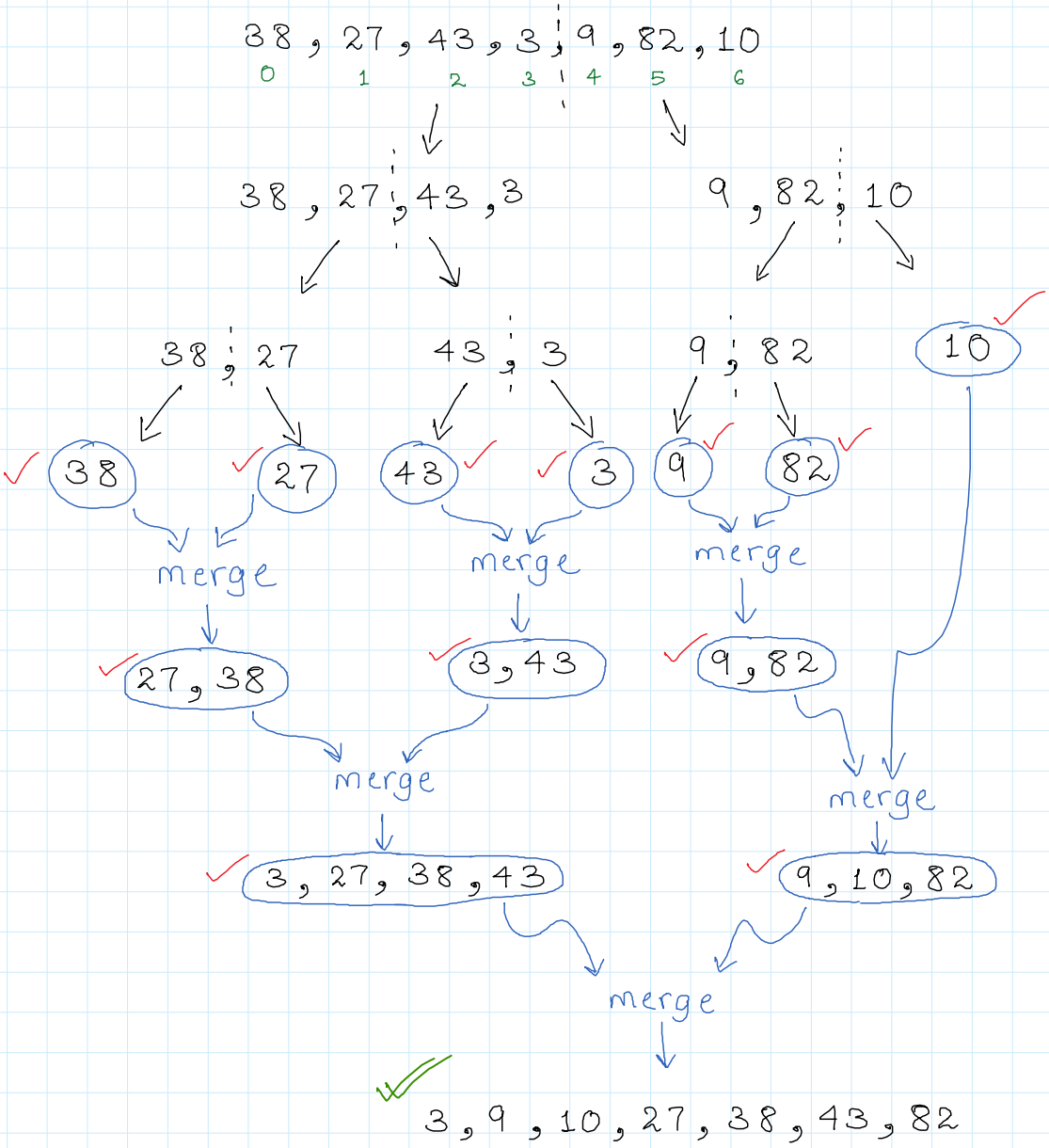
Merge Sort is a sorting algorithm that uses the concept of divide and conquer.

