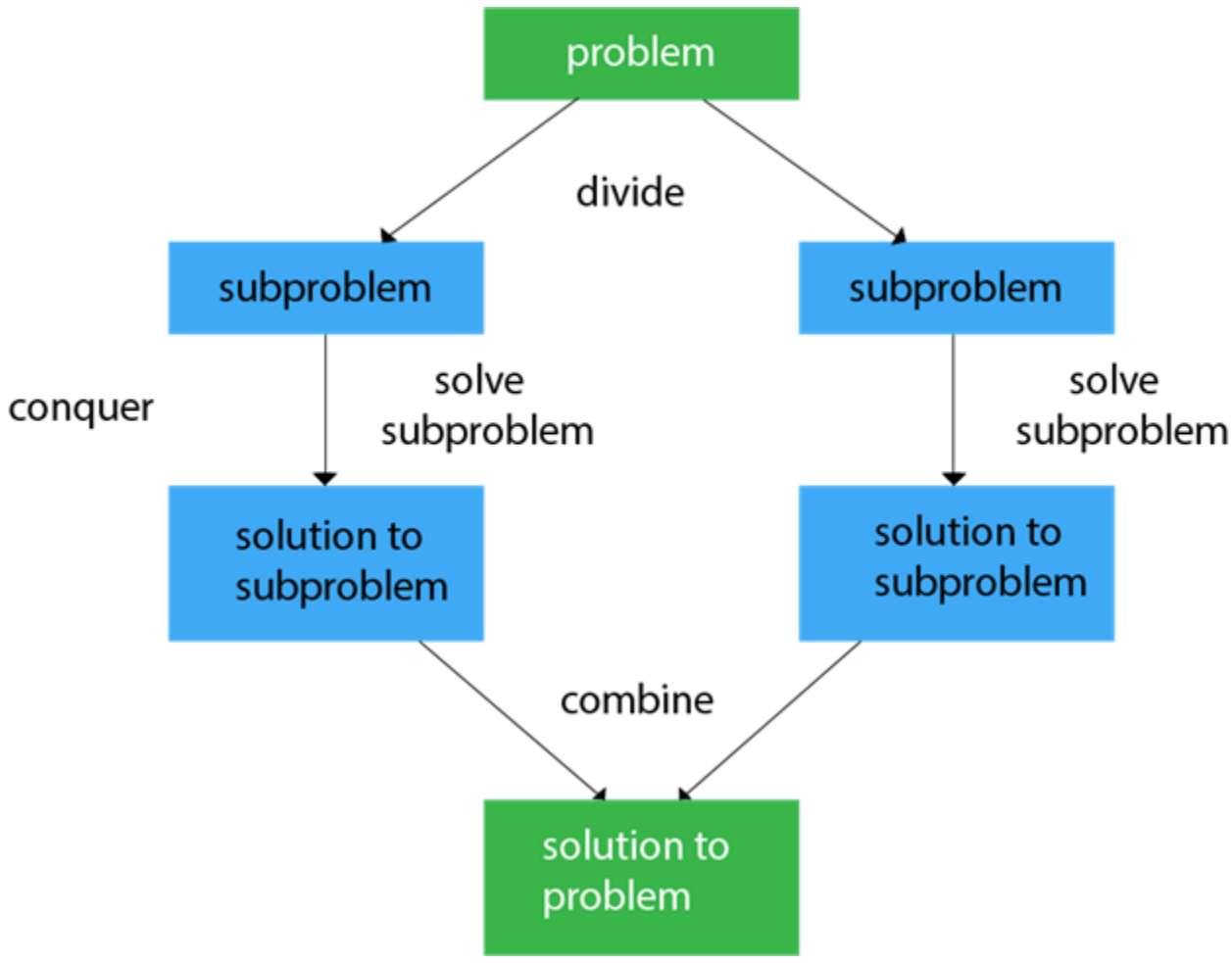
Divide and Conquer Introduction

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.
5. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.
6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

Max - Min Problem

Problem: Analyze the algorithm to find the maximum and minimum element from an array.

