Learning Objectives

Learners will be able to...

- **Identify the underlying mechanics of each of the simple sorting algorithms.**
- **Implement selection, bubble and insertion sorts in Java.**
- **Analyze the time and space complexities of each algorithm.**
- **Identify practical scenarios best suited for each algorithm.**

Introduction to Sorting

Importance of Sorting

Sorting is one of the fundamental operations in computer science, and it plays a crucial role in various applications, ranging from database management to computer graphics. Understanding basic sorting algorithms not only gives insights into how data can be organized efficiently, but also provides a foundation for more complex algorithms and structures.

Recall that a binary search is more efficient than linear search. However, a binary search requires a sorted array in order to work. Understanding how to efficiently sort data structures allows you to make use of other, important algorithms.

Sorting Algorithms

In this assignment, we will dive into three basic sorting algorithms: **selection sort**, **bubble sort**, and **insertion sort**. You will implement them, compare their performances, and delve deep into their complexity analyses.

Selection Sort

Selection sort works by repeatedly selecting the smallest (or largest, depending on the ordering) element from the unsorted part of the array and swapping it with the first unsorted element. Click on the gray canvas below to start the selection sort animation.

Tasks:

1. **Implementation**:
    - Implement a selection sort in Java.
2. **Analysis**:
    - Describe the steps involved in a single iteration of the selection sort.
    - Analyze and determine the best-case, worst-case, and average-case time complexities for selection sort.

Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The pass through the list is repeated until the list is sorted. Click on the gray canvas below to start the bubble sort animation.

Tasks:

1. **Implementation**:
   - Implement a bubble sort in Java.
2. **Analysis**:
   - Describe the key operation that makes the largest unsorted element “bubble up” to its correct position.
   - Analyze and determine the best-case, worst-case, and average-case time complexities for bubble sort.

Insertion Sort

Insertion sort is a simple sorting algorithm that works by repeatedly inserting an unsorted element into a sorted list. The algorithm starts with the first element in the array, which is considered to be sorted. Then, each subsequent element is taken in turn and inserted into the sorted list in the correct position. To do this, the algorithm compares the new element to each element in the sorted list, starting with the first element. If the new element is smaller than the current element, it is swapped with the current element. This process continues until the new element is larger than or equal to all of the elements in the sorted list. Click on the gray canvas below to start the insertion sort animation.

Tasks:

1. **Implementation**:
   - Implement an insertion sort in Java.
2. **Analysis**:
   - Describe the steps involved in a single iteration of the insertion sort.
   - Analyze and determine the best-case, worst-case, and average-case time complexities for insertion sort.

Selection Sort

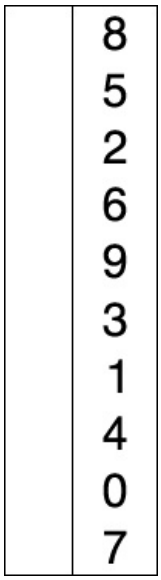
Selection Sort: Detailed Implementation

The Selection Sort algorithm divides the input list into two parts: a sorted and an unsorted sublist. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted sublist and swaps it with the leftmost unsorted element, thereby extending the sorted sublist one element at a time. Selection sort is neither a greedy algorithm nor a divide-and-conquer algorithm. It follows a straightforward, iterative approach.

Info

Sorting Mechanics

The following gif demonstrates how a selection sort works:



By Joestape89, CC BY-SA 3.0, Link

Steps

1. Start with the first element of the list, assuming it to be the minimum.
2. Move through the list and find the smallest element.
3. Swap the found minimum element with the first element.
4. Move to the next element and repeat the process until the entire list is

sorted.

