Divide and Conquer Approach

Mouli Dutta

21-07-2023

Contents

1 Introduction

The ”Divide and Conquer” approach is a powerful algorithmic strategy used to solve complex problems by breaking them down into simpler sub-problems, solving each sub-problem independently, and then combining their solutions to obtain the final result. It is a fundamental algorithmic paradigm and is widely used in various fields, including computer science, mathematics, and engineering.

The basic idea behind the divide and conquer approach is to divide a large problem into smaller, more manageable sub-problems, which are usually of the same type as the original problem but on a reduced input size. These sub-problems are solved recursively until they become simple enough to solve directly. Once the solutions to the sub-problems are obtained, they are combined to produce the solution to the original problem.

The divide and conquer approach typically follows three main steps:

Divide: The first step is to break down the original problem into smaller sub-problems. This division should be done in a way that each sub-problem resembles the original problem, but on a smaller scale. The goal is to simplify the problem and make it easier to handle.

Conquer: Once the problem is divided into sub-problems, each sub-problem is solved independently. If the sub-problems are still complex, the algorithm will recursively apply the divide and conquer approach to further break them down into even smaller sub-problems until they become trivial enough to be solved directly.

Combine: After obtaining the solutions to the sub-problems, the algorithm combines them to produce the final solution to the original problem.

2 Example

A classic example of the divide and conquer approach is the ”Merge Sort” algorithm, used to efficiently sort an array of elements. The algorithm divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves to obtain the final sorted array.

3 Merge Sort Algorithm

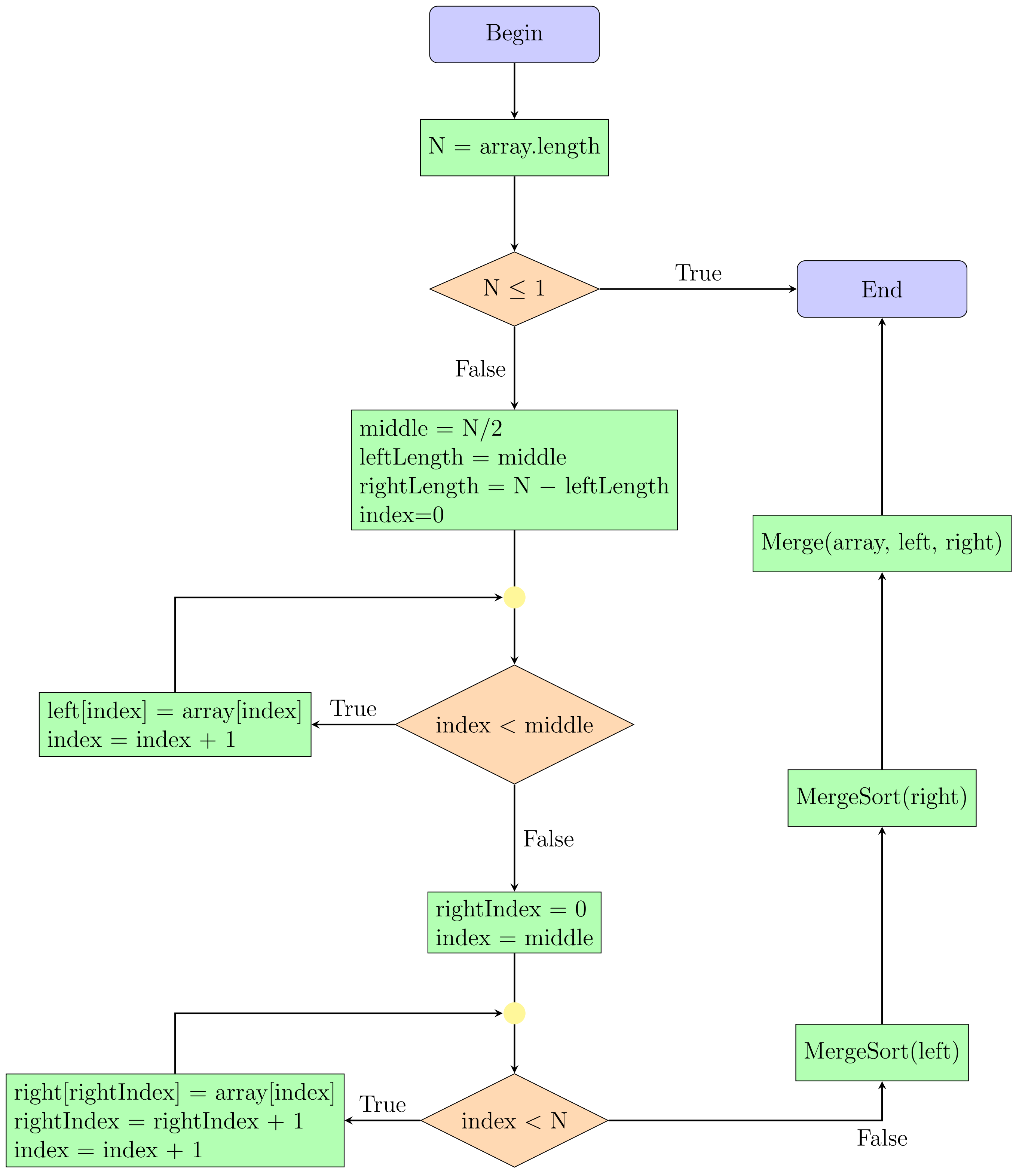
Algorithm 1 Algorithm for Merge

```
 1: procedure: Algorithm for Merge( Merge(A, low, high, mid))
 2: Input: Array A[low, high] is the input array and $mid = \frac{(low+high)}{2}$
 3: Output: Intermediate sorted array
 4: $i \leftarrow low$
 5: $k \leftarrow low$
 6: $j \leftarrow mid + 1$
 7: while $(i \leq mid \ \&\& \ j \leq high)$ do
 8:     if $(A[i] \leq A[j])$ then
 9:         $B[k] \leftarrow A[i]$                    ▷ B[low, high] is an auxiliary array
10:         $i \leftarrow i + 1$
11:     else
12:         $B[k] \leftarrow A[j]$
13:         $j \leftarrow j + 1$
14:     end if
15:     $k \leftarrow k + 1$
16: end while
17: while $(i \leq mid)$ do
18:     $B[k] \leftarrow A[i]$
19:     $i \leftarrow i + 1$
20:     $k \leftarrow k + 1$
21: end while
22: while $(j \leq high)$ do
23:     $B[k] \leftarrow A[j]$
24:     $j \leftarrow j + 1$
25:     $k \leftarrow k + 1$
26: end while
27: for $i \leftarrow low$ to k in steps of 1 do
28:     $A[i] \leftarrow B[i]$
29: end for
```

