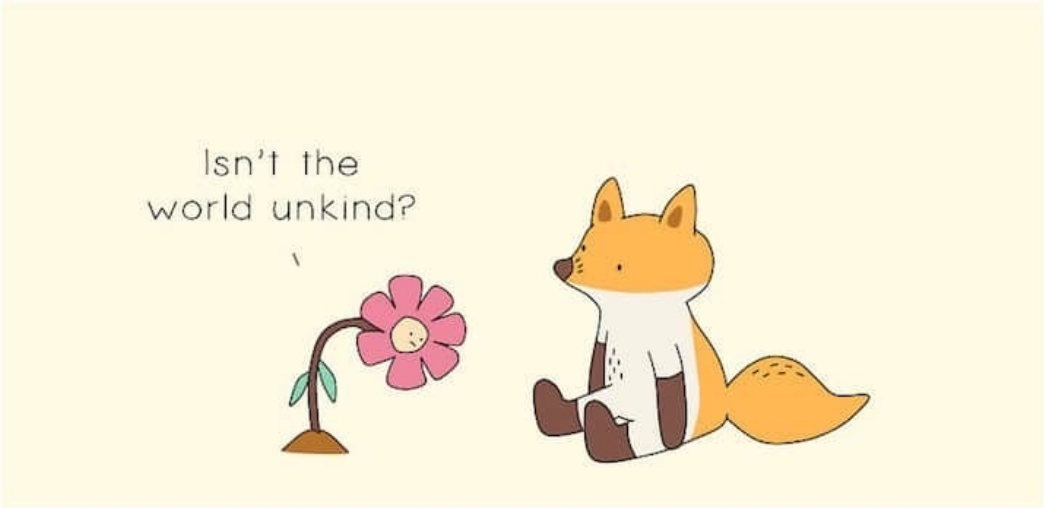
Sorting - 1

1. Smallest Number
2. Merge 2 sorted arrays
3. Merge Sort
4. Stable Sort & Inplace

Notes



Count Sort

< *Question* > : Find the smallest number that can be formed by rearranging the digits of the given number in an array. Return the smallest number.

arr[ ] → [6 3 4 2 7 2 1]

1 2 2 3 4 6 7

arr[ ] → [4 2 7 3 9 0]

0 2 3 4 7 9

Idea -1 Use inbuilt sorting, sort the array in Ascending order.

T.C : O(N log N)

S.C : O(1)

Observations : No in the array are digits.

Range : 0 to 9

Idea -2

Smallest no will be of form.

0000...0 111...1 222...1 ... 999...9

freq of 0 freq of 1

freq[10]

freq[i] → frequency of ith element

arr[ ] → [9 1 2 5 4 2 1 2 5 8]

farr →

0	1	2	3	4	5	6	7	8	9
0	2	3	0	1	2	0	0	1	1

o/p : 1 1 2 2 2 4 5 5 8 9

< / > Code

```
void CountSort(int[] arr) {
    int freq[] = new int[10];

    // Update freq of elements
    for (i=0; i<n; i++) {
        int val = arr[i];
        freq[val]++;
    }

    // Iterate on freq array to get ans
    for (i=0; i<10; i++) {
        int cnt = freq[i];
        // Print each element "cnt" times
        for (j=0; j<cnt; j++) {
            print(i);
        }
    }
}
```

N iterations

10 * N iterations

T.C : O(N)
S.C : O(1)

Will count sort work on large values? No

$[2, 38, 10^6, 10^9]$

Range of no : $[0, 10^9]$ ⟶ freq $[10^9+1]$

1 int : 4 bytes

10^6 int : 4×10^6 bytes = 4MB

10^6 int : 4×10^9 bytes = 4GB Not practical

Count Sort can work if no are in range $\sim 10^6$

Can count sort work for range bound negative numbers? Yes.