Algorithm: Max ?Min Element (a [])

```
Max:  a [i]
Min:   a [i]
For i= 2 to n do
If a[i]> max then
max = a[i]
if a[i] < min then
min: a[i]
return (max, min)
```

Analysis:

Method 1: if we apply the general approach to the array of size n, the number of comparisons required are 2n-2.

Method-2: In another approach, we will divide the problem into sub-problems and find the max and min of each group, now max. Of each group will compare with the only max of another group and min with min.

Let n = is the size of items in an array

Let T (n) = time required to apply the algorithm on an array of size n. Here we divide the terms as T(n/2).

2 here tends to the comparison of the minimum with minimum and maximum with maximum as in above example.

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

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2 \rightarrow \text{Eq (i)}$$

T (2) = 1, time required to compare two elements/items. (Time is measured in units of the number of comparisons)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + 2 \rightarrow \text{Eq (ii)}$$

Put eq (ii) in eq (i)

$$\begin{aligned} T(n) &= 2\left[2T\left(\frac{n}{2^2}\right) + 2\right] + 2 \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2 \end{aligned}$$

Similarly, apply the same procedure recursively on each subproblem or anatomy

{Use recursion means, we will use some stopping condition to stop the algorithm}

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \quad \ldots\ldots \text{(Eq. 3)}$$

Recursion will stop, when $\frac{n}{2^i} = 2 \Rightarrow n = 2^{i+1}$ → (Eq. 4)

Put the equ.4 into equation3.

$$\begin{aligned} T(n) &= 2^i T(2) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \\ &= 2^i .1 + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \\ &= 2^i + \frac{2\left(2^i - 1\right)}{2-1} \\ &= 2^{i+1} + 2^i - 2 \\ &= n + \frac{n}{2} - 2 \\ &= \frac{3n}{2} - 2 \end{aligned}$$

Number of comparisons requires applying the divide and conquering algorithm on n elements/items = $\frac{3n}{2}-2$

Number of comparisons requires applying general approach on n elements = (n-1) + (n-1) = 2n-2

From this example, we can analyze, that how to reduce the number of comparisons by using this technique.

Analysis: suppose we have the array of size 8 elements.

Method1: requires (2n-2), (2x8)-2=14 comparisons

Method2: requires $\frac{3\times 8}{2}-2=10 \ \ comparsions$

It is evident; we can reduce the number of comparisons (complexity) by using a proper technique.

Binary Search

1. In Binary Search technique, we search an element in a sorted array by recursively dividing the interval in half.

2. Firstly, we take the whole array as an interval.

3. If the Pivot Element (the item to be searched) is less than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.

4. If the Pivot Element (the item to be searched) is greater than the item in the middle of the interval, we discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.

5. Repeatedly, check until the value is found or interval is empty.

Analysis:

1. **Input:** an array A of size n, already sorted in the ascending or descending order.
2. **Output:** analyze to search an element item in the sorted array of size n.
3. **Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.

- Set BEG = 1 and END = n
- Find mid = $\text{int}\left(\frac{beg+end}{2}\right)$
- Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1

Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.

And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

T (n) = $T\left(\frac{n}{2}\right)+1$ ..... (Equation 1)

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right)$ = $T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: T (n) = ($T\left(\frac{n}{2^2}\right) + 1$) +1.....By putting T $\frac{n}{2}$ in (1) equation

T (n) = $T\left(\frac{n}{2^2}\right) + 2$..... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$..... Putting $\frac{n}{2}$ in place of n in eq 1.

T (n) = $T\left(\frac{n}{2^3}\right) + 1 + 2$

T (n) =$T\left(\frac{n}{2^3}\right) + 3$..... (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$..... Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

T (n) = $T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

T (n) = $T\left(\frac{n}{2^i}\right) + i$ .....

