisonCount);
36
37          // Recursively sort elements before and after partition
38          quickSortArray(dataArray, startIndex, pivotIndex - 1, comparisonCount);
39          quickSortArray(dataArray, pivotIndex + 1, endIndex, comparisonCount);
40      }
41  }
```

Listing 9: Partition Function for Quick Sort

**Explanation:**

Quick Sort is another divide-and-conquer algorithm that selects a pivot element and partitions the array.

Here's how the code works:

- **Partition Function (`partitionArray`)**:
  - **Line 3**: Select the pivot as the last element in the array segment.
  - **Line 6**: Initialize `smallerElementIndex` to one less than `startIndex`.
  - **Lines 8–17**: Iterate over the array segment.
    - **Line 10**: Increment the comparison counter.
    - **Lines 12–16**: If the current element is less than the pivot, increment `smallerElementIndex` and swap elements to position smaller elements before the pivot.
  - **Line 20**: Place the pivot in its correct sorted position.
  - **Line 22**: Return the index of the pivot.
- **Quick Sort Function (`quickSortArray`)**:
  - **Line 28**: Check if the array segment has at least two elements.
  - **Line 31**: Partition the array and get the pivot index.
  - **Lines 34–35**: Recursively apply Quick Sort to the subarrays before and after the pivot.

This method efficiently sorts the array by recursively partitioning and sorting subarrays.

## 6 Test Cases and Results

Multiple test runs were conducted to measure the performance of each sorting algorithm in terms of the number of comparisons.

Table 6.1: Comparison Counts for Sorting Algorithms

| Test | Selection Sort | Insertion Sort | Merge Sort | Quick Sort |
|---|---|---|---|---|
| 1 | 221,445 | 109,152 | 5,442 | 7,103 |
| 2 | 221,445 | 114,483 | 5,415 | 6,406 |
| 3 | 221,445 | 109,128 | 5,415 | 6,535 |
| 4 | 221,445 | 109,809 | 5,417 | 7,787 |
| 5 | 221,445 | 113,839 | 5,424 | 7,075 |
| 6 | 221,445 | 110,957 | 5,420 | 6,881 |
| 7 | 221,445 | 109,311 | 5,435 | 6,343 |
| 8 | 221,445 | 111,295 | 5,428 | 6,234 |
| 9 | 221,445 | 113,412 | 5,431 | 6,653 |
| 10 | 221,445 | 113,869 | 5,437 | 6,493 |
| **Average** | 221,445 | 111,425 | 5,426 | 6,751 |

**Observations:**

- **Selection Sort** consistently performed the same number of comparisons due to its deterministic nature.
- **Insertion Sort** showed variability depending on the initial order after shuffling.
- **Merge Sort** had minimal variation and consistently low comparison counts.
- **Quick Sort** showed some variability due to the randomness in pivot selection and initial ordering.

# 7 Asymptotic Analysis

In this section, we discuss the mathematical foundations behind the expected efficiency, upper and lower bounds of each sorting algorithm.

## 7.1 Selection Sort: $\mathcal{O}(n^2)$

**Time Complexity Contribution:**

- The quadratic time complexity arises from the nested loops, where the inner loop depends linearly on the size of the unsorted portion.

**Mathematical Explanation:**

Selection Sort always performs a fixed number of comparisons, regardless of the initial order.

$$C(n) = \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2}$$

**Expected Efficiency:**

- **Best Case**: $\mathcal{O}(n^2)$
- **Average Case**: $\mathcal{O}(n^2)$
- **Worst Case**: $\mathcal{O}(n^2)$

**Upper and Lower Bounds:**

Since the number of comparisons is fixed, both the upper and lower bounds are $\Theta(n^2)$.

## 7.2 Insertion Sort: $\mathcal{O}(n^2)$

**Time Complexity Contribution:**

- The nested nature of the loops, where the number of operations in the inner loop depends on the current index `i`, leads to a quadratic time complexity.

**Mathematical Explanation:**

The number of comparisons depends on the initial ordering.

