Selection And Merge Sort

Group:04

January 23, 2024

Presented by
2003008-Nahid Niyaz
2003016-Jobayer Mansur Mufti
2003019-MD. Shakibul Hasan
2003028-Md. Abdullah Al Mamun
2003038-Abir Chandra Das
2003055-Eazdan Mostafa Rafin

Selection Sort

Key Points - Selection Sort

- Selection Sort is a simple sorting algorithm.
- It works by finding the minimum or maximum element in the unsorted part of the array and placing it in the sorted part.
- In each pass, one element is sorted. For n elements, $n - 1$ passes are required.
- The algorithm starts by setting the first element as the minimum.
- It then compares the minimum with the second element. If the second element is smaller, it assigns the second element as the new minimum.
- This process is repeated for each element in the unsorted part of the array until the last element is reached.
- After each pass, the minimum is placed at the front of the unsorted list.
- Indexing starts from the first unsorted element after each pass.
- The algorithm can sort the array in ascending or descending order, depending on whether it finds the minimum or maximum element in each iteration.
- Not suitable for large datasets due to its average and worst-case

Selection Sort Algorithm

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int minIndex = i;
        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;
    }
}
```

Selection Sort Step-by-Step

Selection Sort Steps

- Start with the first element as the minimum.
- Compare the minimum with the second element. If the second element is smaller, update the minimum.
- Repeat the process for each element in the unsorted part of the array until the last element is reached.
- After each pass, place the minimum at the front of the unsorted list.
- Move to the next unsorted element and repeat the process until the entire array is sorted.

Selection Sort Visualization

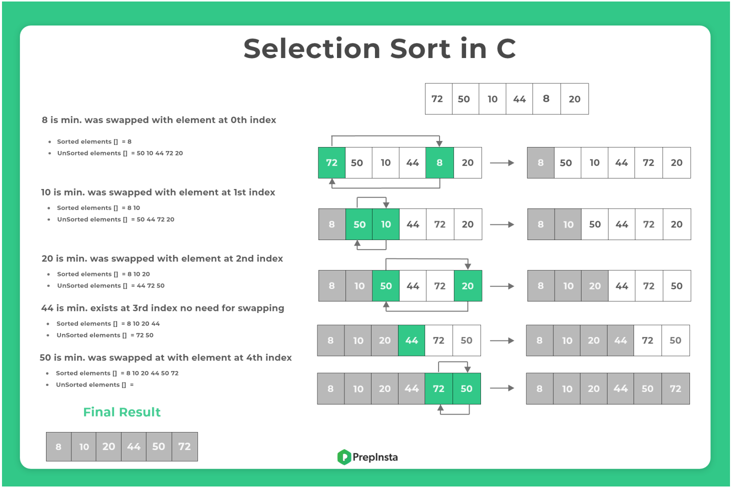


Figure: Selection Sort Visualization

Time Complexity of Selection Sort

- Selection Sort has a time complexity of $O(n^2)$ in both the average and worst cases.
- For each element in the array, it needs to traverse the remaining unsorted part to find the minimum or maximum element.
- The nested loop structure results in quadratic time complexity.
- Inefficient for large datasets as the number of comparisons and swaps grows quadratically with the size of the input.
- Despite its simplicity, it's not suitable for large datasets compared to more efficient sorting algorithms.

Application of Selection Sort

- **Small Datasets:**
  - Selection Sort can be suitable for sorting small datasets where its simplicity may outweigh its relatively higher time complexity.
- **Educational Purposes:**
  - It is often used in educational settings to introduce the concept of sorting algorithms due to its straightforward logic.
- **Memory Usage:**
  - Selection Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space, making it useful in situations with limited memory.
- **Stable Sorting:**
  - While not stable by default, modifications can be made to Selection Sort to make it stable (maintaining the relative order of equal elements).

Merge Sort

Introduction to Merge Sort

Key Points

- Merge Sort is a popular sorting algorithm.
- It follows the **divide-and-conquer** paradigm.
  - **Divide :** The unsorted list is recursively divided into smaller sublists until each sublist contains only one element
  - **conquer :** Once the list is divided into individual elements (each considered a sorted sublist of size 1), the algorithm starts merging these sublists in a way that builds up a sorted order. This is the "conquer" step, where the individual sorted sublists are merged to produce larger sorted sublists.
- Efficient for large datasets.

Merge Sort Algorithm

1. Divide the unsorted list into *n* sublists.

Listing 1: Merge Sort in C++

