PW SKILLS

Lesson Plan

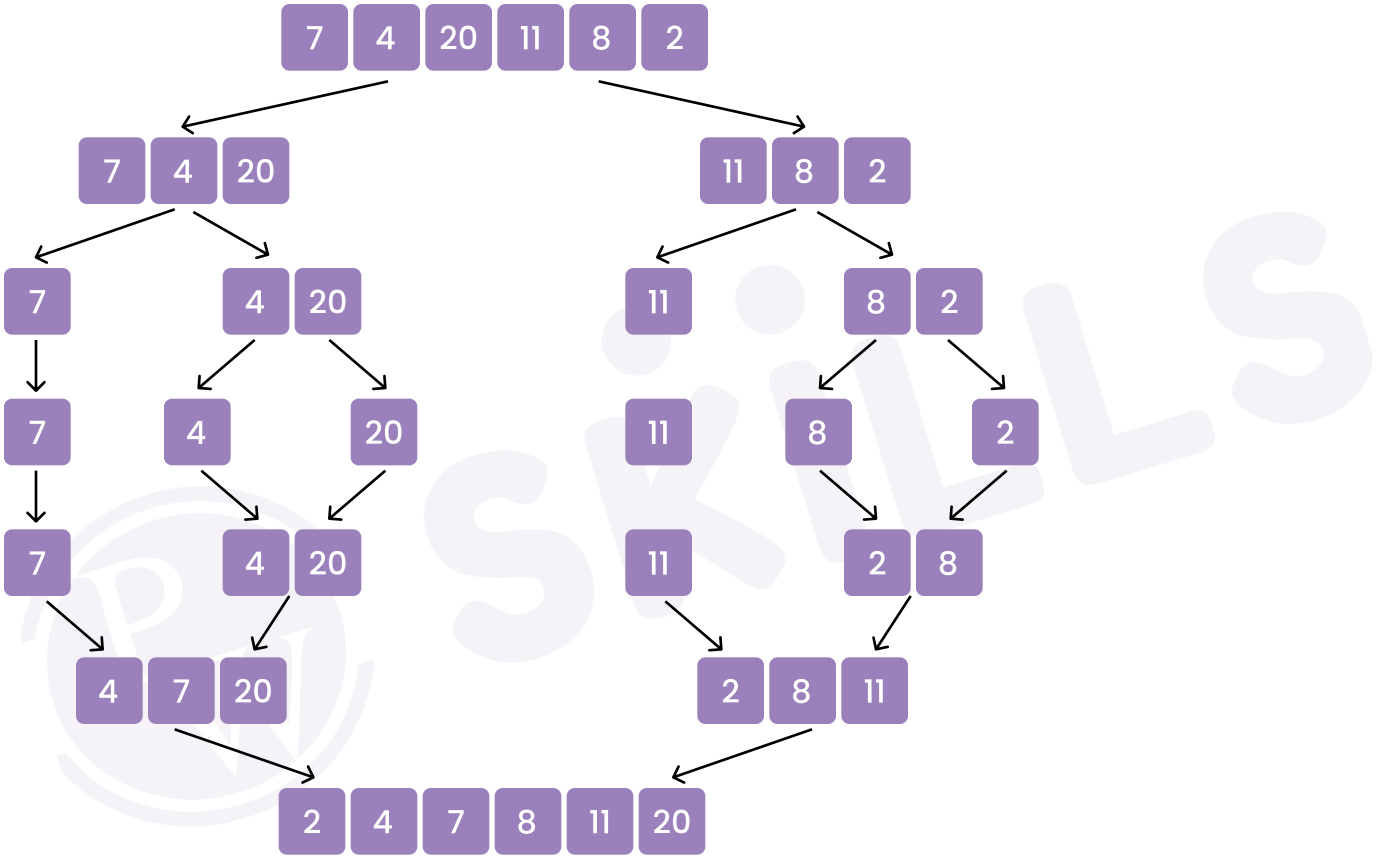
Merge Sort Algorithm

PW SKILLS

This algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are merged in a sorted manner.. We have to define the **merge()** function to perform the merging.

The sub-arrays are divided again and again into halves until the array cannot be divided further. Then we combine the pair of one element array into two-element array, sorting them in the process. The sorted two-element pairs are merged into the four-element array, and so on until we get the sorted array.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are merged to form the final solution.



Now, let's see the algorithm of merge sort.

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

According to the merge sort algorithm, the first array is divided into two equal halves. Merge sort keeps dividing the array into equal parts until it cannot be further divided.

As there are eight elements in the given array, it is divided into two arrays of size 4.

Now, again divide these two arrays into halves. As they are of size 4, divide them into new arrays of size 2.

Now, again divide these arrays to get the atomic value that cannot be further divided.