Java Implementation

Create the `selectionSort` method that takes an array of integers. The method does not return a value. The first step is to calculate the length of the array that is to be sorted.

```
    public static void selectionSort(int[] arr) {
        int n = arr.length;

    }
```

Iterate over the array. Assume that the smallest value in the array is located at the same index as the looping variable (in this case `i`).

```
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            // Assume the smallest value to be at position 'i'
            int minIndex = i;

    }
```

Loop over the array one more time. This time, however, start at `i + 1`. Be sure to stop the loop one element before the end of the array. If not, you will get an index out of bounds error. If the element at index `j` is less than the element at `minIndex`, set `minIndex` to `j`.

```
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            // Assume the smallest value to be at position 'i'
            int minIndex = i;

            // Iterate over the array to find the actual minimum
        value in the unsorted sublist
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

        }
    }
```

After the second loop has finished running, swap the elements at indices `minIndex` and `i`.

```
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            // Assume the smallest value to be at position 'i'
            int minIndex = i;

            // Iterate over the array to find the actual minimum
        value in the unsorted sublist
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first
        element
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
```

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

```
[64, 25, 12, 22, 11]
```

Your code should produce:

```
Sorted array:
11 12 22 25 64
```

The `selectionSort` function handles the sorting process. It starts with the leftmost element and goes through the list to find the minimum element. Once the minimum element in the unsorted sublist is found, it is swapped with the first unsorted element. This process continues until the list is entirely sorted.

The outer loop (`for (int i = 0; i < n - 1; i++)`) effectively places the smallest element from the unsorted sublist in its correct position in the sorted sublist. The inner loop (`for (int j = i + 1; j < n; j++)`) is responsible for finding the minimum element from the unsorted sublist.

Selection Sort Analysis

A Single Iteration of Selection Sort

Each iteration of the selection sort algorithm consists of the following steps:

1. **Initialization**: At the start of each iteration, the algorithm designates the first unsorted element as the minimum (or maximum, depending on the desired order).
2. **Searching for the Actual Minimum**: The algorithm then iteratively compares this designated minimum with each subsequent unsorted element. If it encounters an element smaller than the designated minimum, it updates its notion of the minimum to this new element.
3. **Swapping**: Once the true minimum of the unsorted segment is found, the algorithm swaps it with the first unsorted element. This effectively places the smallest unsorted element in its correct final position in the sorted segment of the list.
4. **Progression**: The boundary between the sorted and unsorted segments moves one element to the right.

By the end of each iteration, the smallest element from the unsorted segment is placed in its correct position in the sorted segment.

Time Complexity Analysis

- **Best Case**: Even if the list is already sorted, the algorithm still goes through the motions of finding the minimum element in the unsorted segment for each iteration. This gives a best-case time complexity of $n \times n$ or $O(n^2)$.
- **Worst Case**: In the worst case, the list is in reverse order. However, selection sort will still perform almost the same number of operations as in the best case since it does not exploit any existing order in the list. Thus, the worst-case time complexity is also $O(n^2)$.
- **Average Case**: On average, for any random arrangement of numbers, the algorithm will still need to compare elements and search for the minimum element in the unsorted segment for each iteration. This leads to an average-case time complexity of $O(n^2)$.

Space Complexity Analysis

The algorithm sorts the list in-place, meaning it does not use any additional storage that grows with the size of the input list. Thus, the space complexity of selection sort is $O(1)$, meaning, it uses a constant amount of extra space.

Selection sort, while easy to understand and implement, is not efficient for large lists due to its quadratic time complexity. Its main advantage is its simplicity and the fact that it sorts in-place, requiring no additional space

Bubble Sort

Bubble Sort: Detailed Implementation

Bubble sort, named due to elements “bubbling up” to their correct positions, is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is entirely sorted.

Info

Sorting Mechanics

The following gif demonstrates how a bubble sort works:

Steps

1. **Traversal & Comparison**: Begin from the first element of the list and compare it with the adjacent element.
2. **Swapping**: If the current element is larger than the next one (for ascending order), they are swapped.
3. **Repeat**: This continues for the entire length of the list, ensuring the largest element “bubbles up” to the end.
4. **Decreasing the Length**: With each outer iteration, the length of the inner loop decreases by one since the largest elements from the previous iterations are already correctly placed.

