**OPTIMIZED SELECTION SORT**

By: Logeswarar G(VIT CHENNAI)

Guided By: Dr. Helen Vijitha P(VIT CHENNAI)

**Abstract**

The custom sorting algorithm implemented in this study iteratively identifies the minimum and maximum elements within the unsorted portion of the array and positions them at the beginning and end, effectively reducing the unsorted portion of the array with each iteration. The analysis of this algorithm involves evaluating its performance across a diverse range of datasets, including random, sorted, and reverse-sorted arrays, as well as arrays with identical elements. By comparing its efficiency with established standard sorting algorithms such as quicksort, merge sort, and insertion sort, this study seeks to elucidate the algorithm's comparative advantages and limitations.

In addition to performance evaluation, the paper delves into the algorithm's stability, adaptability to different data distributions, and scalability to larger datasets. Utilizing empirical validation and computational complexity analysis, the research aims to provide comprehensive insights into the behavior of the custom sorting algorithm under various scenarios, shedding light on

its practical utility and potential applications in real-world contexts.

Moreover, through rigorous experimentation and benchmarking, the study endeavors to uncover the strengths and weaknesses of the custom sorting

algorithm, identifying the specific characteristics of datasets for which it excels, as well as those for which it may

exhibit suboptimal performance. Furthermore, the paper discusses potential refinement mechanisms and optimization

strategies for the custom algorithm, aiming to enhance its efficiency and broaden its applicability in diverse sorting scenarios.

Finally, the findings and conclusions drawn from the comparative analysis of the custom sorting algorithm and standard sorting techniques contribute to a deeper understanding of algorithmic design principles and provide valuable practical implications for sorting large datasets in various computational domains.

**Introduction**

Sorting algorithms are fundamental in computer science, with numerous applications in data processing, database management, and more. This paper introduces a custom sorting algorithm and evaluates its performance. The algorithm is designed to sort an array by repeatedly finding the minimum and maximum elements in the unsorted portion of the array and placing them at the beginning and end, respectively. The study evaluates the algorithm based on time complexity, space complexity, and practical performance on various datasets.  
  
Additionally, the paper explores the significance of sorting algorithms in computer science and various fields, highlighting their pivotal role in optimizing search operations, facilitating efficient information retrieval, and enabling the manipulation of structured data. By introducing the custom sorting algorithm and rigorously evaluating its performance, this study aims to contribute to the ongoing discourse surrounding sorting mechanisms and their impact on computational efficiency and data management.

Furthermore, the evaluation of the custom sorting algorithm encompasses an in-depth analysis of its time complexity and space complexity, delving into the theoretical underpinnings of the algorithm's computational efficiency and resource utilization. The investigation includes comparisons with established sorting algorithms to contextualize the algorithm's performance within the broader landscape of sorting approaches, thereby offering insights into its relative advantages and drawbacks.

Moreover, the practical performance evaluation of the custom sorting algorithm across diverse datasets provides valuable empirical evidence regarding its adaptability to different data distributions and characteristics. By examining its behavior on varying input sizes and data types, the study aims to elucidate the algorithm's practical utility and its suitability for real-world applications in data processing, scientific computing, and algorithmic problem-solving.

In addition to performance metrics, the paper delves into the algorithm's stability, handling of duplicate elements, and resilience to edge cases, enriching the understanding of its applicability in scenarios with specific data properties and constraints. This comprehensive assessment serves to inform practitioners and researchers about the algorithm's suitability for specific use cases and aids in guiding informed decision-making regarding algorithm selection for sorting tasks.

Moreover, the research endeavors to provide actionable insights into potential optimization strategies and refinements to enhance the algorithm's performance, scalability, and adaptability. By identifying areas for improvement and suggesting avenues for further exploration, the study aims to contribute to the refinement and evolution of sorting algorithms, fostering advancements in computational efficiency and data processing capabilities.