Eg:



✓ - Sorted Array.

Visualization (Source: Hacker Earth)

Approach 1 :

Logic - We will divide / split the array into 2 parts, call the function for the 2 parts recursively (with the

hope that this will return two sorted sub-parts).

We then merge the 2 sorted arrays and return the new sorted array.

Our base case is when we have 1 element in the array to be sorted, i.e. start >= end, we should return without doing anything.

Process : arr : 38 , 27 , 43 , 3 , 9 , 82 , 10
0 1 2 3 4 5 6

mergeSort(arr, 0, 6) is called for the entire array. This checks for the base case, (6-0+1) = length of the array = 7 ≠ 1, thus, base condition not satisfied.

We then call mergeSort(arr, 0, 3) & mergeSort(arr, 4, 6) with the faith that our 2 halves will get sorted.

We will then merge these 2 sorted arrays using merge(arr, 0, 6).

Code :

mergeSort function.

```
void mergeSort(int *arr, int s, int e) {

    //base case
    if(s >= e) {
        return;
    }

    int mid = (s+e)/2;

    //left part sort karna h
    mergeSort(arr, s, mid);

    //right part sort karna h
    mergeSort(arr, mid+1, e);

    //merge
    merge(arr, s, e);
}
```

helper merge function

```
void merge(int *arr, int s, int e) {

    int mid = (s+e)/2, len1 = mid - s + 1, len2 = e - mid;
    int *first = new int[len1];
    int *second = new int[len2];
    //copy values
    int mainArrayIndex = s;
    for(int i=0; i<len1; i++) {
        first[i] = arr[mainArrayIndex++];
    }

    mainArrayIndex = mid+1;
    for(int i=0; i<len2; i++) {
        second[i] = arr[mainArrayIndex++];
    }
    //merge 2 sorted arrays
    int index1 = 0;
    int index2 = 0;
    mainArrayIndex = s;
    while(index1 < len1 && index2 < len2) {
        if(first[index1] < second[index2]) {
            arr[mainArrayIndex++] = first[index1++];
        }
        else{
            arr[mainArrayIndex++] = second[index2++];
        }
    }
    while(index1 < len1) {
        arr[mainArrayIndex++] = first[index1++];
    }

    while(index2 < len2 ) {
```

```
        arr[mainArrayIndex++] = first[index1++];
    }

    while(index2 < len2 ) {
        arr[mainArrayIndex++] = second[index2++];
    }
    delete []first;
    delete []second;
}
```

Time Complexity : O(nlogn) → O(n) for every O(logn) partitions.

Space Complexity : O(n)

→ We are using 2 auxilliary arrays to merge the 2 sorted arrays.

Homework : Count inversions in an array.

Explaination - An inversion is when arr[i] > arr[j] for some i < j.