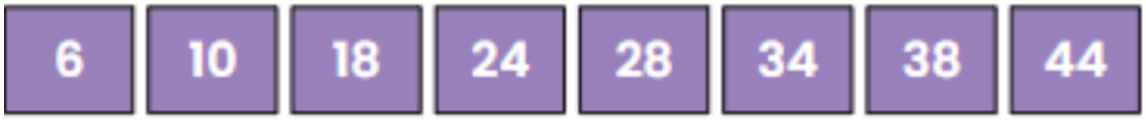
Now, combine them in the same manner they were broken.

In combining, first compare the elements of each array and then combine them into another array in sorted order.

So, first compare 10 and 28, both are in sorted positions. Then compare 24 and 6, and in the list of two values, put 6 first followed by 24. Then compare 34 and 18, sort them and put 18 first followed by 34. After that, compare 38 and 44, and place them sequentially.

In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Algorithm:

Step 1: start

Step 2: declare an array and left, right, mid variable

Step 3: perform merge function.

```
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
```

Step 4: Stop

Follow the steps below the solve the problem:

MergeSort(arr[], l, r)

If r > l

- Find the middle point to divide the array into two halves:
  - middle mid = l + (r – l)/2
- Call mergeSort for first half:
  - Call mergeSort(arr, l, mid)
- Call mergeSort for second half:
  - Call mergeSort(arr, mid + 1, r)
- Merge the two halves sorted in steps 2 and 3:
  - Call merge(arr, l, mid, r)

Below is the implementation of the above approach:

Link to code: https://pastebin.com/zkb194qh

```
Before sorting array elements are -
10 28 24 6 34 18 38 44
After sorting array elements are -
6 10 18 24 28 34 38 44

...Program finished with exit code 0
Press ENTER to exit console.
```

Topic - Merge Sort Time Complexity:

Now, let us calculate time complexity with the steps. our very own first step was to divide the input into two halves which comprised us of a logarithmic time complexity ie. log(N) where N is the number of elements in the array.

Our second step was to merge back the array into a single array, so if we observe it in all the number of elements to be merged N, and to merge back we use a simple loop which runs over all the N elements giving a time complexity of O(N).

finally, total time complexity will be - step -1 + step-2
$T(n) = 2T(n/2) + O(n)$
The solution of the above recurrence can be written as O(nLogn).

The array of size N is divided into a max of Logn parts, and the merging of all subarrays into a single array takes O(N) time, the worst-case run time of this algorithm is O(nLogn).

The time complexity of MergeSort is O(n*Log n) in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

Topic - Merge Sort Space Complexity:

Let us take this example again.

Once we call merge sort for the entire array (N = 6), the array is divided into two parts, (each of size N / 2 = 3).

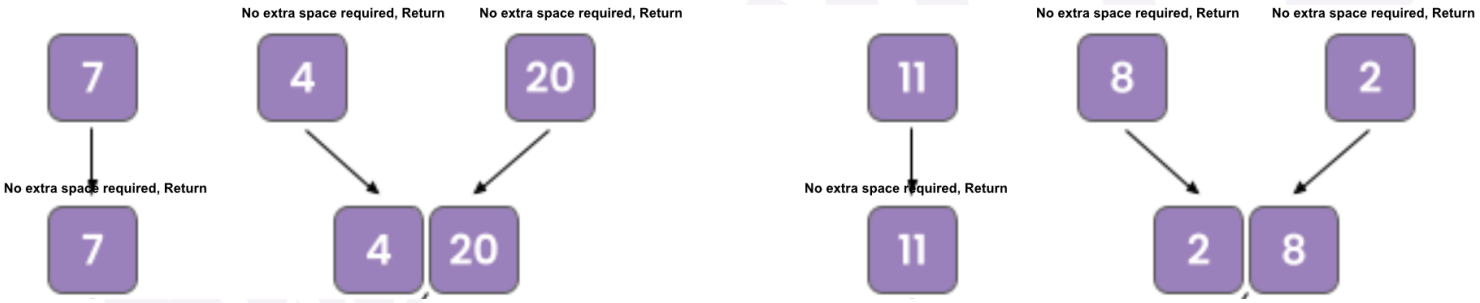
However, we must note that function calls are **not** running in parallel. Everytime during the **divide phase**, we are making a single function call, first with the left part and then waiting for its return (with sorted array in place) to call for the right part.

The same happens for the next function calls, till we get a single element, in which case we return.

So, the first time we encounter a single element in a function call, it would be as follows:

At any point, in the call stack, we would be having a maximum of **O(logN)** function calls.

Now, once we are returned from the left and the right call, the **merge phase** for that function call would begin. So, if the size of the subarray for that function call is N with its left and right part of size N/2, we would be creating a new array of size N to do the merge. So, to merge the arrays: The extra space that we will be needing is as follows:



Also note that once we return from any function, the extra space we take to merge the two sorted subarrays is also freed. So, we need not sum up the extra space taken in each function call since maximum space taken at any point is something we are concerned about.