-3	2	2	1	-4	5	-3

o/p :

-4	-3	-3	1	2	2	5

Min No : -4

Max No : 5

Range of digits :

$\Rightarrow 5-(-4)+1 \Rightarrow 10$

arr[]:	-3	2	2	1	-4	5	-3
idx:	1	6	6	5	0	9	1

(arr[i] + |-4|)

Digits : -4 to 5 —A[i] + |Min|→ Idx : 0 to 9

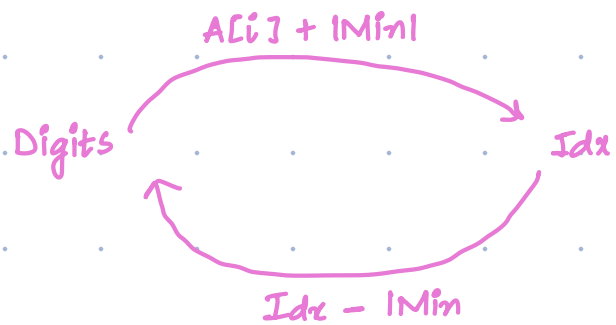
Min = -4

Max = 5

-4	-3	-2	-1	0	1	2	3	4	5
0	1	2	3	4	5	6	7	8	9
1	2	0	0	0	1	2	0	0	1

idx - |Min|

Olp: -4 -3 -3 1 2 2 5



< / > Code

```
void CountSort(int[] arr) {

    // Find Min & Max of arr.

    int size = Max - Min + 1;

    int[] freq = new int[size];

    // Update freq of elements

    for(i = 0; i < arr.length; i++) {

        int val = arr[i];

        int idx = val + abs(Min);

        freq[idx]++;
    }

    // Print each element

    for(i = 0; i < size; i++) {

        int cnt = freq[i];

        int digit = i - abs(Min);

        for(j = 0; j < cnt; j++) {

            print(digit);
        }
    }
}
```

T.C. : O(N)

S.C. : O(Range of No)

Merge Two Sorted Arrays?

a[ ] → [2 4 7 8 12] → N

($-10^9 \le$ element $\le 10^9$)

b[ ] → [3 5 6 7] → M

farr →

2	3	4	5	6	7	7	8	12
0	1	2	3	4	5	6	7	8

Idea -1

Create a new array of size (N+M). Fill all elements of arr1 & arr2 in new array & then sort the new array.

T.C : (N+M) log (N+M)

Idea -2

Using 2 ptrs

a[ ] → [2 4 7 8 12] (p_1 points to 8)

b[ ] → [3 5 6 7] (p_2 points to 7)

farr →

2	3	4	5	6	7	7	8	12
0	1	2	3	4	5	6	7	8

(k points to 8)

< / > Code

```
int[] MergeArrays(int[] A, int[] B){
    int n = A.size();
    int m = B.size();
    int P1 = 0, P2 = 0, K = 0;
    int[] C = new int[n+m];
    while(P1 < n && P2 < m){
        // Case 1:
        if(A[P1] < B[P2]){
            C[K] = A[P1];
            K++;
            P1++;
        }
        else {
            C[K] = B[P2];
            K++;
            P2++;
        }
    }
    while(P1 < n){
        C[K] = A[P1];
        K++;
        P1++;
    }
    while(P2 < m){
        C[K] = B[P2];
        K++;
        P2++;
    }
    return C;
}
```

T.C : O(N+M)
S.C : O(1)

