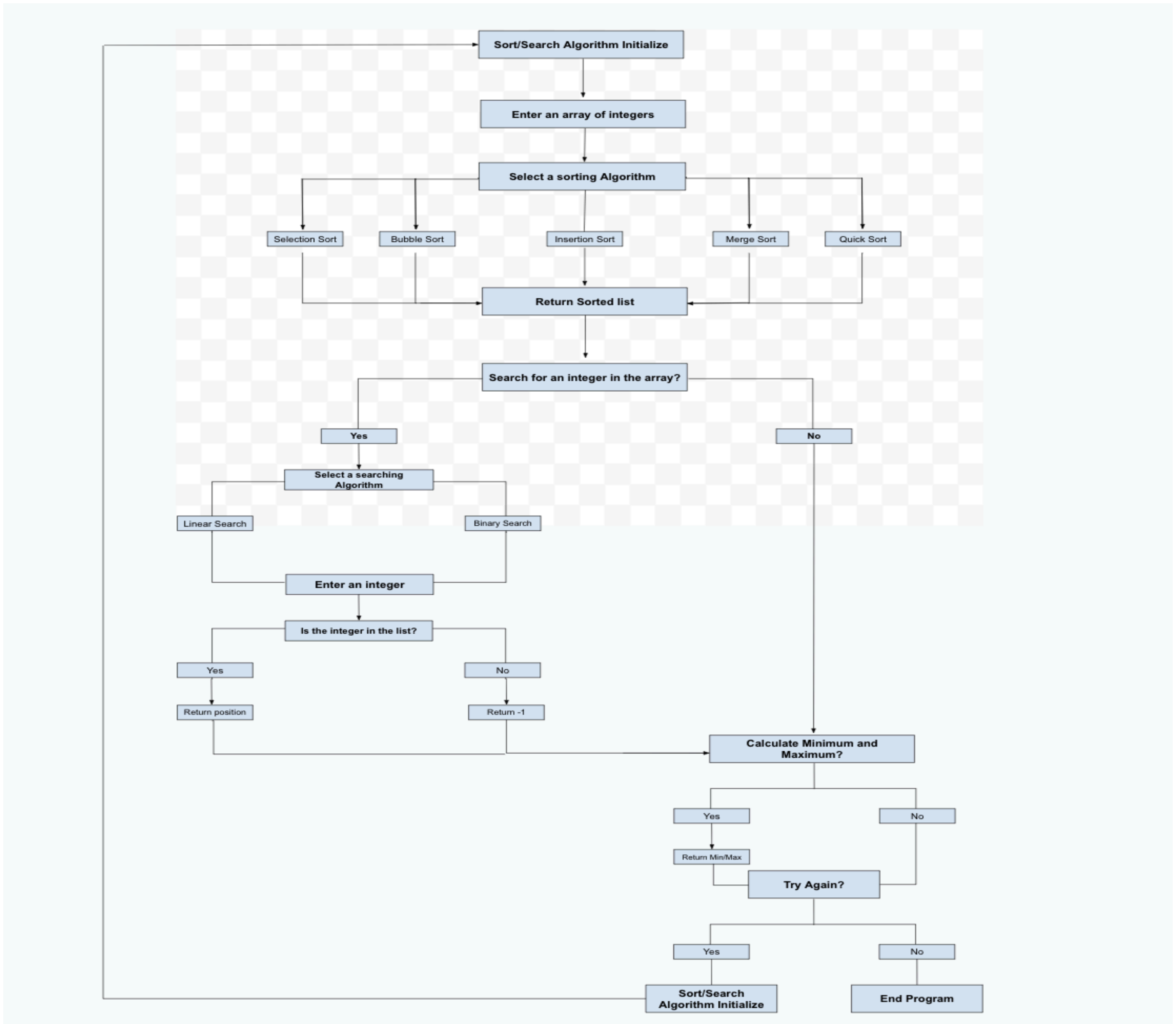
Visual representation of sorting/searching algorithms

Mikey Hainey

My program will include sorting algorithms such as Selection sort, Bubble sort, insertion sort, Merge sort, and Quick sort. The program will also include search algorithms such as linear search and binary search. The program will ask the user to input an array of integers, then will ask the user which sorting algorithm they would like to use; after the array is sorted, it will ask if the user wants to search for an integer in the array using either Linear or Binary search. Then, the program will ask the user if they will calculate the array's minimum and maximum values. After that, the program will ask the user if they would like to use the program again.