In conclusion, the comprehensive evaluation of the custom sorting algorithm across theoretical, empirical, and practical dimensions seeks to offer a holistic perspective on its capabilities and limitations, thereby enriching the body of knowledge pertaining to sorting algorithms and their applications in diverse computational domains.

**Methodology**

Custom Sorting Algorithm

The custom sorting algorithm is designed to sort an array by repeatedly finding the minimum and maximum elements in the unsorted portion of the array and placing them at the beginning and end, respectively. The algorithm is described in the following pseudocode:

function customSort(array, size):

start = 0

end = size - 1

while start < end:

max = array[start]

min = array[start]

max\_index = start

min\_index = start

for k from start to end:

if array[k] > max:

max = array[k]

max\_index = k

if array[k] < min:

min = array[k]  
  
  
  
  
 **Explanation**

Iteration Process: During each iteration of the algorithm, the following steps are executed sequentially:

Initialization: The algorithm begins by initializing two pointers, start and end, which respectively point to the start and end of the unsorted portion of the array.

Finding Min and Max: Within the unsorted portion delimited by the start and end pointers, the algorithm scans for the minimum and maximum elements. This involves iterating through the elements and updating the current minimum and maximum values along with their corresponding indices.

Swapping Elements: Upon determining the minimum and maximum elements, the algorithm performs element swaps to reposition them. The minimum element is swapped with the element at the start index, while the maximum element is swapped with the element at the end index, effectively placing them at the appropriate ends of the array segment.

Updating Pointers: Following the element swaps, the start pointer is advanced to the next index, and the end pointer is decremented, reducing the size of the unsorted portion of the array for subsequent iterations.

Repeat or Terminate: The algorithm continues iterating through the array segment as long as the start pointer remains less than the end pointer. Once the start and end pointers meet or cross each other, indicating that the entire array has been sorted, the iteration process concludes.

Complexity Analysis: The algorithm's time complexity arises primarily from the iterative nature of finding the minimum and maximum elements within the unsorted segment during each iteration. The space complexity is constant, as the algorithm operates in place without requiring additional data structures. By analyzing the theoretical time and space complexities, the algorithm's efficiency and suitability for varying dataset sizes and characteristics can be better understood.

Optimization Considerations: To enhance the algorithm's performance, considerations for optimizations such as early termination conditions, adaptive thresholding for switching to alternate sorting techniques, or parallelization for improved scalability can be explored. By refining the algorithm based on empirical observations and theoretical insights, its overall effectiveness and applicability in practical scenarios can be further enhanced.

By elaborating on the iteration process, complexities, and optimization avenues, a comprehensive understanding of the custom sorting algorithm's design, functionality, and potential enhancements can be achieved.  
  
  
  
  
#include <stdio.h>

void printArray(int l[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", l[i]);

}

printf("\n");

}

void customSort(int l[], int size) {

int st = 0;

int end = size - 1;

while (st < end) {

int max = l[st];

int min = l[st];

int max\_index = st;

int min\_index = st;

for (int k = st; k <= end; k++) {

if (l[k] > max) {

max = l[k];

max\_index = k;

}

if (l[k] < min) {

min = l[k];

min\_index = k;

}

}

int temp = l[st];

l[st] = l[min\_index];

l[min\_index] = temp;

if (max\_index == st) {

max\_index = min\_index;

}

temp = l[end];

l[end] = l[max\_index];

l[max\_index] = temp;

st += 1;

end -= 1;

}

}

int main() {

int testCases[10][5] = {

{3, 7, 1, 9, 5},

{8, 4, 6, 2, 10},

{5, 5, 5, 5, 5},

{10, 20, 30, 40, 50},

{99, 88, 77, 66, 55},

{1, 2, 3, 4, 5},

{9, 3, 6, 2, 7},

{-5, -3, -7, -2, -6},

{11, 22, 11, 22, 11},

{0, 0, 0, 0, 0}

};

for (int i = 0; i < 10; i++) {

int l[5] = {testCases[i][0], testCases[i][1], testCases[i][2], testCases[i][3], testCases[i][4]};

customSort(l, 5);

printArray(l, 5);