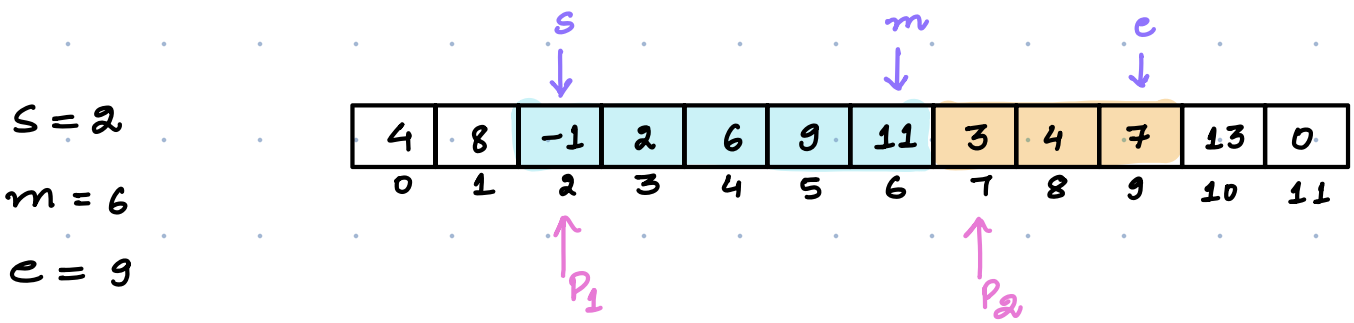
Merge 2 Sorted Subarrays

Given array of N elements and three indices s, m, e.

Subarray [s m] is sorted.

Subarray [m+1 e] is sorted.

Sort the entire subarray from [s e].

