Sorting Algorithms

Exercise 1: Exploration of Bubble Sort

Objective: Explore and implement the Bubble Sort algorithm, analyze its performance, and enhance its efficiency.

Task 1: Implement the conventional Bubble Sort algorithm.

Task 2: Enhance the Bubble Sort algorithm. Propose and implement an optimization strategy.

Task 3: Analyze the time complexity for both the conventional and enhanced versions of Bubble Sort.

Additional Inquiry: Investigate the impact of diverse input datasets on the performance of Bubble Sort.

Exercise 2: Delving into Insertion Sort

Objective: Implement and optimize the Insertion Sort algorithm and evaluate its computational complexity.

Task 1: Implement the traditional Insertion Sort algorithm.

Task 2: Optimize your Insertion Sort implementation.

Task 3: Examine and report the time complexity for both the standard and optimized versions of Insertion Sort.

Additional Inquiry: Explore the scenarios where Insertion Sort might outperform more complex sorting algorithms.

Exercise 3: Analyzing Selection Sort

Objective: Gain a comprehensive understanding of the Selection Sort algorithm.

Task 1: Implement the fundamental Selection Sort algorithm.

Task 2: Advance the Selection Sort algorithm.

Task 3: Conduct a time complexity analysis for both the basic and advanced versions of Selection Sort.

Additional Inquiry: Investigate the behavior of Selection Sort on nearly sorted lists.

Exercise 4: Investigating Merge Sort (Fusion Sort)

Objective: Dissect and enhance the Merge Sort algorithm, and examine its computational efficiency.

Task 1: Implement the classic Merge Sort algorithm.

Task 2: Enhance your Merge Sort implementation.

Task 3: Analyze the time complexity for both the standard and improved versions of Merge Sort.

Additional Inquiry: Evaluate the performance of Merge Sort with large datasets.

Exercise 5: Quick Sort In-Depth

Objective: Delve into the Quick Sort algorithm, refine its implementation, and scrutinize its time complexity.

Task 1: Implement the basic Quick Sort algorithm.

Task 2: Refine the Quick Sort algorithm.

Task 3: Evaluate the time complexity for both the basic and enhanced Quick Sort implementations.

Additional Inquiry: Examine how the choice of pivot influences Quick Sort's efficiency.

Solution to Exercise 1: Bubble Sort

Basic Bubble Sort Implementation:

```c
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Listing 1: C code for analysis

Enhanced Bubble Sort Implementation:

```c
void enhancedBubbleSort(int arr[], int n) {
    int i, j;
    int swapped;
    for (i = 0; i < n-1; i++) {
        swapped = 0;
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = 1;
            }
        }
        if (swapped == 0)
            break;
    }
}
```

Listing 2: C code for analysis

Time Complexity Analysis:

- Basic Bubble Sort: $O(n^2)$ in average and worst case.
  - This is because, in the worst case, Bubble Sort compares each pair of adjacent items in the list for every item in the list. This results $O(n^2)$complexity.
- Enhanced Bubble Sort: $O(n^2)$ in the worst case but $O(n)$ in the best case (when the list is already sorted).
  - If the list is already sorted (which is the best case), the enhanced version of Bubble Sort can detect this using a flag to indicate whether any swaps were made. If no swaps occur in a pass, the list is sorted, and the algorithm stops. This results in a best-case complexity of $O(n)$.

Additional Question: Investigate the impact of diverse input datasets on the performance of Bubble Sort. How does Bubble Sort perform with datasets that are already sorted, reverse sorted, and containing many duplicate elements?

Answer: Bubble Sort performs optimally on datasets that are already sorted, having a time complexity of $O(n)$ due to the early termination in the enhanced version. In the case of reverse sorted datasets, it performs the worst, taking $O(n^2)$ time. With many duplicate elements, its performance depends on the distribution of these elements but generally stays closer to $O(n^2)$.

Solution to Exercise 2: Insertion Sort

Basic Insertion Sort Implementation:

```c
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Listing 3: C code for analysis

Enhanced Insertion Sort Implementation (Binary Insertion Sort):

```c
int binarySearch(int arr[], int item, int low, int high) {
    if (high <= low)
        return (item > arr[low]) ? (low + 1) : low;
    int mid = (low + high) / 2;
    if (item == arr[mid])
        return mid + 1;
    if (item > arr[mid])
```

```
        return binarySearch(arr, item, mid + 1, high);
    return binarySearch(arr, item, low, mid - 1);
}

void binaryInsertionSort(int arr[], int n) {
    int i, loc, j, k, selected;
    for (i = 1; i < n; ++i) {
        j = i - 1;
        selected = arr[i];
        loc = binarySearch(arr, selected, 0, j);
        while (j >= loc) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = selected;
    }
}
```

Listing 4: C code for analysis

Time Complexity Analysis:

- Basic Insertion Sort: $O(n^2)$ in average and worst case.
  - In the worst case, each new element has to be compared with all the other elements already sorted, resulting in $O(n^2)$.
- Enhanced Insertion Sort: $O(n \log n)$ for comparisons but still $O(n^2)$ for swaps.
  - The binary search reduces the number of comparisons to $O(n \log n)$, but the movement of elements to create space remains $O(n)$ for each insertion and $O(n^2)$ overall. Therefore, the overall complexity is dominated by the movement of elements.