}

return 0;

}

Result:

1 3 5 7 9

2 4 6 8 10

5 5 5 5 5

10 20 30 40 50

55 66 77 88 99

1 2 3 4 5

2 3 6 7 9

-7 -6 -5 -3 -2

11 11 11 22 22

0 0 0 0 0

**Applications**:

One potential application of this code could be in the context of embedded systems or resource-constrained environments, where the simplicity and low memory requirements of the this sort algorithm can be advantageous.

Sensor Data Processing: Imagine a scenario where you have a sensor that collects a small set of measurements (e.g., temperature, humidity, or pressure) at regular intervals. The selection sort algorithm implemented in this code could be used to efficiently sort and analyze the sensor data, which is typically small in size and may have a limited range of values.

Inventory Management: In a small retail or warehouse setting, the inventory data may consist of a limited number of items, each with a specific quantity. The selection sort algorithm could be used to sort the inventory data by item quantity, making it easier to identify low-stock items or items that need to be reordered.

Sorting Student Grades: In an educational setting, the selection sort algorithm could be used to sort student grades, which typically range from 0 to 100. This could be useful for tasks such as generating grade reports or identifying the top-performing students in a class.

Sorting Demographic Data: In a small-scale demographic study, the selection sort algorithm could be used to sort data such as age, income, or household size, which may have a limited range of values.

Sorting Preferences or Ratings: Imagine a scenario where users are asked to rate or rank a set of items (e.g., products, services, or entertainment options) on a scale. The selection sort algorithm could be used to efficiently sort the user preferences or ratings, which are typically small in size and may have a limited range of values.  
  
  
**Time complexities**

Best Case: In the best-case scenario for Selection Sort, the input array is already sorted or nearly sorted. However, the nature of the Selection Sort algorithm involves scanning through the list to find the minimum element and then swapping it into place. As a result, even in the best-case scenario, Selection Sort still requires Θ(n^2) comparisons and swaps, where n is the number of elements in the array.

Worst Case: In the worst-case scenario for Selection Sort, the input array is in reverse sorted order. In this case, Selection Sort will require the maximum number of comparisons and swaps. The time complexity of Selection Sort in the worst case is O(n^2), where n is the number of elements in the array. This is because for each element, Selection Sort compares it with all other elements in the unsorted portion of the array.

Average Case: In the average case for Selection Sort, the time complexity is also O(n^2). This holds true even if the input array is randomly shuffled. On average, Selection Sort performs the same number of comparisons and swaps as in the worst case. The average-case time complexity is O(n^2) because, for each element, Selection Sort compares it with all other elements in the unsorted portion of the array, resulting in n(n-1)/2 comparisons.

**Conclusion:**

Time Complexity: Both the customSort function and the standard Selection Sort algorithm have the same time complexity of O(n^2) in the average and worst cases. The additional steps in the customSort function do not change the overall asymptotic time complexity.

Optimization Potential: The customSort function may have some minor optimization potential compared to the standard Selection Sort, such as the step of updating the max\_index when the maximum element's index is the same as the start index. However, the impact of these optimizations is likely to be small.

Readability and Maintainability: The customSort function may be slightly more concise and easier to read compared to the standard Selection Sort implementation, as it combines the steps of finding the maximum and minimum elements in a single loop. This can make the code more readable and maintainable, depending on the preferences of the development team.

Flexibility: The customSort function is more flexible than the previous implementation, as it allows the caller to specify the size of the input array, whereas the previous implementation assumed a fixed size of 5.

Stability: The standard Selection Sort algorithm is not a stable sorting algorithm, while the customSort function does not appear to have any specific provisions for maintaining the relative order of equal elements, so its stability is unclear.

