6. Efficient Sorting Algorithms

6.1 Merge Sort

Concept

Merge Sort is a divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges the sorted halves.

Divide and Conquer Strategy:

1. **Divide:** Split array into two halves
2. **Conquer:** Recursively sort each half
3. **Combine:** Merge the two sorted halves

Visual Example:

```
Initial Array: [38, 27, 43, 3, 9, 82, 10]


DIVIDE Phase:
                [38, 27, 43, 3, 9, 82, 10]
                    /               \
        [38, 27, 43, 3]          [9, 82, 10]
           /        \              /       \
      [38, 27]    [43, 3]      [9, 82]   [10]
       /    \      /    \       /    \      |
     [38]  [27]  [43]  [3]    [9]  [82]  [10]

MERGE Phase:
     [38]  [27]  [43]  [3]    [9]  [82]  [10]
       \    /      \    /       \    /      |
      [27, 38]    [3, 43]      [9, 82]   [10]
           \        /              \       /
        [3, 27, 38, 43]          [9, 10, 82]
                    \               /
                [3, 9, 10, 27, 38, 43, 82]
```

Implementation

Merge Sort Algorithm:

```
#include <stdio.h>
#include <stdlib.h>

// Merge two sorted subarrays
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;  // Size of left subarray
    int n2 = right - mid;      // Size of right subarray

    // Create temporary arrays
    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    // Merge the temporary arrays back
    int i = 0;      // Initial index of left subarray
    int j = 0;      // Initial index of right subarray
    int k = left;   // Initial index of merged array

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

    // Copy remaining elements of L[]
    while (i < n1) {
```

```
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[]
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

// Main merge sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Sort first half
        mergeSort(arr, left, mid);

        // Sort second half
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
```

```
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Merge Sort with Visualization:

```
void mergeSortVisualize(int arr[], int left, int right, int depth) {
    if (left < right) {
        // Print indentation based on recursion depth
        for (int i = 0; i < depth; i++) printf("  ");

        printf("Divide: [");
        for (int i = left; i <= right; i++) {
            printf("%d", arr[i]);
            if (i < right) printf(", ");
        }
        printf("]\n");

        int mid = left + (right - left) / 2;

        mergeSortVisualize(arr, left, mid, depth + 1);
        mergeSortVisualize(arr, mid + 1, right, depth + 1);

        merge(arr, left, mid, right);

        // Print result after merge
        for (int i = 0; i < depth; i++) printf("  ");
        printf("Merge:  [");
        for (int i = left; i <= right; i++) {
            printf("%d", arr[i]);
```

```
            if (i < right) printf(", ");
        }
        printf("]\n");
    }
}
```

Iterative Merge Sort:

```
void mergeSortIterative(int arr[], int n) {
    // Start with merge subarrays of size 1, then 2, 4, 8, ...
    for (int currSize = 1; currSize < n; currSize *= 2) {
        // Pick starting index of left sub array
        for (int leftStart = 0; leftStart < n - 1; leftStart += 2 * currSize) {
            // Find ending point of left subarray
            int mid = leftStart + currSize - 1;

            // Find ending point of right subarray
            int rightEnd = (leftStart + 2 * currSize - 1 < n - 1) ?
                           leftStart + 2 * currSize - 1 : n - 1;

            // Merge subarrays
            if (mid < rightEnd) {
                merge(arr, leftStart, mid, rightEnd);
            }
        }
    }
}
```

Complexity Analysis

Time Complexity:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n)
- **Always O(n log n)!**

Space Complexity: O(n) - requires temporary arrays

Why O(n log n)?

```
Tree height: log₂(n) levels
Work at each level: n comparisons/copies

Total work = height × work per level
           = log₂(n) × n
           = O(n log n)

Example with n=8:
Level 0: [8 elements] → n operations
Level 1: [4,4] → n operations (4+4)
Level 2: [2,2,2,2] → n operations (2+2+2+2)
Level 3: [1,1,1,1,1,1,1,1] → n operations

Total levels = log₂(8) = 3
Total operations = 3n = O(n log n)
```

Comparison with Simple Sorts:

```
n=1,000:
- Insertion Sort: ~500,000 operations
- Merge Sort: ~10,000 operations
→ 50x faster!