Stopping Condition: T (1) =1

At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why

T (1) =1 (Item, Comparison)

Is the last term of the equation and it will be equal to 1

$\frac{n}{2^i} = 1$

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

$n = 2^i$

Applying log both sides

$\log n = \log_2 i$

Log n = i log 2

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

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

$\frac{n}{2^i}$ = 1 as in eq 5

= T (1) + i

= 1 + i..... T (1) =1 by stopping condition

= 1+ $\log_2 n$

= $\log_2 n$ ..... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Merge Sort

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array **A**, such that our main concern will be to sort the subsection, which starts at index **p** and ends at index **r**, represented by **A[p..r]**.

Divide

If assumed **q** to be the central point somewhere in between **p** and **r**, then we will fragment the subarray **A[p..r]** into two arrays **A[p..q]** and **A[q+1, r]**.

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays **A[p..q]** and **A[q+1, r]**. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays **A[p..q]** and **A[q+1, r]**, after which we merge them back to form a new sorted array **[p..r]**.

Merge Sort algorithm

The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

1. ALGORITHM-MERGE SORT
2. 1. If p**<r**
3. 2. Then q → (p+ r)/2
4. 3. MERGE-SORT (A, p, q)
5. 4. MERGE-SORT (A, q+1,r)
6. 5. MERGE (A, p, q, r)

Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

The following figure illustrates the dividing (splitting) procedure.