Java Implementation

Create the method `bubbleSort` that takes an array of integers and returns nothing. Calculate the length of the array passed to the sorting algorithm.

```
    public static void bubbleSort(int[] arr) {
        int n = arr.length;

    }
```

Create a set of nested loops. The outer loop iterates over the array, stopping one element before the end of the array. We will see why in a bit. The inner loop starts at the beginning of the array but stops at `n - i - 1`. That means the inner loops runs one fewer time with each pass.

```
    public static void bubbleSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {

        }
    }
```

Compare the elements at indices `j` and `j + 1`. This is why we stop the outer loop from running to the end of the loop; we would get an index out of bounds error otherwise. If `arr[j]` is greater than `arr[j + 1]`, swap the two values.

```
    public static void bubbleSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
```

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

```
[62, 88, 17, 95, 34]
```

Your code should produce:

```
Sorted array:
17 34 62 88 95
```

In the provided Java code, the function `bubbleSort` manages the sorting task. The outer loop runs `n - 1` times, where `n` is the number of elements in the list. The inner loop, which is responsible for comparing and potentially swapping elements, runs `n - i - 1` times during the `i`-th outer iteration.

With each complete traversal of the inner loop, the largest unsorted element has “bubbled up” to its correct position. Thus, with each iteration of the outer loop, one more element is correctly placed, and the inner loop can consider one less element.

Bubble Sort Analysis

Bubble sort, while intuitive, is often less efficient for larger lists compared to more advanced sorting algorithms. Understanding its time and space complexities can shed light on why this is the case and in what scenarios it might be useful.

Time Complexity Analysis

- **Best Case**: This occurs when the list is already sorted. For bubble sort, even in this scenario, the algorithm will still traverse the list to check if any swaps are needed. If an optimized version is used that checks if any swaps were made in a pass and stops if none were made, then the best-case time complexity is $(O(n))$. Without this optimization, it remains $O(n^2)$.
- **Worst Case**: The list is inversely sorted, i.e., in reverse order. The algorithm has to swap every adjacent pair for the first pass, then every pair except the last one for the next pass, and so on. This gives a worst-case time complexity of $O(n^2)$.
- **Average Case**: For a randomly ordered list of length n, the average number of comparisons and swaps in each pass is roughly half of the list length. As the outer loop also runs n times, the average-case time complexity is $O(n^2)$.

Space Complexity Analysis

Bubble sort is an in-place sorting algorithm. It requires only a constant amount of extra memory (for swapping elements, etc.), irrespective of the size of the input list. Hence, the space complexity is $O(1)$.

Things to keep in mind when thinking of a bubble sort:

- **Adaptive Nature**: Bubble sort is adaptive, meaning its efficiency improves if given a partially sorted list. The optimized version (that checks for swaps) will run faster if there are fewer elements out of order.
- **Stable Sort**: Bubble sort is stable, implying that relative ordering of equal elements remains unchanged even after sorting. This can be crucial in scenarios where the order of duplicate elements carries significance.

- **Usage**: Due to its quadratic time complexity, bubble sort is not suitable for large datasets when compared to more efficient algorithms like merge sort or quicksort. However, it can be a suitable choice for small datasets or for teaching purposes due to its simplicity.

Understanding the intricacies of bubble sort gives one a foundational knowledge of sorting algorithms. While not the most efficient, its conceptual simplicity offers a stepping stone to grasp more complex algorithms. When dealing with real-world applications, it's essential to weigh the benefits of simplicity against the needs of efficiency and choose an appropriate sorting algorithm accordingly.

Insertion Sort

Insertion Sort: Detailed Implementation

Insertion Sort is a simple sorting algorithm. It operates by considering one element at a time, inserting it into its correct position within the already-sorted part of the array. Insertion sorts are really useful to compare it to more advance ones like quicksort, heapsort, or merge sort that we will cover in the next assignment.

