Divide-and-Conquer

2200010013, 220001014, 220001015

January 16th , 2024

1 Introduction

Divide the problem into one or more **disjoint** sub-problems that are smaller instances of the same problem.

Conquer the sub-problems by solving them individually.

Merge the sub-problem solutions to form a solution to the original problem

Following problems are to be discussed:

- Merge Sort
- Quick Sort
- Maximum sub-array sum

2 Merge-Sort

2.1 Merge

The pseudo-code for merge step is as follows:

procedure MERGE(A, m, B, n, C)
  $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$
  while $k < m + n$ **do**
    if $i == m$ **then** ▷ works when array A is exhausted
      $C[k] \leftarrow B[j]$, $k \leftarrow k + 1$, $j \leftarrow j + 1$
    else if $j == n$ **then** ▷ works when array B is exhausted
      $C[k] \leftarrow A[i]$, $k \leftarrow k + 1$, $i \leftarrow i + 1$
    else if $A[i] \leq B[j]$ **then**
      $C[k] \leftarrow A[i]$, $k \leftarrow k + 1$, $i \leftarrow i + 1$
    else if $A[i] > B[j]$ **then**
      $C[k] \leftarrow B[j]$, $k \leftarrow k + 1$, $j \leftarrow j + 1$
    end if
  end while
end procedure

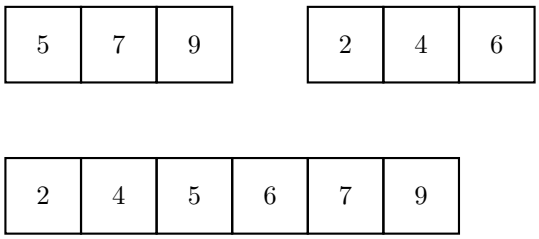

Figure 1: Merging Sorted Arrays $A = \{5, 7, 9\}$ and $B = \{2, 4, 6\}$

2.2 Complexity of merging operation

The time to merge a total of $m+n$ items is $\Theta(m+n)$. Within each iteration of while loop:

- 7 operations are performed in the worst case.
- 4 operations are performed in the best case.

Merging sorted arrays **in-place** is time consuming because the algorithm needs to shift elements within the arrays to make room for the merged elements.

2.3 Sort

procedure MERGESORT(A, *left*, *right*, B) ▷ Array goes from *left* to *right* -1
  if *right* $-$ *left* $==1$ **then**
    $B[0] = A[left]$ ▷ $\Theta(1)$
  end if
  if *right* $-$ *left* >1 **then**
    $mid = left + \frac{right-left}{2}$
    MERGESORT(A, *left*, *mid*, L) ▷ $T(\frac{n}{2})$
    MERGESORT(A, *mid*, *right*, R) ▷ $T(\frac{n}{2})$
    MERGE(L, *mid* $-$ *left*, R, *right* $-$ *mid*, B) ▷ $\Theta(n)$
  end if
end procedure

The recurrence relation is given by:

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

This recurrence relation is in the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where:

$$\begin{aligned} a &= 2 \quad \text{(number of sub-problems)} \\ b &= 2 \quad \text{(factor by which the problem size is reduced)} \\ f(n) &= \Theta(n) \quad \text{(cost of dividing and combining steps)} \end{aligned}$$

Comparing with the given recurrence relation:

$$a = 2$$
$$b = 2$$
$$f(n) = \Theta(n)$$

The solution for Master Theorem Case 2 is:

$$T(n) = \Theta(n^{log_b a} \cdot \log n) = \Theta(n \log n)$$

Therefore, the time complexity of Merge Sort is $O(n \log n)$.

2.4 Use-Cases

Merge sort is useful in distributed systems in which the problem is unsolvable for one memory unit. Therefore, it is distributed among multiple memory units and the solutions are recombined to obtain the final solution,

Merge sort is less useful for low-memory systems.

Since **Divide** step precedes the **Conquer** step, merge-sort is recursive.

Quick sort can be iterative as well since **Partition** step has the precedence.

3 Quick-Sort

QuickSort is based on the Divide and Conquer paridgm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

3.1 Choosing Pivot Element

There are many different versions of quickSort that pick the pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot.
- Pick a random element as a pivot.
- Pick the median as the pivot.

3.2 Partition

- Suppose, we assume last element as a pivot.

The pseudo-code for partition is as follows:

```
procedure PARTITION(A, low, high)
    pivot ← A[high]
    PI ← low − 1, j ← low
    while j <= high do
        if A[j] < pivot then
            PI ← PI + 1
            SWAP(A[PI], A[j])
        end if
        j ← j + 1
    end while
    SWAP(A[PI + 1], A[high])
    RETURN(PI + 1)
end procedure
```

- We can proceed similarly using the first element as pivot as well.

3.3 Complexity of Partition function

Time Complexity to find the index for pivot is O(n), because both the pointers combined traversing whole array sequentially to find appropriate position.

3.4 Sort

```
procedure QUICKSORT(A, low, high)
    if low < high then
        Pindex ← PARTITION(A, low, high)
        QUICKSORT(A, low, Pindex − 1)
        QUICKSORT(A, Pindex + 1, high)
    end if
end procedure
```

The recurrence relation for the best case is given by(Which is possible when pivot is median element):

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

This recurrence relation is in the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where:

$$a = 2$$
$$b = 2$$
$$f(n) = \Theta(n)$$

By solving this recurrence relation using Master Theorem Case-2, We get:

$$T(n) = \Theta(n^{log_b a} \cdot \log n) = \Theta(n \log n)$$

Therefore, best case time complexity of the Quick Sort is $O(n \log n)$.

The recurrence relation for the worst case(occurs when the array is already sorted or reverse sorted) is given by:

$$T(n) = T(n-1) + O(n)$$

By solving this recurrence relation using substitution method, we get worst case time complexity T(n)=O(n^2).

3.5 Use-Cases

Quick Sort is useful in all the cases except for sorted array and reverse sorted array.Quick Sort is an in-place algorithm. However, the space complexity is O(log(n)) due to the usage of system stack.

Quick Sort is used in many inbuilt sorting algorithms like sort() , because the probability of getting worst cases are very less and also in-place as well.

Quick Sort can be implemented using both recursive approach and iterative approach.

4 Problem: Maximum Sum Sub-Array

- Given an array arr[] of size N. The task is to find the sum of the contiguous subarray within a arr[] with the largest sum.
- We can think of a Naive approach with time complexity of O(n^2) by running two for loops to get all the possible subarrays.
- To get the solution with a better time complexity we can use a Divide and Conquer approach.
- If we divide the array in two equal halves , we can get the maximum possible sum in 3 different ways: 1)The sum lies entirely in the first half. 2)The sum lies entirely in the second half. 3)The sum lies across both the halves.

- For the first two cases, we can simply call the same function for the first half of the array and the second half of the array respectively.
- The third part is interesting. If we can to consider the case where the sum is lying across the two halves, it will need to include the middle element.So this task is now reduced to calculating the sum with the condition that the middle element is included. Now this can be calculated by just adding the max sum possible while going to the right linearly , while starting at the middle element and the max possible sum while going to the left from the middle element. This whole operation will have O(n) time complexity.

  We get the recurrence relation as:

$$T(n) = 2T(n/2) + O(n)$$

  The net time complexity of this approach is: O(nlogn), which you can see is an improvement over the Naive approach.