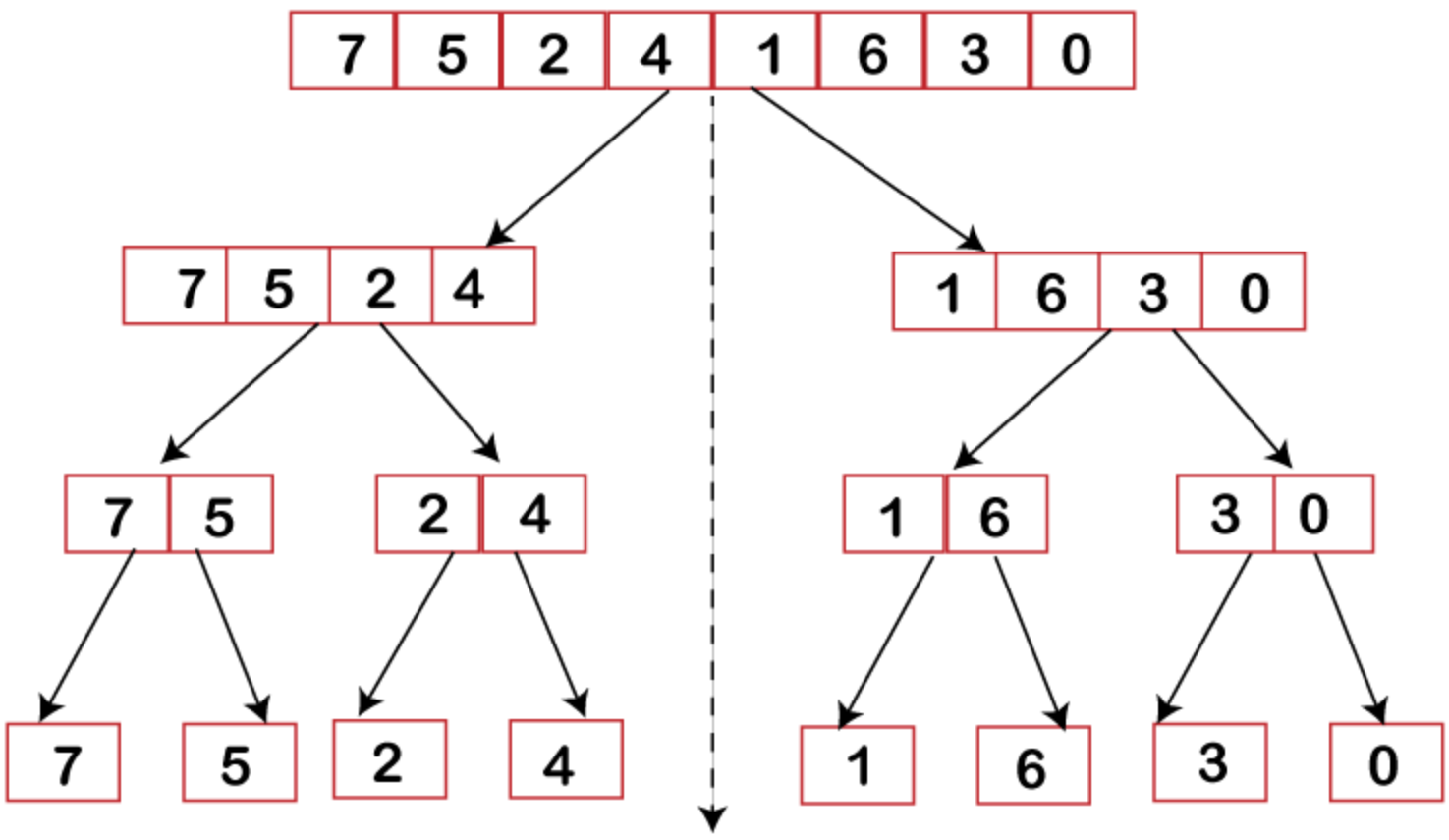


Figure 1: Merge Sort Divide Phase

1. FUNCTIONS: MERGE (A, p, q, r)
2.
3. 1. n 1 = q-p+1
4. 2. n 2= r-q
5. 3. create arrays [1.....n 1 + 1] and R [1.....n 2 +1]
6. 4. for i ← 1 to n 1
7. 5. do [i] ← A [p+ i-1]
8. 6. for j ← 1 to n2
9. 7. do R[j] ← A[q + j]
10. 8. L [n 1+ 1] ← ∞
11. 9. R[n 2+ 1] ← ∞
12. 10. I ← 1
13. 11. J ← 1
14. 12. For k ← p to r
15. 13. Do if L [i] ≤ R[j]
16. 14. then A[k] ← L[i]
17. 15. i ← i +1
18. 16. else A[k] ← R[j]
19. 17. j ← j+1