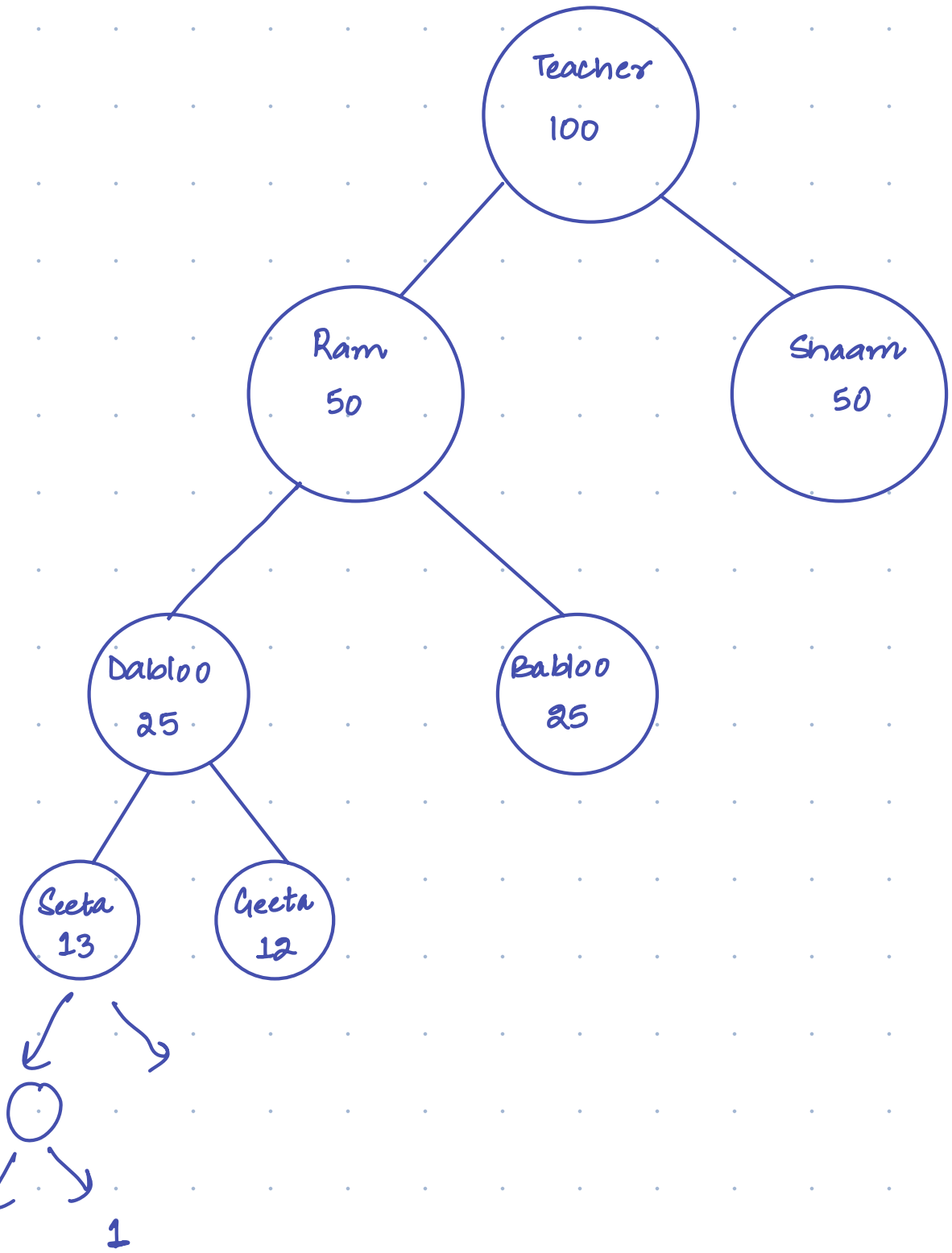


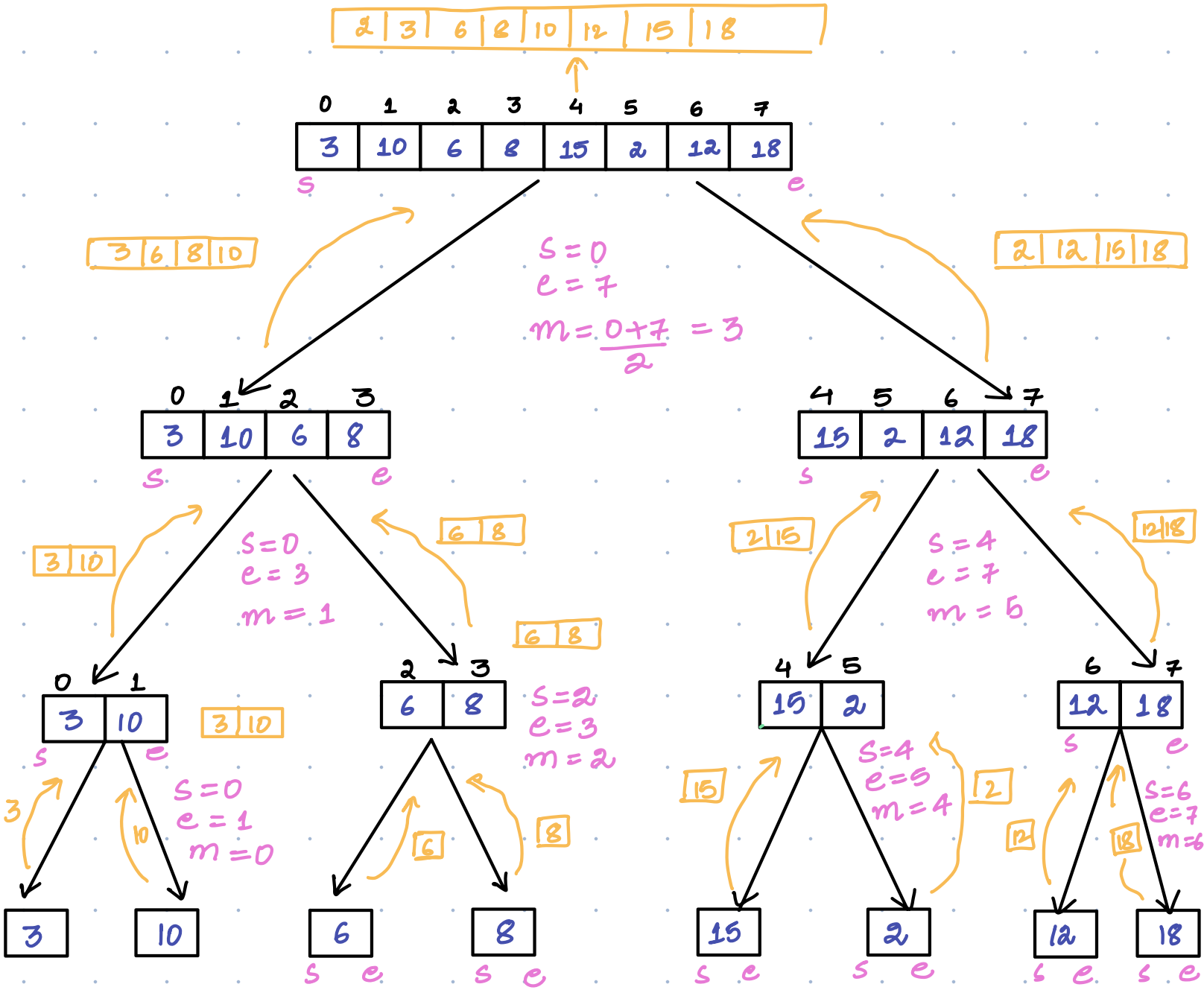
< / > Code

```
int[] merge(int[] A, int s, int m, int e){
    int n = A.size();

    int P1 = s , P2 = m+1 , K = 0;

    int [] C = new int[e-s+1];
    while ( P1 <= m && P2 <= e) {
        // Case 1 :
        if ( A[P1] < A [P2]){
            C[K] = A[P1];
            K++;
            P1++;
        }
        else {
            C[K] = A[P2];
            K++;
            P2++;
        }
    }
    while (P1 <= m){
        C[K] = A[P1];
        K++;
        P1++;
    }

    while (P2 <= e){
        C[K] = A[P2];
        K++;
        P2++;
    }

    for(i = 0; i < C.length; i++) {
        A[S] = C[i];
        S++;
    }
    return A;
}
```

