Learning Objectives

Learners will be able to...

- **Identify the underlying mechanics of each of the merge and quick sort algorithms**
- **Implement each sorting method in Java**
- **Analyze the time and space complexities of each algorithm**
- **Identify practical scenarios best suited for each algorithm**

Advanced Sorting Algorithms and Analysis

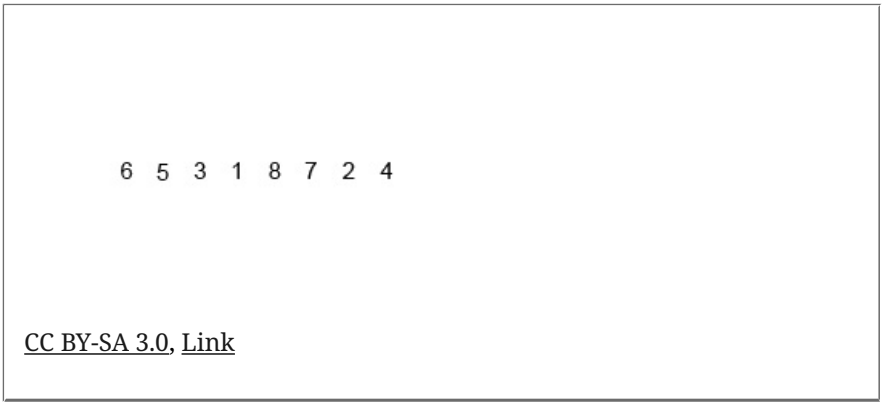
While basic sorting algorithms like bubble sort, insertion sort, and selection sort are simple and intuitive, they are not optimized for large datasets. As we venture into larger and more complex data structures, the need for advanced sorting algorithms becomes paramount. In this assignment, we will delve into two fundamental advanced sorting algorithms:

- **Merge Sort**: A classic divide-and-conquer algorithm that splits the array into halves, recursively sorts the halves, and then merges them in a sorted manner.
- **Quick Sort**: Another divide-and-conquer technique but with a twist – it selects a “pivot” element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Merge Sort

Merge Sort: Overview, Steps, and Java Implementation

Merge sorts are a recursive, divide-and-conquer sorting algorithm known for its consistent performance and stable sorting characteristics. The core idea is to continually divide an array into two halves, sort each half, and then merge the sorted halves together.



Steps

1. **Divide**: If the array has one or zero elements, it is considered sorted. Otherwise, divide the array into two halves.
2. **Conquer**: Recursively sort both halves of the array.
3. **Merge**: Combine (merge) the sorted halves to produce a single sorted array. This step is crucial and is where the two halves are merged in sorted order.

Java Implementation

Create the method `mergeSort` that takes an array of integers, an integer that represents the left index, and an integer that represents the right index. The method should not return anything. This is a recursive algorithm, so we need to set up the base case. As long as `left` is less than `right`, calculate the midpoint, sort the left half, sort the right half, and then merge the two halves together.

```
    public static void mergeSort(int[] arr, int left, int right)
        {
        if (left < right) {
            int mid = (left + right) / 2;

            // Sort the halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

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

Next, create the `merge` method. It takes an array of integers and three integers representing the left index, the midpoint, and the right index. Calculate the lengths of the left array (`n1`) and the right array (`n2`). Then create temporary arrays for the left half and the right half.

```
    private static void merge(int[] arr, int left, int mid, int
        right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

    }
```

Copy data from the left half of the original array (`arr`) into the temporary left array (`L`). Do the same thing for the right half.

```
    private static void merge(int[] arr, int left, int mid, int
        right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data
        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];

    }
```

Now that the original array has been divided into to temporary arrays, we can merge the elements back into the original array in order. Create the index variables `i`, `j`, and `k`. They will point to the position in the left array

(`i`), the right array (`j`), and the original array (`k`). As long as `i` and `j` have both not reached the end of their arrays, ask if the element in the temporary left array is less than the element in the temporary right array. If so, put the element from the temporary left array into the original array. Then increment `i`. Otherwise, put the element from the temporary right array into the original array and increment `j`. Before the loop iterates again, increment `k`.