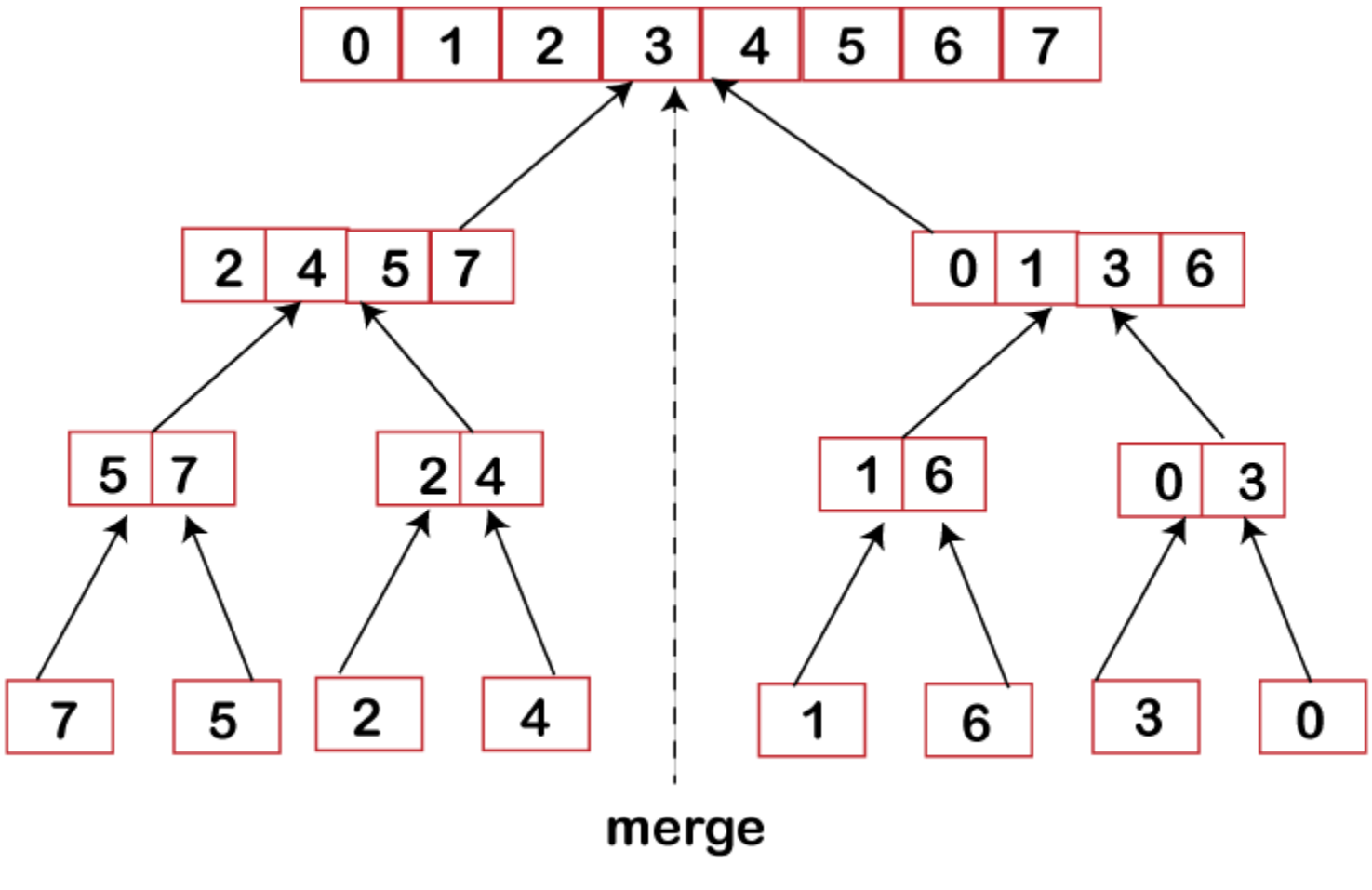


Figure 2: Merge Sort Combine Phase

The merge step of Merge Sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

1. Did you reach the end of the array?
2. No:
3. Firstly, start with comparing the current elements of both the arrays.
4. Next, copy the smaller element into the sorted array.
5. Lastly, move the pointer of the element containing a smaller element.
6. Yes:
7. Simply copy the rest of the elements of the non-empty array

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

A= (36,25,40,2,7,80,15)