Info

Sorting Mechanics

The following gif demonstrates how an insertion sort works:

Steps

1. **Initialization**: Start from the second element (index 1) assuming the element at index 0 is sorted.
2. **Extraction**: Extract the current element to be compared. This element will be compared with the elements to its left.
3. **Comparison & Insertion**: Compare the current element with the previous elements:
   - If the current element is smaller than the previous element, we continue comparing with the elements before until we reach an element smaller or reach the beginning of the array.
   - Insert the current element in its correct position so that the left part of the array remains sorted.
4. **Iteration**: Repeat the process for each of the elements in the array.

Java Implementation

Start by creating the `insertionSort` method that takes an array of integers and does not return anything. Iterate over the array starting at index `1`. Create the variable `key` which is assigned `arr[i]` and variable `j` which is assigned to `i - 1`. If we started iterating at index `0` then creating `j` would result in an index out of bounds error.

```
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;

    }
```

Create a while loop that runs as long as `j >= 0` and `arr[j] > key`. As long as this loop is running, assign `arr[j + 1]` the value of `arr[j]`. Remember, `j = i - 1` so `arr[j + 1]` is actually `arr[i]`. However, we are not writing over any data since `arr[i]` has been stored in the variable `key`. Then decrement `j` by `1`.

```
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;

            // Move elements of arr[0..i-1] that are greater
        than the key
            // to one position ahead of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }

        }
    }
```

Finally, after the while loop concludes, assign `arr[j + 1]` the value of `key`. Think of an insertion sort as dividing the array into the left half and the right half. The dividing line between the two halves is the index `i`. The left half is sorted, while the right half is unsorted. Assigning `key` to `arr[j + 1]` means that you are placing `key` into its proper position on the left half of the array.

```
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;

            // Move elements of arr[0..i-1] that are greater
        than the key
            // to one position ahead of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
```

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

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

Your code should produce:

```
Sorted array:
5 6 11 12 13
```

Insertion Sort can be visualized as the method most people use to arrange playing cards in their hands. While not the most efficient for large datasets, its simplicity offers advantages:

- **Adaptive**: It's efficient for nearly-sorted lists.
- **Stable**: Retains relative order of input data.
- **In-Place**: Requires a constant amount of extra memory.

Insertion Sort Analysis

After understanding the mechanics and implementation of an insertion sort, the next vital step is to analyze its performance in terms of time and space complexity. This allows us to gauge its efficiency and identify appropriate scenarios for its use.

Time Complexity Analysis

- **Best Case**: The best-case scenario for an insertion sort is when the input list is already sorted. In this case, the inner loop won't be executed, resulting in a time complexity of $O(n)$.
- **Worst Case**: The worst-case scenario is when the input list is sorted in reverse order. For every `i`th element, the inner loop does `i` operations. Summing this over all `n` elements results in a time complexity of $O(n^2)$.
- **Average Case**: On average, the inner loop iterates half of the time compared to the worst case, leading to a time complexity of $O(n^2)$.

Space Complexity Analysis

Insertion sort is an in-place sorting algorithm. This means it uses a constant amount of extra space (for variables like `key` and `j` in our implementation). Hence, the space complexity is $O(1)$.

Auxiliary Space Analysis

Recall that auxiliary space is the extra space or temporary space used by an algorithm. For an insertion sort, the auxiliary space is $O(1)$ since it only uses a constant amount of additional space (the variable `key`).

Things to keep in mind when thinking of insertion sorts:

- **Adaptivity**: Insertion sorts are adaptive. If only a few elements are out of their correct positions (i.e., the list is partially sorted), an insertion sort can be much faster than other sorting methods. For almost sorted data, it becomes linear in time complexity, i.e., $O(n)$.
- **Stability**: Insertion sorts are stable, meaning that the relative order of equal data elements remains unchanged. This property can be crucial in specific applications where data order matters.

While an insertion sort has a quadratic time complexity, making it less efficient for large datasets, its simplicity and efficiency on nearly sorted data make it a good choice in specific scenarios. It works particularly well for small datasets and can outperform more advanced algorithms in such cases. Insertion sorts often used as the base case for higher overhead divide-and-conquer sorting algorithms, like merge sort or quicksort, when the problem size is below a certain threshold.

Formative Assessment 1

Formative Assessment 2