```
    private static void merge(int[] arr, int left, int mid, int
        right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data
        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
        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

    }
```

There may be some remaining data in the left and right arrays that have not yet been merged. This could happen if either `i` or `j` reach the end of their respective array before the other has. Copy over any outstanding data from the temporary arrays back to the original array.

```
    private static void merge(int[] arr, int left, int mid, int
        right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data
        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
        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy any remaining elements
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

▼ Code

Your code should look like this:

```java
public static void mergeSort(int[] arr, int left, int right)
    {
    if (left < right) {
        int mid = (left + right) / 2;

        // Sort the halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

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

private static void merge(int[] arr, int left, int mid, int
    right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int L[] = new int[n1];
    int R[] = new int[n2];

    // Copy data
    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
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

In the `main` method, your code will be provided with the unsorted array:

```
[12, 11, 13, 5, 6, 7]
```

Your code should produce:

```
Sorted Array:
5 6 7 11 12 13
```

Merge sorts stand out due to their predictable and consistent performance. This makes merge sorts a common choice in many applications.$O(nlogn)$ time complexity in all cases (best, average, and worst). Though it has a space overhead because of the temporary arrays used in the merge step, its stable and consistent performance makes it a reliable choice in many applications.

Merge Sort Analysis

Merge Sort: Detailed Analysis

Merge sorts are popular due to their predictable performance. That does not mean, however, that this is the best sorting algorithm to use. There are a few cases where the merge sort may not be the optimal solution. Let's take a look at the sort in detail.

Time Complexity

- **Best Case**: The best-case scenario for a merge sort is when the array is already sorted or nearly sorted. However, due to the recursive nature of the algorithm and the way it divides and merges the array, the time complexity remains $O(nlogn)$.
- **Average Case**: On average, irrespective of the initial order of elements, merge sorts take $O(nlogn)$ time to sort an array.
- **Worst Case**: Even in the worst scenario, the time complexity of a merge sort is $O(nlogn)$, making it one of the more consistent sorting algorithms in terms of performance.

Space Complexity

Merge sorts use additional space for the temporary left (`L[]`) and right (`R[]`) arrays during the merge step. Thus, its space complexity is $O(n)$, making it less space-efficient compared to some in-place sorting algorithms.

Stability

A merge sort is a **stable** sorting algorithm. This means that the relative order of equal elements remains unchanged after sorting. Stability is essential in scenarios where multiple sorting passes are needed, like when sorting by multiple criteria.

Advantages

A merge sort guarantees $O(nlogn)$ performance irrespective of the initial order of the data. Many developers desire the consistency this sort offers. Merge sorts are also stable which is advantageous, especially in database management and other areas where data integrity and order are crucial. They are also well-suited for external sorting, where data to be sorted does

not fit into memory and resides in slower external memory (like disk). It efficiently manages chunks of data, making it a preferred choice in such scenarios.

Disadvantages

The major drawback is its space complexity of $O(n)$. This additional space requirement can be a concern for large arrays. However, that does not mean merge sorts are well-suited for small datasets either. For smaller lists, other algorithms, like insertion sort, might be faster due to the overhead of recursive function calls and memory allocations in a merge sort.

Merge sorts offer a blend of stable and consistent performance, making it a standard choice for many sorting tasks. While its predictable time complexity is a strength, potential users should be wary of its additional space requirements, especially in memory-constrained environments. When choosing a sorting algorithm, it's vital to consider the dataset's size, the available memory, and the importance of stability in sorting.

Quick Sort

Quick Sort: Overview, Steps, and Java Implementation

Quick sort is a divide-and-conquer sorting algorithm that operates by selecting a “pivot” element from the array and partitioning the other elements into two sub-arrays, based on whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Info

Sorting Mechanics

The following gif demonstrates how a quick sort works:

Steps

1. **Select Pivot**: Choose a pivot element from the array. The choice of pivot can be the first element, the last element, the middle one, or any random element.
2. **Partition**: Reorder the array by comparing with the pivot. All elements smaller than the pivot come before the pivot and all greater elements come after. After this step, the pivot is in its sorted position.
3. **Recursive Sort**: Recursively apply the above steps to the two sub-arrays (elements less than and greater than the pivot).

Java Implementation

Create the recursive method `quickSort`. It takes an array of integers, as well as two integers representing the low and high indices. The method should not return anything. As long as `low` is less than `high`, find the partition

index by calling the `partition` method. Then call `quickSort` on the left half and then again on the right half.

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

Next, we need to define the `partition` method. It takes an array of integers as well as two integers representing the low and high indices. The method should return an integer. Set the pivot to the last element in the array. Create the variable `i` and set its value to `low - 1`. Create a for loop that has the loop variable `j` that starts at `low` and iterates as long as `j` is less than `high`.