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

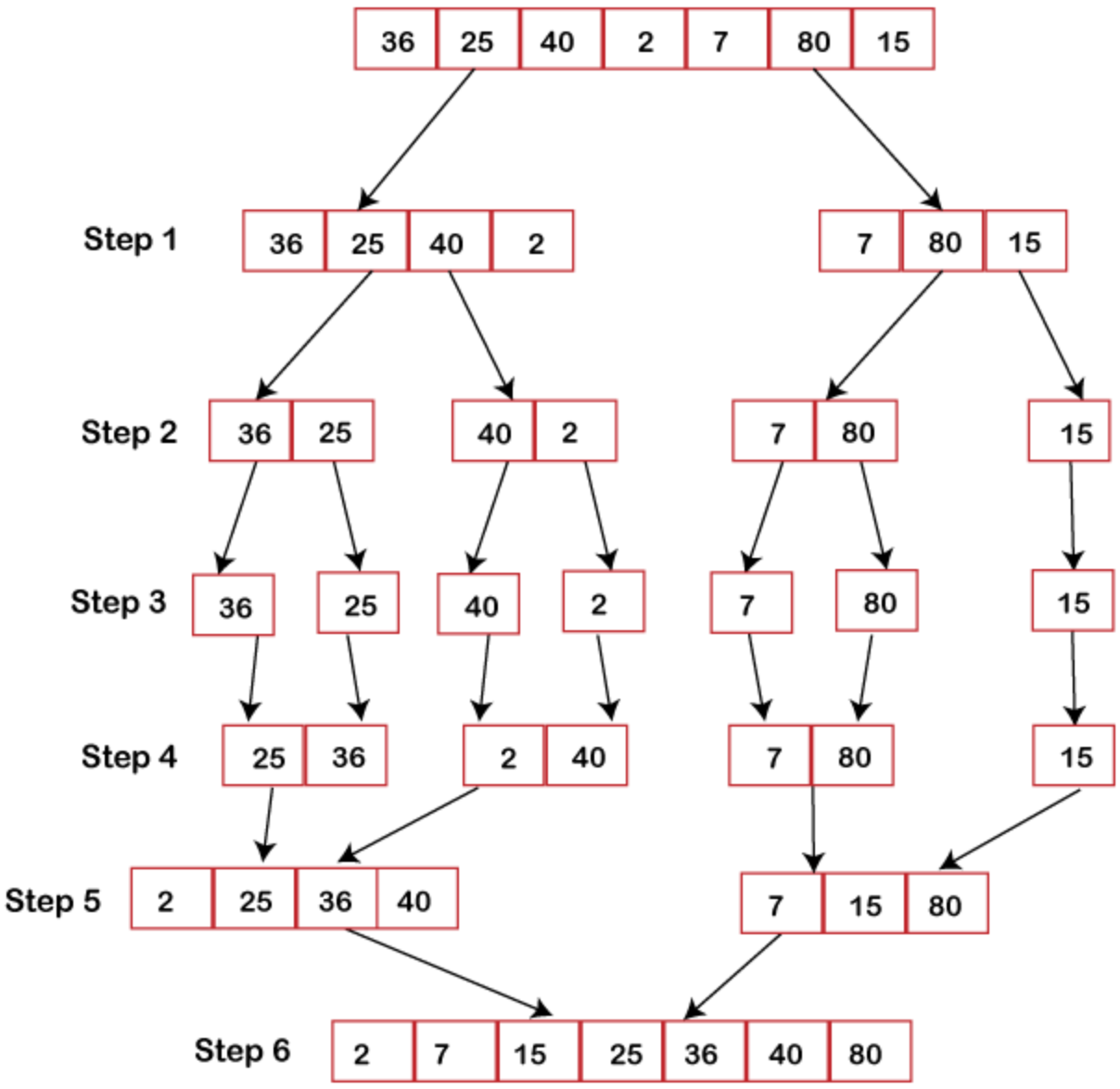
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.

Step5: For each list, we will first compare the element and then combine them to form a new sorted list.

Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



Hence the array is sorted.

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

> ***Note: Stopping Condition T (1) =0 because at last, there will be only 1 element left that need to be copied, and there will be no comparison.***

Putting $n=\frac{n}{2}$ in place of n in .....equation 1

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$.....equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

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

$$T(n) = 2^2\,T\left(\frac{n}{2^2}\right) + 2n$$.....equation 3

Putting $n = \frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$.....equation4

Putting 4 equation in 3 equation

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

$$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + n + 2n$$

$$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + 3n$$.....equation5

From eq 1, eq3, eq 5.....we get

$$T(n) = 2^i\,T\left(\frac{n}{2^i}\right) + in$$.....equation6

From Stopping Condition:

$$\frac{n}{2^i} = 1 \text{ And } T\left(\frac{n}{2^i}\right)=0$$

$$n = 2^i$$

Apply log both sides:

log $n=\log_2 i$

$\log n = i \log 2$

$\frac{\log n}{\log 2} = i$

$\log_2 n = i$

From 6 equation

$$T(n) = 2^i\,T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \cdot n$$

$$= T(n) = n.\log n$$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

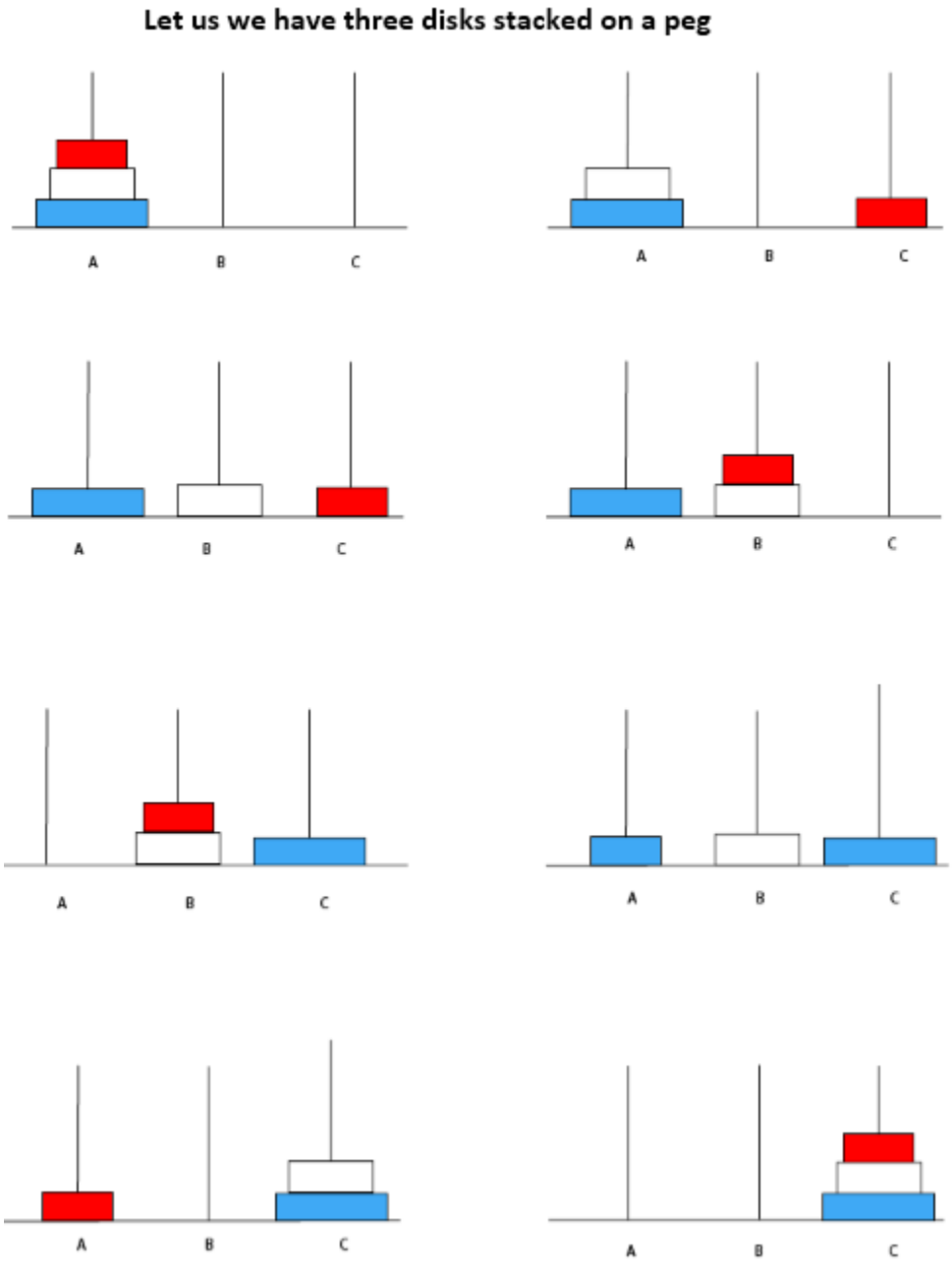
Tower of Hanoi

1. It is a classic problem where you try to move all the disks from one peg to another peg using only three pegs.

2. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks.

3. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time.

This problem can be easily solved by Divide & Conquer algorithm



In the above 7 step all the disks from peg A will be transferred to C given Condition:

1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk.

Let T (n) be the total time taken to move n disks from peg A to peg C

1. Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
2. Moving larger disks from the first peg to the third peg will require first one step.
3. Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

So, total time taken T (n) = T (n-1)+1+ T(n-1)

Relation formula for Tower of Hanoi is:

T (n) = 2T (n-1) +1

Note: Stopping Condition: T (1) =1

Because at last there will be one disk which will have to move from one peg to another.

T (n) = 2T (n-1) +1.....eq1

Put n = n-1 in eq 1

T (n-1) = 2T (n-2) +1.....eq2

Putting 2 eq in 1 eq

T (n) = 2(2T (n-2) +1) +1

T (n) = 2^2T (n-2) +2+1.....eq3

Put n = n-2 in eq 1

T (n-2) = 2T (n-3) +1.....eq4

Putting 4 eq in 3 eq

$T(n) = 2^2[2T(n-3)+1] + 2 + 1$

$T(n) = 2^3T(n-3) + 2^2 + 2 + 1$.....eq5

From 1 eq, 3 eq, 5 eq

We get,

$$T(n) = 2^iT(n-i) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots + 2^0$$

Now n-i=1 from stopping condition

And T (n-i) =1

n-1 =i ← A equation

Now, $T(n) = 2^i(1) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots + 2^0$

It is a Geometric Progression Series with common ratio, r=2
First term, a=1(2^0)

Sum of n terms in G.P= $S_n = \frac{a(1-r^n)}{1-r}$

So $T(n) = \frac{1(1-2^{i+1})}{(1-2)}$

Because exponents are from 0 to 1

n=i+1

$T(n) = \frac{2^{i+1}-1}{2-1} = 2^{i+1} - 1$

$T(n) = 2^{i+1} - 1$

From A

$$T(n) = 2^{n-1+1} - 1$$

$T(n) = 2^n - 1$.....B Equation

B equation is the required complexity of technique tower of Hanoi when we have to move n disks from one peg to another.

T (3) = 2^3- 1
= 8 - 1 = **7 Ans**

[As in concept we have proved that there will be 7 steps now proved by general equation]

Program of Tower of Hanoi:

```c
#include<stdio.h>
void towers(int, char, char, char);
 int main()
 {
      int num;
      printf ("Enter the number of disks : ");
       scanf ("%d", &num);
     printf ("The sequence of moves involved in the Tower of Hanoi are :\n");
     towers (num, 'A', 'C', 'B');
     return 0;

}
   void towers( int num, char from peg, char topeg, char auxpeg)
{
        if (num == 1)
{
     printf ("\n Move disk 1 from peg %c to peg %c", from peg, topeg);
        return;
 }
  Towers (num - 1, from peg, auxpeg, topeg);
   Printf ("\n Move disk %d from peg %c to peg %c", num, from peg, topeg);
  Towers (num - 1, auxpeg, topeg, from peg);
 }
```

Output:

```
Enter the number of disks: 3
The sequence of moves involved in the Tower of Hanoi is
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg
```

METHOD NAME	EQUATION	STOPPING CONDITION	COMPLEXITIES
1.Max&Min	$T(n) = 2T\left(\frac{n}{2}\right)+2$	T(2)=1	$T(n) = \frac{3N}{2} - 2$
2.Binary Search	$T(n) = T\left(\frac{n}{2}\right) + 1$	T(1)=1	T(n)=log n
3.Merge Sort	$T(n)=2T\left(\frac{n}{2}\right)+n$	T(1)=0	T(n)=nlog n
4. Tower of Hanoi	T(n)=2T(n-1)+1	T(1)=1	$T(n)=2^n-1$