$$\text{Best Case Comparisons} = n - 1$$

$$\text{Worst Case Comparisons} = \frac{n(n-1)}{2}$$

$$\text{Average Case Comparisons} \approx \frac{n^2}{4}$$

**Expected Efficiency:**

- **Best Case**: $\mathcal{O}(n)$
- **Average Case**: $\mathcal{O}(n^2)$
- **Worst Case**: $\mathcal{O}(n^2)$

**Upper and Lower Bounds:**

$$\Omega(n) \leq C(n) \leq \mathcal{O}(n^2)$$

## 7.3 Merge Sort: $\mathcal{O}(n \log n)$

- The recursion depth is $\log n$, as the array is continually divided in half.
- At each level of recursion, the merging process handles all elements, resulting in $\mathcal{O}(n)$ operations.
- Multiplying the number of levels by the operations at each level gives a total time complexity of $\mathcal{O}(n \log n)$.

**Mathematical Explanation:**

Merge Sort's number of comparisons can be represented as:

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1$$

Solving the recurrence:

$$C(n) = n \log n - n + 1$$

**Expected Efficiency:**

- **Best Case**: $\mathcal{O}(n \log n)$
- **Average Case**: $\mathcal{O}(n \log n)$
- **Worst Case**: $\mathcal{O}(n \log n)$

**Upper and Lower Bounds:**

The number of comparisons is tightly bounded by $\Theta(n \log n)$.

### 7.4 Quick Sort: Average Case $\mathcal{O}(n \log n)$, Worst Case $\mathcal{O}(n^2)$

**Time Complexity Contribution:**

- In the average case, the pivot divides the array into two roughly equal halves, resulting in a recursion depth of $\log n$.
- Each partitioning operation involves $\mathcal{O}(n)$ comparisons (the for loop iterates over the array segment).
- Thus, the average time complexity is $\mathcal{O}(n \log n)$.
- In the worst case (e.g., when the array is already sorted), the pivot does not partition the array effectively, leading to $\mathcal{O}(n)$ recursion depth and total time complexity of $\mathcal{O}(n^2)$.

**Mathematical Explanation:**

Quick Sort's performance depends heavily on how balanced the partitions are after each pivot selection. In the best and average cases, the pivot divides the array into two nearly equal halves, leading to a recurrence relation similar to:

$$C(n) = 2C\left(\frac{n}{2}\right) + n$$

Solving this recurrence, we get:

$$C(n) = \mathcal{O}(n \log n)$$

However, in the worst case, where the pivot is always the smallest or largest element, the partitions are highly unbalanced, and the recurrence becomes:

$$C(n) = C(n-1) + n$$

Solving this recurrence yields:

$$C(n) = \mathcal{O}(n^2)$$

**Expected Efficiency:**

- **Best Case**: $\mathcal{O}(n \log n)$ (Balanced partitions)
- **Average Case**: $\mathcal{O}(n \log n)$
- **Worst Case**: $\mathcal{O}(n^2)$ (Unbalanced partitions)

**Upper and Lower Bounds:**

$$\Omega(n \log n) \leq C(n) \leq \mathcal{O}(n^2)$$

## 8 Conclusion

The sort results align with the expectations for each sorting algorithm's time complexity. Selection Sort and Insertion Sort, both with $\mathcal{O}(n^2)$ time complexity, had significantly more comparisons than Merge Sort and Quick Sort, which have $\mathcal{O}(n \log n)$ average-case time complexity.

Merge Sort consistently exhibited reliable performance due to its deterministic divide-and-conquer strategy. In contrast, Quick Sort performed slightly less efficiently, averaging approximately 1,300 more comparisons in our tests. This discrepancy can be attributed to suboptimal pivot selection in Quick Sort. When pivot choices

result in unbalanced partitions, Quick Sort's performance degrades toward its worst-case time complexity of $\mathcal{O}(n^2)$. Imperfect pivot choices cause the recursive calls to process significantly larger subarrays, increasing the number of comparisons.

## References


[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, MIT Press, 2009.

[2] Robert Sedgewick, *Algorithms in C++*, Third Edition, Addison-Wesley, 1998.

[3] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Second Edition, Addison-Wesley, 1998.