```
    private static int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {

        }
    }
```

Before we continue with the `partition` method, we are going to create the `swap` method that swaps to values in an array. We will be swapping a few different times, so it makes sense to put this logic into its own method. The method takes an array of integers as well as two other integers (the values to be swapped). It does not return anything.

```
    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

Return to the `partition` method. Inside the for loop, ask if the element at index `j` is less than the pivot. If so, increment `i` and swap the values `arr[i]` and `arr[j]` with the newly created `swap` method.

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

    }
```

After the for loop finishes running, swap the values at `arr[i + 1]` and `arr[high]`. Then return `i + 1`.

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

▼ Code

Your code should look like this:

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

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

    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

In the `main` method, your code will be provided with the unsorted array:

```
[10, 7, 8, 9, 1, 5]
```

Your code should produce:

```
Sorted array:
1 5 7 8 9 10
```

As its name implies, the quick sort is a faster sort. In practice, it can even outperform other sorting algorithms with the same time complexity. Developers have developed common strategies like using a random pivot or the “median-of-three” approach to help compensate for some of the algorithm’s shortcomings.

The Pivot

The Importance of the Pivot

The choice of the pivot element is crucial as it directly influences the efficiency and performance of the sorting process. The pivot element is used to partition the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot. The partitioning step is a fundamental operation in the quick sort algorithm, and the choice of pivot significantly affects the algorithm's behavior. Choosing a bad pivot leads to imbalanced partitioning. For example, if you consistently choose a pivot that is either the smallest or largest element in the array, the partitioning process will result in highly imbalanced subarrays. This means one subarray will have significantly more elements than the other, which leads to inefficient sorting. Here are some common techniques used to select a pivot for a quick sort.

Random Pivot Selection

In this strategy, we randomly select a pivot element from the array. This helps reduce the chances of consistently encountering a "bad" pivot. Start by importing the `Random` library. This should come at the beginning of the program.

```
import java.util.Random;
```

Then update the `QuickSort` class so that it instantiates a `Random` object. This should happen before the `quickSort` method.

```
public class QuickSort {
    private static final Random rand = new Random();

    // code below remains unchanged
```

Next, update the `partition` method. Generate a random number between the indices `low` and `high`. Then swap the pivot value to the end of the subarray. It will be swapped back to its correct position before returning `arr[i + 1]`.

```
    private static int partition(int arr[], int low, int high) {
        int pivotIndex = rand.nextInt(high - low + 1) + low;
        int pivotValue = arr[pivotIndex];

        swap(arr, pivotIndex, high);
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] < pivotValue) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return i + 1;
    }
```

First Element as Pivot

In this strategy, we always select the first element of the subarray as the pivot. Update the `partition` method to use the first element in the subarray as the pivot value. Notice that `i` is set to `low` and the for loop sets `j` to `low + 1`. We are also swapping `arr[i]` and `arr[low]` after the for loop runs.