Algorithm 2 Algorithm for Merge Sort

```
1: procedure: Algorithm for Merge Sort( MergeSort(A, low, high))
2: Input: Array A[low, high] is the input array
3: Output: The elements in non-decreasing order
4: if ($low < high$) then                                   ▷ if runsize ! = 1
5:     $mid \leftarrow \frac{(low+high)}{2}$                ▷ mid == midpoint
6:     MergeSort(A, low, mid)                               ▷ sort the left run
7:     MergeSort(A, mid+1, high)                            ▷ sort the right run
8:     Merge(A, low, high, mid)      ▷ merge two intermediate sorted arrays A[low, mid] and
   A[mid+1, high]
9: end if
10: return
```

4 Merge Sort Flowchart



5 Python code for Merge Sort

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)

def merge(left, right):
    merged_arr = []
    left_idx, right_idx = 0, 0

    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] < right[right_idx]:
            merged_arr.append(left[left_idx])
            left_idx += 1
        else:
            merged_arr.append(right[right_idx])
            right_idx += 1

    merged_arr.extend(left[left_idx:])
    merged_arr.extend(right[right_idx:])

    return merged_arr

# Example usage:
unsorted_list = [38, 27, 43, 3, 9, 82, 10]
sorted_list = merge_sort(unsorted_list)
print("Original List:", unsorted_list)
print("Sorted List:", sorted_list)


# Output:
# Original List: [38, 27, 43, 3, 9, 82, 10]
# Sorted List: [3, 9, 10, 27, 38, 43, 82]
```