Hence, during the divide phase, the maximum space we take in the recursion call stack is **O(logN)** and while merging, out of all the function calls, the maximum space we take is **O(N)** when we merge two subarrays of the **entire array**. Since **O(N)** is dominating, we would consider **O(N)** to be the space complexity of the merge sort.

Apart from the intuition, the space complexity can be found as:

Let S(N) denote the amount of extra space required for a function sorting array of size N. We need to calculate the recursion call stack space and the extra space we use for merging at each function call. In general terms for sorting an array of size N:

S(1) = O(logN) (recursive stack space)
S(2) = O(logN - 1 (recursive stack space) + 2 (Number of elements))
S(4) = O(logN - 2 (recursive stack space) + 4 (Number of elements))
S(8) = O(logN - 3 (recursive stack space) + 8 (Number of elements))

So, for sorting an array of N elements, S(N) = O(N (Number of elements)), since after returning from left and right, the recursion stack would contain just a single function call, which would be the function call for the original array.

Topic - Is Merge Sort Stable?

Yes, Merge Sort is a stable sorting algorithm which means that the same element in an array maintains their original positions with respect to each other.

Merge sort is not in place because **it requires additional memory space to store the auxiliary arrays.**

Topic : Applications of merge sort

- More efficient and works fast in case of large Data sets.
- It is the best Sorting technique used for sorting Linked Lists.(will be explained in coming lectures)

Topic : Drawbacks of Merge Sort:

- Slower compared to the other sort algorithms for smaller tasks.
- This algorithm requires an additional memory space of 0(n) for the temporary array.
- It goes through the whole process even if the array is sorted.

Q1. Given an array of N integers, count the inversion of the array (using merge-sort).

What is an inversion of an array? Definition: for all i & j < size of array, if i < j then you have to find a pair (A[i],A[j]) such that A[j] < A[i].

Solution:

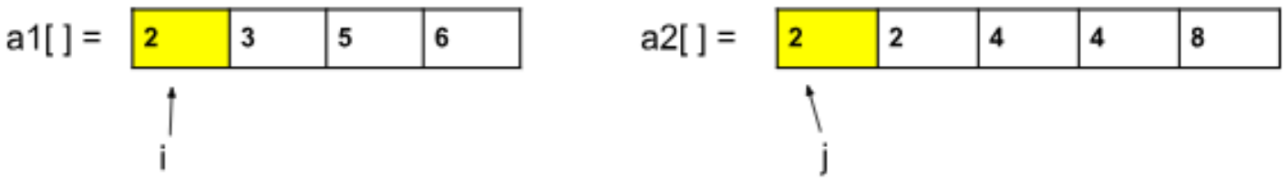
We are required to give the total number of inversions and the inversions are: For i & j < size of an array if i < j then you have to find pair (a[i], a[j]) such that a[i] > a[j].

For example, for the given array: [5,3,2,1,4], (5, 3) will be a valid pair as 5 > 3 andindex 0 < index 1. But (1, 4) cannot be valid pair.

Assume two sorted arrays are given i.e. a1[] = {2, 3, 5, 6} and a2[] = {2, 2, 4, 4, 8}. Now, we have to count the pairs i.e. a1[i] and a2[j] such that a1[i] > a2[j].

In order to solve this, we will keep two pointers i and j, where i will point to the first index of a1[] and j will point to the first index of a2[]. Now in each iteration, we will do the following:

- **If a1[i] <= a2[j]:** These two elements cannot be a pair and so we will move the pointer i to the next position. This case is illustrated below:

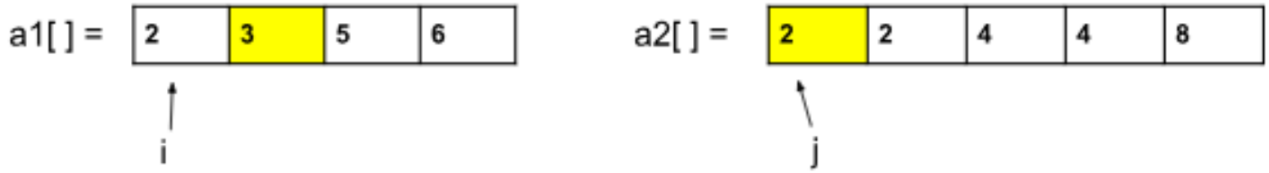


Here, a1[i] == a2[j], so we will move the i pointer to next position.

- **Why we moved the i pointer:** We know, that the given arrays are sorted. So, all the elements after the pointer j, should be greater than a2[j]. Now, as a1[i] is smaller or equal to a2[j], it is obvious that a1[i] will be smaller or equal to all the elements after a2[j]. We need a bigger value of a1[i] to make a pair and so we move the i pointer to the next position i.e. next bigger value.