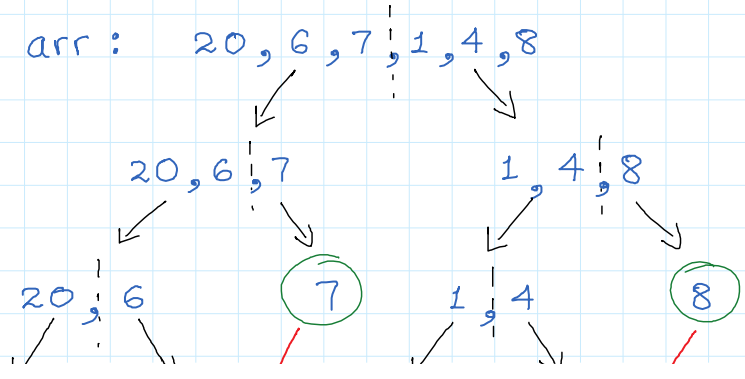
Example : 8 3 6 4 2

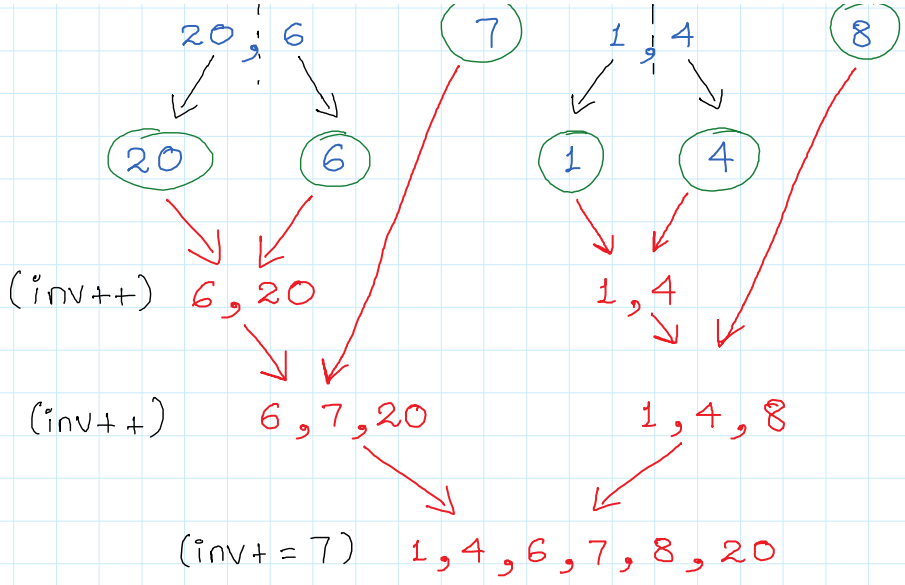
Inversions — ① 8 — 3 ② 8 — 6 ③ 8 — 4 ④ 8 - 2 ⑤ 3 — 2 ⑥ 6 - 4 ⑦ 6 — 2 ⑧ 4 — 2

Answer = 8 (Elements occuring before, which are greater than element(s) after them.)

Logic : When we use the Merge Sort's partitioning approach, we merge the sorted arrays. We use this merging to count the number of inversions in the array.

Usecase : arr : 20, 6, 7, 1, 4, 8





answer = 9

For two sorted arrays | 6 | 7 | 20 | , | 1 | 4 | 8 |

i ↑ ↑ j ↑

① 6 > 1, so all elements in arr1 > 1. (arr1 is sorted)

⟹ inversion += 3 (length of arr1)

② 6 > 4, so again, all elements in arr1 > 4. (arr1 is sorted)

⟹ inversion += 3

③ 6 < 8, increment i. 7 < 8, increment i.

20 > 8, inversion += 1.

Logic is to fix j for all i and count the number of inversions. Repeat for all j.

Code:

inversionCount:

```
int inversionCount(vector<int>& nums, int l, int r) {
    if(l >= r) return 0;
    vector<int> temp(r - l + 1);
    int mid = l + (r - l)/2, inv = 0;

    inv += inversionCount(nums, 0, mid);
    inv += inversionCount(nums, mid+1, r);
    inv += merge(nums, l, mid, r, temp);

    return inv;

}
```

Counts the inversions for the split arrays and then for the merged arrays.

merge function:

counts the inversions while merging the 2 sub-arrays.

```cpp
int merge(vector<int>& nums, int l, int mid, int r, vector<int>& temp) {
    int i = l, j = mid+1, k = l;
    int inv = 0;
    while(i <= mid && j <= r) {
        if(nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        }
        else {
            temp[k++] = nums[j++];
            // all elements from nums[i] to nums[mid] will be greater than nums
            [j]
            inv += (mid - i + 1);
        }
    }
    while(i <= mid) {
        temp[k++] = nums[i++];
    }
    while(j <= r) {
        temp[k++] = nums[j++];
    }

    for(i=l;i<=r;i++) {
        nums[i] = temp[i];
    }
    return inv;
}
```