```
    private static int partition(int[] arr, int low, int high) {
        int pivotValue = arr[low];
        int i = low;
        for (int j = low + 1; j <= high; j++) {
            if (arr[j] < pivotValue) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i, low);
        return i;
    }
```

Median-of-Three Pivot Selection

In this strategy, we choose the median of the first, middle, and last elements of the subarray as the pivot. Update the `quickSort` method so that it creates the `median` variable and sets its value as the result of the `medianOfThree` method. Then pass median to the `partition` method as a fourth argument.

```
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int median = medianOfThree(arr, low, high);
            int pivotIndex = partition(arr, low, high, median);
            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }
```

Next, update the `partition` method so that it has a fourth parameter, an integer representing the pivot index. After setting `pivotValue` to the element at `pivotIndex`, swap `arr[pivotIndex]` and `arr[high]`. The variable `i` is now `low - 1`, and the loop variable `j` is initialized to `low`. Finally, the swap after the for loop switches `arr[i + 1]` and `arr[high]`.

```
    private static int partition(int[] arr, int low, int high,
        int pivotIndex) {
        int pivotValue = arr[pivotIndex];
        swap(arr, pivotIndex, high);
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivotValue) {
                i++;
                swap(arr, i, j);
            }
        }

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

Create the `medianOfThree` method that takes an array of integers, and integer representing the first element in a subarray, and another integer representing the last element in a subarray. The method should return an integer. This method calculates the midpoint of the subarray and returns the middle element if you order the elements for `arr[low]`, `arr[mid]`, and `arr[high]`.

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

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

        // At this point, arr[low] <= arr[mid] <= arr[high]
        // The median is now at arr[mid]

        return mid;
    }
```

Quick Sort Analysis

Quick Sort: Detailed Analysis

The quick sort offers some advantages over other sorting algorithms like the merge sort. However, as we have seen so far, algorithm design choices are nothing but a series of tradeoffs. Let's take a look at the quick sort and see how it compares to other sorting algorithms.

Time Complexity

- **Best Case**: If the pivot splits the array into roughly equal halves at every step, then the algorithm performs at its best, leading to a time complexity of $O(nlogn)$. This is because we're dividing the array into halves at each step (which gives the $(logn)$ factor) and then taking linear time to process each level.
- **Average Case**: On average, a quick sort runs in $O(nlogn)$ time, assuming that the pivots chosen divide the array into reasonably balanced partitions.
- **Worst Case**: The worst-case scenario for a quick sort is when the smallest or largest element is always chosen as the pivot (like a sorted or reverse sorted array). This results in a time complexity of $O(n^2)$ since we get (n) partitions of size $(n-1)$, $(n-2)$, etc.

Space Complexity

Quick sorts are an in-place sorting algorithm, meaning it sorts the array using a constant amount of extra space. The recursive stack might require $(logn)$ layers in the best case, leading to a space complexity of $O(logn)$. In the worst case, if each recursive call processes a list of size one less than the previous list, it results in $O(n)$ space complexity.

Stability

Quick sorts are **not stable** by default. Stability in sorting algorithms means that when two elements have equal keys, the original order is preserved in the sorted output. However, it can be made stable with appropriate modifications.

Advantages

Quick sorts are an in-place sort (it does not require extra storage space), making it memory efficient. They often have better cache performance than other sorting algorithms, like a merge sort, because they access array elements in a sequential manner. Quick sorts also have a tail-recursive structure, which for some architectures means a smaller call stack overhead.

Disadvantages

If not implemented correctly, quick sorts can have a time complexity of $O(n^2)$ in its worst-case scenario. which can be a severe inefficiency for large datasets. This algorithm is not stable, which means it is not a good choice for when you want to maintain the order of equal values. Finally, the choice in pivot that you make can have an effect on efficiency. A bad pivot choice can lead to reduced performance.

Quick Sorts are often the sorting algorithm of choice in practice for in-memory sorting due to its average-case performance and memory efficiency. However, care should be taken in scenarios where worst-case performance might be unacceptable. To avoid worst-case performance, many practical implementations use a hybrid approach that switches to another algorithm like insertion sort for small sub-arrays.

Formative Assessment 1

Formative Assessment 2