n=1,000,000:
- Insertion Sort: ~500 billion operations
- Merge Sort: ~20 million operations
→ 25,000x faster!
```

Advantages and Disadvantages

Advantages:

1. **Guaranteed O(n log n)** - always efficient
2. **Stable** - preserves relative order
3. **Predictable** - no worst-case scenarios
4. **Good for linked lists** - O(1) space possible
5. **Parallelizable** - can sort halves independently

Disadvantages:

1. **O(n) space** - requires temporary storage
2. **Not in-place** - not memory efficient

3. **Overhead** - slower than Quick Sort in practice
4. **Not adaptive** - doesn't benefit from sorted data

When to Use

Use Merge Sort when:

- Guaranteed O(n log n) required
- Stability is important
- Working with linked lists
- External sorting (large files)
- Parallel processing available

Don't use when:

- Memory is limited
- In-place sorting required
- Small datasets (overhead not worth it)

6.2 Quick Sort

Concept

Quick Sort is a divide-and-conquer algorithm that picks a pivot element and partitions the array around it, then recursively sorts the subarrays.

How it works:

1. **Choose Pivot:** Select an element as pivot
2. **Partition:** Rearrange so elements < pivot are left, elements > pivot are right
3. **Recursively sort:** Sort left and right subarrays

Visual Example:

```
Initial: [10, 80, 30, 90, 40, 50, 70]
         Pick pivot = 70 (last element)

Partition:
[10, 30, 40, 50] 70 [80, 90]
 ← less than 70     greater than 70 →

Recursively sort left: [10, 30, 40, 50]
Pick pivot = 50
[10, 30, 40] 50 []
```

```
Recursively sort right: [80, 90]
Pick pivot = 90
[80] 90 []

Final: [10, 30, 40, 50, 70, 80, 90]
```

Partitioning Process:

```
Array: [10, 80, 30, 90, 40, 50, 70]
Pivot: 70 (last element)

i = -1 (tracks position of smaller elements)
j = 0 (scans through array)

j=0: arr[0]=10 < 70 → swap arr[++i] with arr[j]
     [10, 80, 30, 90, 40, 50, 70]
      i

j=1: arr[1]=80 > 70 → no swap
     [10, 80, 30, 90, 40, 50, 70]
      i

j=2: arr[2]=30 < 70 → swap arr[++i] with arr[j]
     [10, 30, 80, 90, 40, 50, 70]
          i

j=3: arr[3]=90 > 70 → no swap

j=4: arr[4]=40 < 70 → swap arr[++i] with arr[j]
     [10, 30, 40, 90, 80, 50, 70]
               i

j=5: arr[5]=50 < 70 → swap arr[++i] with arr[j]
     [10, 30, 40, 50, 80, 90, 70]
                  i

Finally: Place pivot at i+1
     [10, 30, 40, 50, 70, 90, 80]
                       ↑
```

```
                    pivot in place
```

Implementation

Quick Sort with Last Element as Pivot:

```
#include <stdio.h>

// Swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choose last element as pivot
    int i = low - 1;        // Index of smaller element

    for (int j = low; j < high; j++) {
        // If current element is smaller than pivot
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Place pivot in correct position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// Quick sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
```

```
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {10, 80, 30, 90, 40, 50, 70};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Quick Sort with Middle Element as Pivot:

```
int partitionMiddle(int arr[], int low, int high) {
    int mid = low + (high - low) / 2;
    int pivot = arr[mid];

    // Move pivot to end
    swap(&arr[mid], &arr[high]);

    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
```

```
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
```

Quick Sort with Random Pivot:

```
#include <stdlib.h>
#include <time.h>

int partitionRandom(int arr[], int low, int high) {
    // Generate random pivot index
    srand(time(NULL));
    int randomIndex = low + rand() % (high - low + 1);

    // Move pivot to end
    swap(&arr[randomIndex], &arr[high]);

    return partition(arr, low, high);
}

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

Quick Sort with Visualization:

```
void quickSortVisualize(int arr[], int low, int high, int depth) {
    if (low < high) {
        // Print indentation
        for (int i = 0; i < depth; i++) printf("  ");

        printf("Sorting: [");
```

```
        for (int i = low; i <= high; i++) {
            printf("%d", arr[i]);
            if (i < high) printf(", ");
        }
        printf("] Pivot=%d\n", arr[high]);

        int pi = partition(arr, low, high);

