SELECTION SORT

UNIT 3: SORTING & SEARCHING

WHAT IS SELECTION SORT ?

- Selection sort is a simple comparison-based sorting algorithm.
- It works by repeatedly finding the minimum element from the unsorted part of the list and placing it at the beginning.
- In each iteration, the smallest element is selected and swapped with the element at the current position.
- This process continues until the entire list is sorted.

SELECTION SORT - ALGORITHM

1) **Find the minimum element:**
   - Start with the first element as the minimum.
   - Iterate through the remaining unsorted elements and update the minimum if a smaller element is found.
2) **Swap the minimum element:**
   - Once the minimum element is determined, swap it with the element at the current position.
   - This places the minimum element in its correct sorted position.
3) **Repeat for the remaining list:**
   - Move to the next position and repeat steps 1 and 2 for the remaining unsorted part of the list.
   - Each iteration will find the minimum element from the remaining unsorted portion and swap it into its correct position.
4) **Continue until the list is sorted:**
   - Repeat the above steps until the entire list is sorted.
   - The sorted part of the list will gradually grow from left to right.

WORKING OF SELECTION SORT

EXAMPLE DEMONSTRATION

Initial unsorted list: [5, 2, 8, 3, 1]

First iteration:

- Minimum element: 1 (at index 4)
- Swap: Swap 1 with the element at the first position
- Updated list: [1, 2, 8, 3, 5]

Second iteration:

- Minimum element: 2 (at index 1)
- Swap: No swap as the minimum element is already in the correct position
- Updated list: [1, 2, 8, 3, 5]

Third iteration:

- Minimum element: 3 (at index 3)
- Swap: Swap 3 with the element at the third position
- Updated list: [1, 2, 3, 8, 5]

Fourth iteration:

- Minimum element: 5 (at index 4)
- Swap: Swap 5 with the element at the fourth position
- Updated list: [1, 2, 3, 5, 8]

Fifth iteration:

- The list is already sorted. No minimum element to find or swap.

The final sorted list: [1, 2, 3, 5, 8]

COMPLEXITY ANALYSIS

Time Complexity:

- Selection sort has a time complexity of O(n^2), where n is the number of elements in the list. It involves two nested loops, making it less efficient for large input sizes.

Space Complexity:

- Selection sort has a space complexity of O(1) since it operates on the input list itself without requiring additional space.

Despite its simplicity, selection sort is not suitable for large datasets due to its quadratic time complexity.

IMPLEMENTATION IN PYTHON

```
01  def selection_sort(arr):
02      n = len(arr)
03
04      for i in range(n-1):
05          # Find the minimum element in the unsorted part of the list
06          min_index = i
07          for j in range(i+1, n):
08              if arr[j] < arr[min_index]:
09                  min_index = j
10
11          # Swap the minimum element with the current position
12          arr[i], arr[min_index] = arr[min_index], arr[i]
13
14      return arr
```

1. The **selection_sort** function takes an input list, **arr**, and sorts it using the selection sort algorithm. The sorted list is then returned.
2. **n = len(arr)** stores the length of the input list in the variable **n**. This will be used to control the number of iterations in the outer loop.
3. The outer loop **for i in range(n-1):** iterates over the list, excluding the last element. It represents the current position from which the minimum element will be selected and swapped.
4. Inside the outer loop, we initialize **min_index = i**, which represents the index of the minimum element found so far in the unsorted part of the list.
5. The inner loop **for j in range(i+1, n):** starts from the next position after **i** and iterates until the end of the list. It searches for an element smaller than the current minimum element.
6. if **arr[j] < arr[min_index]:** checks if the element at index **j** is smaller than the current minimum element. If it is, we update **min_index** to store the index of the new minimum element.
7. After the inner loop completes, we have found the minimum element in the unsorted part of the list. We swap it with the element at the current position using the line **arr[i], arr[min_index] = arr[min_index], arr[i].**
8. The outer loop continues for the next position, repeating the process of finding the minimum element and swapping it with the current position.
9. Finally, the sorted list, **arr**, is returned.

COMPARISON OF BUBBLE SORT & SELECTION SORT

Both selection sort and bubble sort are simple comparison-based sorting algorithms with quadratic time complexity. However, there are some differences in their performance characteristics:

Selection Sort:

- In each iteration of the outer loop, selection sort finds the minimum element from the unsorted part of the list and swaps it with the current position. This guarantees that after each iteration, the smallest element is correctly placed at the beginning of the sorted portion.
- Selection sort has a time complexity of O(n^2), where n is the number of elements in the list, regardless of the initial ordering of the elements. It performs the same number of comparisons and swaps in all cases.
- Selection sort has a space complexity of O(1) since it operates on the input list itself without requiring additional space.
- Selection sort performs n-1 comparisons in the first iteration, n-2 comparisons in the second iteration, and so on until the last two elements, resulting in a total of (n-1) + (n-2) + ... + 2 + 1 = (n^2 - n) / 2 comparisons. Similarly, it requires (n-1) swaps in total.

Bubble Sort:

- Bubble sort compares adjacent elements and swaps them if they are in the wrong order. In each iteration, the largest unsorted element "bubbles" to the end of the list.
- Bubble sort also has a time complexity of O(n^2), but its performance can vary depending on the initial ordering of the elements. In the best-case scenario, where the list is already sorted, bubble sort can have a time complexity of O(n) as it requires only one pass to determine that the list is already sorted.
- Bubble sort has a space complexity of O(1) since it operates on the input list itself without requiring additional space.
- Bubble sort may require more swaps than selection sort, as it performs a swap for every inversion (pair of adjacent elements in the wrong order). In the worst-case scenario, where the list is sorted in descending order, bubble sort requires (n^2 - n) / 2 swaps, similar to selection sort.
- Overall, both selection sort and bubble sort are not considered efficient for large datasets due to their quadratic time complexity. However, in practical scenarios, selection sort generally performs better than bubble sort due to a slightly lower number of swaps. Nevertheless, other more efficient sorting algorithms, such as merge sort or quicksort, are typically preferred for larger datasets.

Download Lecture Slides & Executable Programs from

https://github.com/ashiqirphan-AI/AD3251-DSD-Programs