Decision tree for the Sort/Search Algorithm Program(Figure 1.1)


Sort/Search Algorithm Initialize
Enter an array of integers
Select a sorting Algorithm
Selection Sort
Bubble Sort
Insertion Sort
Merge Sort
Quick Sort
Return Sorted list
Search for an integer in the array?
Yes
No
Select a searching Algorithm
Linear Search
Binary Search
Enter an integer
Is the integer in the list?
Yes
No
Return position
Return -1
Calculate Minimum and Maximum?
Yes
No
Return Min/Max
Try Again?
Yes
No
Sort/Search Algorithm Initialize
End Program

Big-O Notation

What is Big-O Notation?

- Big-O notation is a mathematical concept applied in computer science to describe an algorithm's performance or complexity. It expresses how an algorithm's running time or space requirements increase as a function of its input size. Big-O gives the upper bound on an algorithm's growth rate, helping us compare the efficiency of different algorithms.

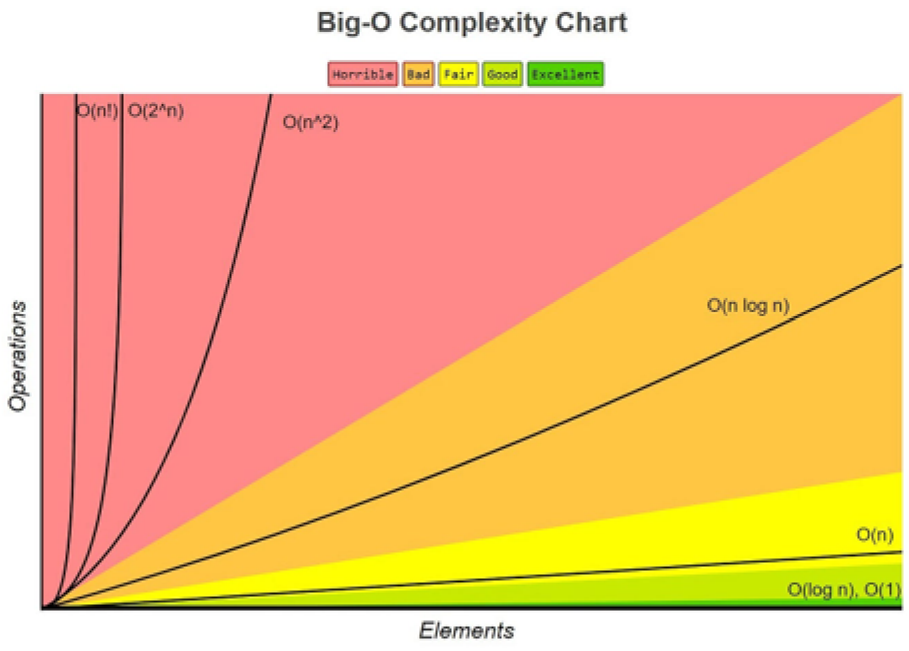


Figure 1.2

Why Big-O is Important

Big-O is essential because it enables developers to:

- Predict Performance: Understand how an algorithm will behave for significant
- inputs.
- Compare Algorithms: Choose the most efficient algorithm for a particular task.
- Optimize Code: Locate bottlenecks and enhance the efficiency of the program.

Main Concepts in Big-O

- Input Size (n):
  The size of the input data, such as the number of elements in an array or the size of a number in digits.
- Big-O describes how runtime or memory usage grows with the size
  growth rate.
- Big-O considers an algorithm's runtime as the dominant term. Constants and lesser terms should be addressed due to their negligible effects in large numbers.

Big-O Notations In Practice

Common Big-O Notations:

Big-O	Name	Description
1. $O(1)$	Constant Time	It takes the same time to execute, irrespective of the input size.
2. $O(\log n)$	Logarithmic Time	It grows slowly as the input size increases.
3. $O(n)$	Linear Time	It grows directly in proportion to the input size.
4. $O(n\log n)$	Linearithmic Time	A combination of linear and logarithmic growth.
5. $O(n^2)$	Quadratic Time	It grows proportionally to the square of the input size.
6. $O(2^n)$	Exponential Time	Performance doubles with each additional input.
7. $O(n!)$	Factorial Time	Performance grows extremely fast, often impractical for significant inputs.