T.C : O(N)
S.C : O(N)

Merge Sort





Idea of Merge Sort : Split the array again & again until it reaches where there is a single element & then start merging the 2 individual sorted parts.

< / > Code

```
void mergeSort(int[]A, int s, int e){
    // Base case
    if(s==e) return;
    m = (s+e)/2;
    // Sort 1st half
    mergeSort(A, s, m);
    // Sort 2nd half
    mergeSort(A, m+1, e);
    // Merge 2 sorted arrays
    merge(A, s, m, e);
}
```

S.C : O(N + Recursive stack space)

: O(N + logN) = O(N)

S.C : O(N)

Time Complexity Analysis [Merge Sort]

Merge sorts ↑ Merge ↑

$T(N) = 2 * T\left(\frac{N}{2}\right) + N$ ——— (1)

$2^1 T(N/2^1) + N$

$N \to N/2$ in (1)

$T(N/2) = 2T(N/4) + N/2$

$T(N) = 2\left[2T\left(\frac{N}{4}\right) + \frac{N}{2}\right] + N$

$T(N) = 4T(N/4) + 2N$ ——— (2)

$2^2 T(N/2^2) + 2N$

$N \to N/4$ in (1)

$T(N/4) = 2T(N/8) + N/4$

$T(N) = 4\left[2T\left(\frac{N}{8}\right) + \frac{N}{4}\right] + 2N$

$T(N) = 8T(N/8) + 3N$ ——— (3)

$2^3 T(N/2^3) + 3N$

After K substitutions :

$T(N) = 2^k T\left(\frac{N}{2^k}\right) + KN$ — (4)

$T(1) = 1$

$N/2^k = 1$

$N = 2^k$

$K = \log_2 N$

Substitute K in (4)

$T(N) = 2^{\log_2 N} \cdot 1 + N \log_2 N$

$T(N) = N + N \log_2 N$

T.C : $O(N + N \log_2 N) \simeq O(N \log_2 N)$

In-place Sorting

Works by rearranging the data within the given input array without the need for any additional memory or any auxillary datastructure.

Stable Sorting

Relative position of duplicate elements is maintained.