Additional Question: Explore the scenarios where Insertion Sort might outperform more complex sorting algorithms. What are the characteristics of such datasets?

Answer: Insertion Sort often outperforms more complex algorithms on small datasets, nearly sorted datasets, or datasets where elements are only a few positions away from their sorted position. Its simplicity and minimal overhead make it efficient for such cases.

Solution to Exercise 3: Selection Sort

Basic Selection Sort Implementation:

```
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

```c
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Listing 5: C code for analysis

Enhanced Selection Sort Implementation:

```c
#include <stdio.h>

void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void enhancedSelectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }

        // Swap the found minimum element with the first element,
        // only if the minimum element is not already in the
    correct position
        if (min_idx != i) {
            swap(&arr[min_idx], &arr[i]);
        }
    }
}
```

Listing 6: C code for analysis

Time Complexity Analysis:

- Basic and Enhanced Selection Sort: $O(n^2)$ in both average and worst case.
  - This algorithm sorts an array by repeatedly finding the minimum element and moving it to the beginning. It performs $n(n-1)/2$ comparisons, leading to a time complexity of $O(n^2)$.

Additional Question: Discuss why Selection Sort might not be the best choice for large datasets. What specific properties of Selection Sort lead to this?

Answer: Selection Sort has a time complexity of $O(n^2)$ regardless of the dataset's initial order, making it inefficient for large datasets. Its lack of adaptability and the need to perform a significant number of comparisons and swaps contribute to its unsuitability for large-scale sorting tasks.

Solution to Exercise 4: Merge Sort

Basic Merge Sort Implementation:

```c
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Listing 7: C code for analysis

Enhanced Merge Sort Implementation:

```c
#include <stdio.h>

#define SIZE 100
#define THRESHOLD 10
void insertionSort(int arr[], int left, int right) {
    int i, key, j;
    for (i = left + 1; i <= right; i++) {
        key = arr[i];
        j = i - 1;
```

```
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[SIZE], R[SIZE];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void enhancedMergeSort(int arr[], int l, int r) {
    if (l < r) {
        if (r - l <= THRESHOLD) {
            insertionSort(arr, l, r);
        } else {
            int m = l + (r - l) / 2;
            enhancedMergeSort(arr, l, m);
            enhancedMergeSort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }
}
```

Listing 8: C code for analysis

Time Complexity Analysis:

- Merge Sort: $O(n \log n)$ in both average and worst case.
  - Merge Sort is a divide and conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the sorted halves. Each division and merge operation is $O(n)$, and the division happens $\log n$ times. Therefore, the overall time complexity is $O(n \log n)$.

Additional Question: Explain why Merge Sort is considered a stable sorting algorithm. What feature of Merge Sort ensures its stability?

Answer: Merge Sort is considered stable because it maintains the relative order of equal elements. During the merge phase, when two elements are equal, it prefers the element from the left subarray, thus preserving their original order.

Solution to Exercise 5: Quick Sort

Basic Quick Sort Implementation:

```c
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Listing 9: C code for analysis

Enhanced Quick Sort Implementation:

```c
int partitionRandom(int arr[], int low, int high) {
    srand(time(NULL));
    int random = low + rand() % (high - low);
    int temp = arr[random];
    arr[random] = arr[high];
    arr[high] = temp;
    return partition(arr, low, high);
```

```
}

void enhancedQuickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partitionRandom(arr, low, high);
        enhancedQuickSort(arr, low, pi - 1);
        enhancedQuickSort(arr, pi + 1, high);
    }
}
```

Listing 10: C code for analysis

Time Complexity Analysis:

- Basic Quick Sort: $O(n^2)$ in the worst case, but $O(n \log n)$ on average.
  - The worst case occurs when the pivot results in highly unbalanced partitions. However, on average, the partitions are reasonably balanced, leading to $O(n \log n)$.
- Enhanced Quick Sort: $O(n \log n)$ on average and worst case, due to random pivot selection.
  - By choosing a random pivot, the algorithm minimizes the chances of consistently poor divisions, thus ensuring a more balanced division and improving the average and worst-case performance to $O(n \log n)$.

Additional Question: Discuss the impact of pivot selection on the performance of Quick Sort. How does choosing a random pivot improve the algorithm?

Answer: Pivot selection is crucial in Quick Sort as it affects the division of the array. A bad pivot choice (e.g., always choosing the last element) can lead to poor performance, especially on already sorted arrays. Choosing a random pivot minimizes the chances of consistently poor divisions, thus ensuring a more balanced division and improving the average and worst-case performance.