Figure 1.3

Notations in Practice:

1. Ignoring Constants:
Big-O notation focuses on the dominant term and ignores constants or lower-order terms.
Examples:

1. $F(n) = 5n + 10$ For larger n, 5n dominates, so $O(f(n)) = O(n)$

2. Nested Loops:
   - When analyzing nested loops, the number of iterations for each loop is multiplied.
     Code Example:

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

The *Big-O* notation for the code would be *$O(n^2)$*

Best, Average, & Worst Case Scenarios

Best Case:

Definition:

A best-case represents the situation where the algorithm does the least work possible. This would show the algorithm's performance when the input is most favorable. Significance: Even though the best case illustrates how an algorithm can be fast, it doesn't necessarily imply its practicality, as actual data rarely meets the best-case scenario.

Examples:

Linear Search:

In the best case, the target element is the first element of the array.

Time Complexity:

O(1), since only one comparison needs to be performed.

Bubble Sort:

The best case is when the array is already sorted.

Time Complexity:

O(n), since one pass will be enough to confirm that it is indeed sorted.

Average Case:

Definition:

The average-case scenario describes the algorithm's expected performance across all possible inputs.

Examples:

Linear Search:

The average case occurs when the target element is in the middle of the array. Thus, on average, the algorithm must examine half the array.

Time Complexity:

$O(n/2) = O(n)$ since half of the comparisons need to be performed.

Quick Sort:

The average case assumes that the pivot divides the array into two roughly equal parts at each step.

Time Complexity:

$O(n \log n)$

Worst Case:

Definition:

The worst-case scenario describes the maximum work the algorithm may need to perform.

Examples:

Linear Search:

The worst case is when the target element is not in the array, requiring the algorithm to search all n elements.

Time Complexity:

$O(n)$

Quick Sort:

The worst case occurs when the pivot is consistently the most minor or significant element, resulting in unbalanced partitions.

Time Complexity:

$O(n^2)$

Why All Three Cases Matter:

Best Case:

It is essential to understand the algorithm's optimal performance.
It helps evaluate how the algorithm handles favorable inputs.

Average Case:

This is crucial for practical scenarios where inputs are unpredictable.
Provides realistic expectations for performance.

Worst Case:

Essential for ensuring reliability in all situations.
It helps design systems that can handle the algorithm's slowest performance.

Introduction to the Sorting/Searching algorithm program

Overview and Major Concepts Sorting Algorithms:

Selection Sort:

Repeatedly selects the smallest element from the unsorted part of the list and places it in the proper position.

Bubble Sort:

Repeatedly swaps adjacent elements if they are in the wrong order.

Insertion Sort:

Repeatedly insert elements into their proper position within a sorted portion of the list.

Merge Sort:

A divide-and-conquer algorithm that splits the list recursively sorts each half and then merges the sorted halves.

Quick Sort:

Partitions the list around a pivot and recursively sorts the partitions.

Searching Algorithms:

Linear Search:

It will search each element in the list sequentially for the target value.

Binary Search:

Searches for the target in a sorted list with a divide-and-conquer approach efficiently.

Other Features:

Min-Max Algorithm:

This algorithm identifies a list's minimum and maximum values in just one pass. The interactive user interface enables users to insert input data, select the sorting methods, search for desired values, and compute the minimum/maximum values.

Error Handling:

Handles invalid input nicely by asking the user to enter valid input.

Algorithmic Complexity:

Best and worst-case complexities are provided, aiding users in understanding the performance of each algorithm.

Sorting Functions Code Snippets

Selection Sort:

```
##Purpose: Sort an array using the Selection sort algorithm.
##Best Case: O(n^2)
def selectionSort(userlist):
    iterationCount = 0
    print(f"The original list: {userlist}")
    n = len(userlist)
    for bottom in range(n-1):
        mp = bottom
        for i in range(bottom+1, n):
            if userlist[i] < userlist[mp]:
                mp = i
                iterationCount += 1
        userlist[bottom], userlist[mp] = userlist[mp], userlist[bottom]
    print(f"The list at {iterationCount} iterations using Selection Sort: {userlist}")
    return userlist
```

Bubble Sort:

```
##Purpose: Sort an array using the Bubble sort algorithm.
##Best Case: O(n^2)
def bubbleSort(userlist):
    n = len(userlist)
    iterationCount = 0
    print(f"The original list: {userlist}")
    for i in range(n - 1):
        swapped = False
        for j in range(n - i - 1):
            if userlist[j] > userlist[j + 1]:  ##Compare adjacent elements
                userlist[j], userlist[j + 1] = userlist[j + 1], userlist[j]
                swapped = True
                iterationCount += 1
        if not swapped:  ##Exit early if no swaps in the current pass
        break
    print(f"The list at {iterationCount} iterations using Bubble Sort: {userlist}")
    return userlist  ##Return the sorted list
```