- **If a1[i] > a2[j]:** These two elements can be a pair and so we will update the count of pairs. Now, here, we should observe that as a1[i] is greater than a2[j], all the elements after a1[i] will also be greater than a2[j] and so, those elements will also make pair with a2[j]. So, the number of pairs added will be n1-i (where n1 = size of a1[ ]).

Now, we will move the j pointer to the next position. This case is also illustrated below:



Here, a1[i] > a2[j], and elements after a1[i] i.e. 5 and 6 is also greater than a2[j]. So, the total number of pairs added to count will be n1-i = 4-1 = 3.
Therefore, cnt = cnt + 3
Now, we will move the j pointer to the next position.

The above process will continue until at least one of the pointers reaches the end.

Until now, we have figured out how to count the number of pairs in one go if two sorted arrays are given. But in our actual question, only a single unsorted array is given. So, how to break it into two sorted halves so that we can apply the above observation?

We can think of the merge sort algorithm that works in a similar way we want. In the merge sort algorithm, at every step, we divide the given array into two halves and then sort them, and while doing that we can actually count the number of pairs.

Basically, we will use the merge sort algorithm to use the observation in the correct way.

Approach:

The steps are basically the same as they are in the case of the merge sort algorithm. The change will be just a one-line addition inside the **merge()** function. Inside the merge(), we need to add the number of pairs to the count when a[left] > a[right].

The steps of the merge() function were the following:

1. In the merge function, we will use a temp array to store the elements of the two sorted arrays after merging. Here, the range of the left array is low to mid and the range for the right half is mid+1 to high.

2. Now we will take two pointers left and right, where left starts from low and right starts from mid+1.

3. Using a while loop(while(left <= mid && right <= high)), we will select two elements, one from each half, and will consider the smallest one among the two. Then, we will insert the smallest element in the temp array.

4. After that, the left-out elements in both halves will be copied as it is into the temp array.

5. Now, we will just transfer the elements of the temp array to the range low to high in the original array.

Modifications in merge() and mergeSort():

- In order to count the number of pairs, we will keep a count variable, cnt, initialized to 0 beforehand inside the merge().

- While comparing a[left] and a[right] in the 3rd step of merge(), if a[left] > a[right], we will simply add this line:cnt += mid-left+1 (mid+1 = size of the left half)

- Now, we will return this cnt from merge() to mergeSort().

- Inside mergeSort(), we will keep another counter variable that will store the final answer. With this cnt, we will add the answer returned from mergeSort() of the left half, mergeSort() of the right half, and merge().

- Finally, we will return this cnt, as our answer from mergeSort().

Code:

```
import java.util.*;

public class InversionCount {
    public static int merge(ArrayList<Integer> arr, int low,
int mid, int high) {
        ArrayList<Integer> temp = new ArrayList<>();
        int left = low;
        int right = mid + 1;
        int cnt = 0;

        while (left ⩽ mid && right ⩽ high) {
```

```
            if (arr.get(left) ⩽ arr.get(right)) {
                temp.add(arr.get(left));
                left++;
            } else {
                temp.add(arr.get(right));
                cnt += (mid - left + 1);
                right++;
            }
        }

        while (left ⩽ mid) {
            temp.add(arr.get(left));
            left++;
        }

        while (right ⩽ high) {
            temp.add(arr.get(right));
            right++;
        }

        for (int i = low; i ⩽ high; i++) {
            arr.set(i, temp.get(i - low));
        }

        return cnt;
    }

    public static int mergeSort(ArrayList<Integer> arr, int
low, int high) {
        int cnt = 0;
        if (low ⩾ high) return cnt;
        int mid = (low + high) / 2;
        cnt += mergeSort(arr, low, mid);
        cnt += mergeSort(arr, mid + 1, high);
        cnt += merge(arr, low, mid, high);
        return cnt;
    }

    public static int numberOfInversions(ArrayList<Integer>
a, int n) {
        return mergeSort(a, 0, n - 1);
    }

    public static void main(String[] args) {
        ArrayList<Integer> a = new
ArrayList<>(Arrays.asList(5, 4, 3, 2, 1));
        int n = 5;
        int cnt = numberOfInversions(a, n);
        System.out.println("The number of inversions are: "
+ cnt);
    }
}
```

Output: The number of inversions is: 10

Complexity Analysis

Time Complexity: O(N*logN), where N = size of the given array.

Reason: We are not changing the merge sort algorithm except by adding a variable to it. So, the time complexity is as same as the merge sort.

Space Complexity: O(N), as in the merge sort We use a temporary array to store elements in sorted order.

PW SKILLS

THANK YOU !

PW SKILLS