```cpp
void merge_sort(int ax[], int lb, int ub) {
    if (lb < ub) {
        int mid = (lb + ub) / 2;
        merge_sort(ax, lb, mid);
        merge_sort(ax, mid + 1, ub);
        merge(ax, lb, mid, ub);
    }
}

```

Merge Sort Algorithm

1. Repeatedly merge sublists.

Listing 2: Merge Sort in C++

```
void merge(int ax[], int lb, int mid, int ub) {
  int i, j, k;
  i = lb;
  j = mid + 1;
  k = lb;
  int bx[n];
  while (i <= mid && j <= ub) {
      if (ax[i] <= ax[j]) {
          bx[k] = ax[i];
          i++;
          k++;
      } else {
          bx[k] = ax[j];
          j++;
          k++;
      }
  }
  if (i > mid) {
      while (j <= ub) {
          bx[k] = ax[j];
          j++;
          k++;
      }
  } else {
      while (i <= mid) {
          bx[k] = ax[i];
          i++;
          k++;
      }
  }
  for (int k = lb; k <= ub; k++) {
      ax[k] = bx[k];
  }
}

```

Merge Sort Example

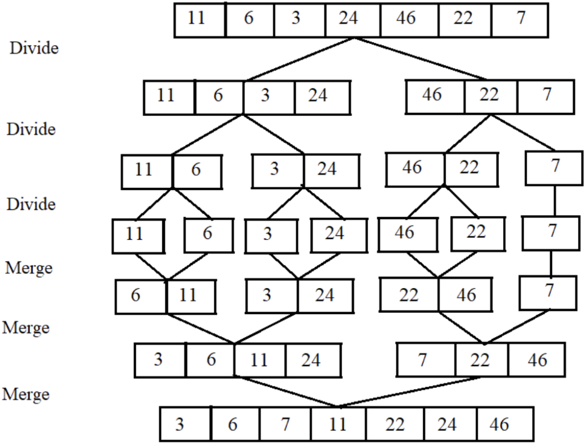
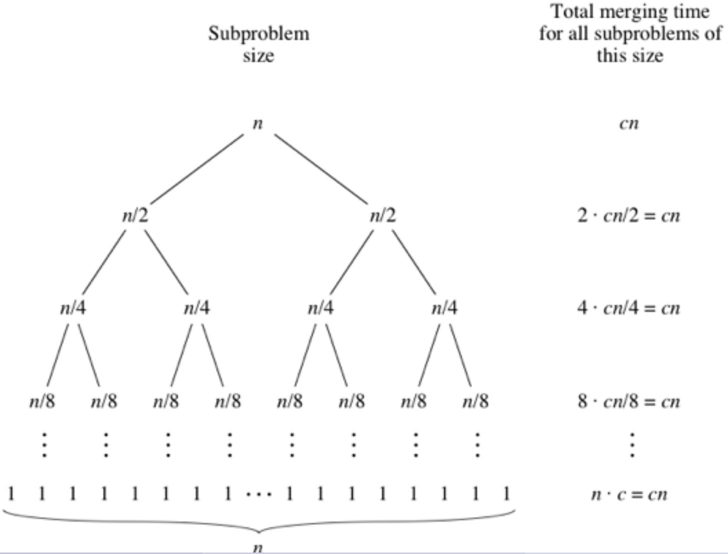


Figure: Illustration of the Merge Sort Algorithm

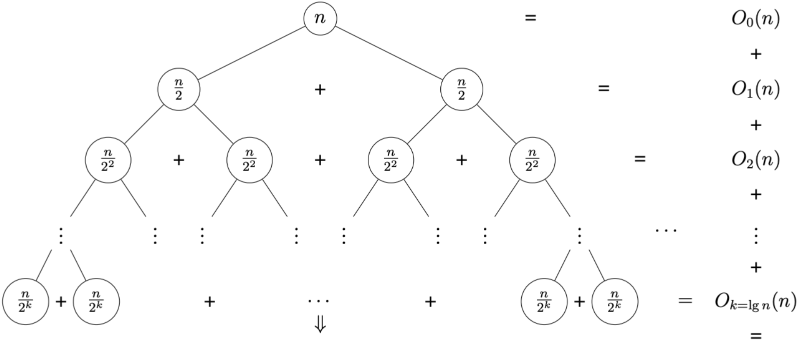
Time Complexity of Merge Sort

Complexity Analysis



Time Complexity of Merge Sort

Complexity Analysis



$$\Downarrow$$

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \log n)$$

Time Complexity of Merge Sort

Complexity Analysis

- The divide step takes constant time. $O(1)$
- Total Merging Time $= l * c * n$
  - Here, $l = O(n \log n) + 1$
  - So, Total Merging Time will Be $= O(n \log n)$
- Efficient for large datasets.

Merge Sort Time Complexity - Recursive Function

Time Complexity Recurrence Relation

The time complexity of Merge Sort is often expressed using the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Time Complexity Breakdown

- $T(n)$: Time complexity for an input of size n.
- $2T\left(\frac{n}{2}\right)$: Time complexity for dividing the array into two halves and recursively sorting each half.
- $O(n)$: Time complexity for merging the two sorted segments.

Master Theorem Analysis