Sorting Functions Code Snippets

Insertion Sort:

```
##Purpose: Sort an array using the insertion sort algorithm.
##Best Case: O(n^2)
def insertionSort(userlist):
    n = len(userlist)
    iterationCount = 0
    print(f"The original list: {userlist}")
    for i in range(1, n):
        currentElement = userlist[i]  # Correct current element for insertion
        j = i - 1
        # Shift elements to the right if greater than currentElement
        while j >= 0 and userlist[j] > currentElement:
            userlist[j + 1] = userlist[j]
            j -= 1
            iterationCount += 1
        userlist[j + 1] = currentElement  # Insert current element at correct position
    print(f"The list at {iterationCount} iterations using Insertion Sort: {userlist}")
    return userlist
```

Merge Sort:

```
##Purpose: Sort an array using the Merge Sort algorithm.
##Best Case: O(n log n)
def mergeSort(userlist):
    if len(userlist) > 1:
        mid = len(userlist) // 2  # Find the middle of the array
        left_half = userlist[:mid]  # Split into left half
        right_half = userlist[mid:]  # Split into right half
        mergeSort(left_half)  # Recursively sort the left half
        mergeSort(right_half)  # Recursively sort the right half
        i = j = k = 0
        # Merge the sorted halves
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            userlist[k] = left_half[i]
            i += 1
```

Sorting Functions Code Snippets

```
         (Merge sort continued from the previous page…)
         else:
            userlist[k] = right_half[j]
            j += 1
        k += 1
    while i < len(left_half):
        userlist[k] = left_half[i]
        i += 1
        k += 1
    while j < len(right_half):
        userlist[k] = right_half[j]
        j += 1
        k += 1
return userlist
```

Quick Sort:

```
##Purpose: Sort an array using the Quick Sort algorithm.
##Best Case: O(n log n)
def quickSort(userlist):
    if len(userlist) <= 1:
        return userlist  # Base case: arrays with 1 or 0 elements are sorted
    else:
        pivot = userlist[len(userlist) // 2]  # Choose the middle element as the pivot
        less_than_pivot = [x for x in userlist if x < pivot]
        equal_to_pivot = [x for x in userlist if x == pivot]
        greater_than_pivot = [x for x in userlist if x > pivot]
        # Recursively sort the partitions
        return quickSort(less_than_pivot) + equal_to_pivot + quickSort(greater_than_pivot)
```

Searching Functions Code Snippets

Linear Search:

```
##Purpose: Search the users list for an integer
##Best Case: O(1), Worst Case: O(n)
def linearSearch(userlist):
    userElement = int(input("Enter an integer you would like to find in the list: "))
    print()
    n = len(userlist)
    for i in range(n):
        if userlist[i] == userElement:
            print(f"{userElement} was found at position: {i}")
            return i  # Return the index as soon as the element is found
    # This will only be executed if the loop completes without finding the element
    print(f"{userElement} was not found in the list.")
    return -1  # Return -1 to indicate the element was not found
```

Binary Search:

```
## Purpose: Search the user's list for an integer.
## Best Case: O(1), Worst Case: O(log n)
def binarySearch(userlist):
    userElement = int(input("Enter an integer you would like to find in the list: "))
    print()
    iElement = 0  # Start index
    jElement = len(userlist) - 1  # End index
    while iElement <= jElement:
        mElement = (iElement + jElement) // 2
        if userlist[mElement] == userElement:
            print(f"{userElement} was found at position: {mElement}")
            return mElement  # Return the index of the found element
        elif userlist[mElement] < userElement:
            iElement = mElement + 1  # Search in the right half
        else:
            jElement = mElement - 1  # Search in the left half
    # This will only be executed if the while loop exits without finding the element
    print(f"{userElement} was not found in the list.")
    return -1  # Return -1 to indicate the element was not found
```