        // Print result
        for (int i = 0; i < depth; i++) printf("  ");
        printf("Result:  [");
        for (int i = low; i <= high; i++) {
            if (i == pi) printf("*");
            printf("%d", arr[i]);
            if (i == pi) printf("*");
            if (i < high) printf(", ");
        }
        printf("]\n");

        quickSortVisualize(arr, low, pi - 1, depth + 1);
        quickSortVisualize(arr, pi + 1, high, depth + 1);
    }
}
```

Three-Way Partitioning (for duplicates):

```
void quickSort3Way(int arr[], int low, int high) {
    if (low >= high) return;

    int pivot = arr[high];
    int i = low;
    int lt = low;      // Elements < pivot
    int gt = high;     // Elements > pivot

    while (i <= gt) {
        if (arr[i] < pivot) {
            swap(&arr[lt++], &arr[i++]);
        }
        else if (arr[i] > pivot) {
            swap(&arr[i], &arr[gt--]);
```

```
        }
        else {
            i++;
        }
    }

    quickSort3Way(arr, low, lt - 1);
    quickSort3Way(arr, gt + 1, high);
}
```

Complexity Analysis

Time Complexity:

- Best Case: O(n log n) - balanced partitions
- Average Case: O(n log n)
- Worst Case: O(n²) - unbalanced partitions

Space Complexity: O(log n) - recursion stack

Best Case (Balanced Partition):

```
Each partition divides array in half

          [8 elements]
         /            \
    [4]                [4]
   /   \              /   \
  [2]  [2]          [2]  [2]
 / \   / \         / \   / \
[1][1][1][1]      [1][1][1][1]

Height = log₂(n)
Work per level = n
Total = O(n log n)
```

Worst Case (Unbalanced Partition):

```
Sorted or reverse sorted with bad pivot choice

[5, 4, 3, 2, 1] pivot = 1
    ↓
```

```
[1] [5, 4, 3, 2] pivot = 2
        ↓
    [2] [5, 4, 3] pivot = 3
            ↓
        [3] [5, 4]
                ↓
            [4] [5]

Height = n
Work = n + (n-1) + (n-2) + ... + 1
     = n(n+1)/2
     = O(n²)
```

Avoiding Worst Case:

1. **Random Pivot:** Makes worst case unlikely
2. **Median-of-Three:** Use median of first, middle, last
3. **Three-Way Partition:** Handle duplicates efficiently

Advantages and Disadvantages

Advantages:

1. **Fast in practice** - usually faster than Merge Sort
2. **In-place** - O(log n) space only
3. **Cache-friendly** - good locality of reference
4. **Parallelizable** - can sort partitions independently

Disadvantages:

1. **Unstable** - doesn't preserve relative order
2. **O(n²) worst case** - rare but possible
3. **Not adaptive** - doesn't benefit from sorted data
4. **Recursive** - stack overflow for deep recursion

Pivot Selection Strategies

1. Last Element (Simple):

```
int pivot = arr[high];
```

- Simple but vulnerable to sorted input

2. First Element:

```
int pivot = arr[low];
```

- Same issue as last element

3. Middle Element:

```
int mid = low + (high - low) / 2;
int pivot = arr[mid];
```

- Better for sorted input

4. Random Element:

```
int randomIndex = low + rand() % (high - low + 1);
int pivot = arr[randomIndex];
```

- Probabilistically avoids worst case

5. Median-of-Three:

```
int medianOfThree(int arr[], int low, int high) {
    int mid = low + (high - low) / 2;

    if (arr[low] > arr[mid])
        swap(&arr[low], &arr[mid]);
    if (arr[low] > arr[high])
        swap(&arr[low], &arr[high]);
    if (arr[mid] > arr[high])
        swap(&arr[mid], &arr[high]);

    return mid;
}
```

- Best practical choice

When to Use

Use Quick Sort when:

- Average case performance matters
- Memory is limited (in-place)
- Cache performance important
- Large datasets

- Stability not required

Don't use when:

- Worst case must be avoided
- Stability required
- Small datasets
- Stack space is limited

Comparison with Merge Sort:

	Quick Sort	Merge Sort
Time (avg):	O(n log n)	O(n log n)
Time (worst):	$O(n^2)$	O(n log n)
Space:	O(log n)	O(n)
Stable:	No	Yes
In-place:	Yes	No
Cache:	Better	Worse
Practice:	Faster	Slower

Revision #1
Created 2025-11-03 02:06:10 UTC by DS
Updated 2025-11-03 02:06:56 UTC by DS