Applying the Master Theorem to the recurrence relation yields a time complexity of $O(n \log n)$.

Merge Sort Time Complexity - Recursive Function

Proof:

$T(n) = 2\,T(n/2) + n$

$= (n/2) \lg (n/2) + 2(n/2) + n$ *(by induction hypothesis)*

$= n \lg (n/2) + 2n$

$= n \lg n - 1)\ 1) + 2n$

$= n \lg n + n$

Space Complexity of Merge Sort

Memory Usage

Merge sort has a space complexity of $O(n)$ due to the need for additional memory to store temporary arrays during the merging process.

Explanation

- Merge sort divides the array into halves recursively, requiring additional memory for each recursive call.
- The merging step involves copying elements to a temporary array, adding to the overall space complexity.
- The total space complexity is $O(n)$ as the maximum depth of the recursion is $\log n$ and at each level, n elements are stored.

Summery of Space Complexity

While merge sort has an optimal time complexity, its space complexity may be a consideration for large datasets with limited memory.

Applications For Merge Sort

Real-Life Applications

- **External Sorting:** Used in scenarios with large datasets that don't fit in main memory.
- **Database Management:** Efficient sorting in database systems.
- **Network Routing:** Sorting routes to optimize network communication.
- **Parallel Processing:** Suitable for parallel computing environments.
- **File Merging:** Merging sorted files efficiently.
- **Flight Scheduling:** Organizing flight information based on criteria.
- **Inversion Counting:** Useful in data analysis, statistics, and optimization.
- **External Memory Algorithms:** Widely used in algorithms for large external datasets.

Merge Sort vs. Selection Sort

Merge Sort

- Uses a divide and conquer strategy.
- Guaranteed time complexity: $O(n \log n)$.
- Efficient for large datasets.
- Stable sorting algorithm.

Selection Sort

- Simple and intuitive.
- Guaranteed time complexity: $O(n^2)$.
- Inefficient for large datasets.
- In-place sorting algorithm.