Minimum/Maximum Function

```
##Purpose: Determines both the min and max values of a list
##Best Case: O(n)
def minMaxAlgorithm(userlist):
    ##Assigns currentmin and currentmax with the first element of the user list
    currentmin = currentmax = userlist[0]
    ##Assigned 'n' for the len of the user list
    n = len(userlist)
    for i in range(1, n):
        ## Compares the userlist[i] with currentmin
        if userlist[i] < currentmin:
            currentmin = userlist[i]
        ## Compares the userlist[i] with currentmax
        if userlist[i] > currentmax:
            currentmax = userlist[i]
    return currentmin, currentmax
```

Main Function

```
def main():
    while True:
        try:
            # Input list
            userinput = input("Enter a list of integers separated by spaces:\n")
            userlist = [int(x) for x in userinput.split()]
            # Sorting
            sortingMethod = int(input("Which sorting method would you like to use? \n"
                                      "1: Selection Sort\n"
                                      "2: Bubble Sort\n"
                                      "3: Insertion Sort\n"
                                      "4: Merge Sort\n"
                                      "5: Quick Sort\n"))
            if sortingMethod == 1:
                userlist = selectionSort(userlist)
            elif sortingMethod == 2:
                userlist = bubbleSort(userlist)
            elif sortingMethod == 3:
                userlist = insertionSort(userlist)
            elif sortingMethod == 4:
                userlist = mergeSort(userlist)
                print(f"The list after Merge Sort: {userlist}")
            elif sortingMethod == 5:
                userlist = quickSort(userlist)
                print(f"The list after Quick Sort: {userlist}")
            else:
                print("Invalid input. Please enter a number between 1 and 5.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter numbers for the list and a valid option for sorting.")
    # Searching
    while True:
        searchingAlgorithm = input("Would you like to search for an integer?\nYes or No\n").strip().lower()
        if searchingAlgorithm == 'yes':
            while True:
                Try:
```

```
            selectAlgorithm = int(input("Which algorithm would you like to use:\n1: Linear
            Search\n2:  Binary Search.\n"))
          if selectAlgorithm == 1:
            print(linearSearch(userlist))
```

Main Function(Continued…)

Continued from the previous page…

```
            break
          elif selectAlgorithm == 2:
            print(binarySearch(userlist))
            break
          else:
            print("Invalid input. Please enter '1' or '2'.")
        except ValueError:
          print("Invalid input. Please enter '1' or '2'.")
      break  # Exit search loop
    elif searchingAlgorithm == 'no':
      print("No value needed to find.")
      break
    else:
      print("Invalid input. Please enter 'yes' or 'no'.")
# Min/Max Calculation
while True:
    minmax = input("Would you like to calculate the min and max value?\nYes or No:\n").strip().lower()
    if minmax == 'yes':
      print("Minimum value and Maximum value:", minMaxAlgorithm(userlist))
      break
    elif minmax == 'no':
      break
    else:
      print("Invalid input. Please enter 'yes' or 'no'.")
# Program Loop
while True:
    tryAgain = input("Would you like to use the program again? (Yes or No)\n").strip().lower()
    if tryAgain == 'yes':
      main()
    else:
      print("Thank you for using my program!")
      break
```

Conclusion

- This document explored the implementation, functionality, and theoretical background of different sorting and searching algorithms. The following program is based on an integrated approach to understanding and interacting with various algorithms: Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort, which will be applied for sorting purposes. Simultaneously, Linear Search and Binary Search will be used for searching purposes.

- The program insists on users' activity, such as array input, choice of sort method, search, and extra options like computing minimum and maximum values. It also covers error handling to ensure sound and user-friendly operations.

- This document will provide a deeper understanding of the algorithm's efficiency and suitability for different contexts based on the theoretical foundations of algorithmic complexities, Big-O Notation, and performance analysis under Best, Average, and worst-case scenarios. This combination of practical implementation and theoretical insights seeks to create a comprehensive grasp of algorithmic concepts in users, preparing them for real-world problem-solving and optimization tasks.

Sources

Web articles:

Rowell, Eric. ***"Big-O Algorithm Complexity Cheat Sheet."*** *Big-O Cheat Sheet, www.bigocheatsheet.com.*

"Big O Notation with Searching & Sorting – A Level Computer Science." *LearnLearn.uk, www.learnlearn.uk/alevelcs/big-o-notation-searching-sorting.*

"Big O Cheat Sheet – Time Complexity Chart." *freeCodeCamp, 8 Oct. 2019, www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart.*

"Sorting Algorithms." *GeeksforGeeks, GeeksforGeeks, https://www.geeksforgeeks.org/sorting-algorithms/.*