Sorting - 2

Agenda:

- Sort 0 & 1
- Pivot Partition
- Quick Sort
- Randomised Quick Sort
- Comparator
- Sorting based on Count of Factors
- Largest Number

Good Morning
(Starting at 7:05)

Sort 0 & 1

< *Question* > : Given an integer array of zeros and ones in random order, segregate 0's on left and 1's on right.

A[] =

0	1	0	0	1	1	0	1	0

o/p :

0	0	0	0	0	1	1	1	1

Idea -1

Use Inbuilt Sorting Algo

T.C : O(N log N)

Idea -2

Count Sort

- Iterate over array & count freq of 0's & 1's. → O(N)
- Iterate again & set 0's & 1's. → O(N)

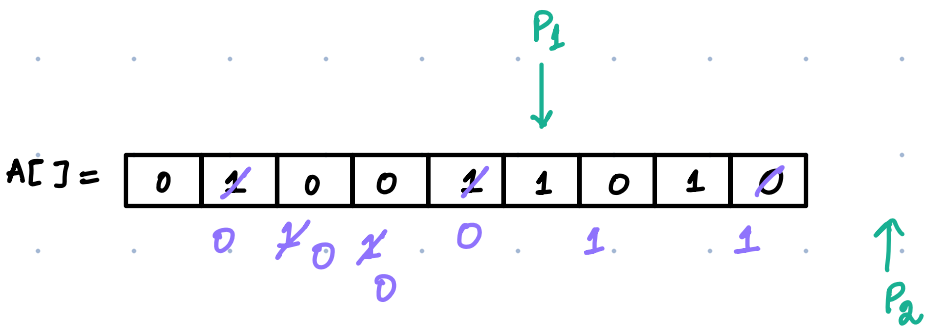
T.C : O(N)

Idea - 3

Two Pointers

P_1 : where 0 should be placed

P_2 : to iterate over the array.



< / > Code

```
void sort01 ( int [ ] arr) {
    P1 = 0 , P2 = 0 ;
    while (P2 < n) {
        if (arr[P2] == 0) {
            swap(arr[P1], arr[P2]);
            P1 ++ ;
        }
        P2 ++ ;
    }
}
```

T.C : O(N)

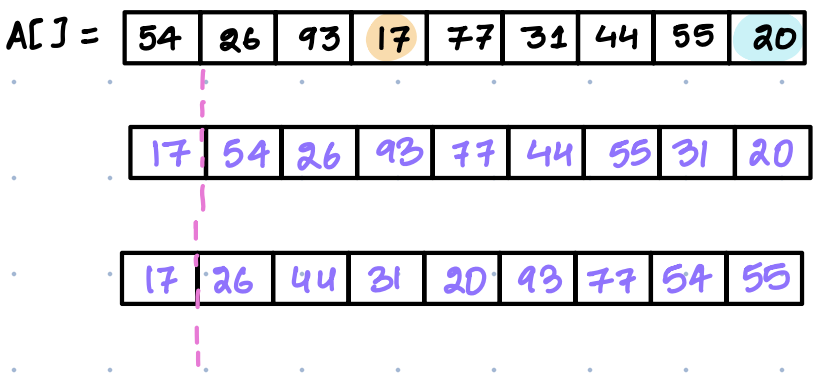
S.C : O(1)

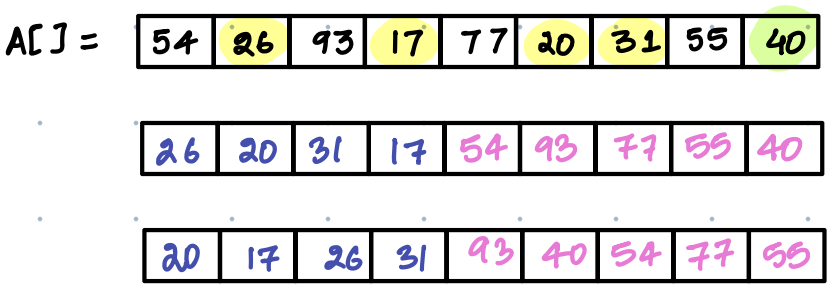
Pivot Partition

< *Question* > : Given an integer array, consider last elements as pivot (p), rearrange the elements such that for all i :

- If A[i] < p then it should be present on left side of array.
- If A[i] >= p then it should be present on right side of array.

$1 <= N <= 10^5$





Idea -1

Sort the array using inbuilt sorting.

T.C: O(N log N)

Idea -2

Two Pointers

P_1 : where ele < pivot should be placed

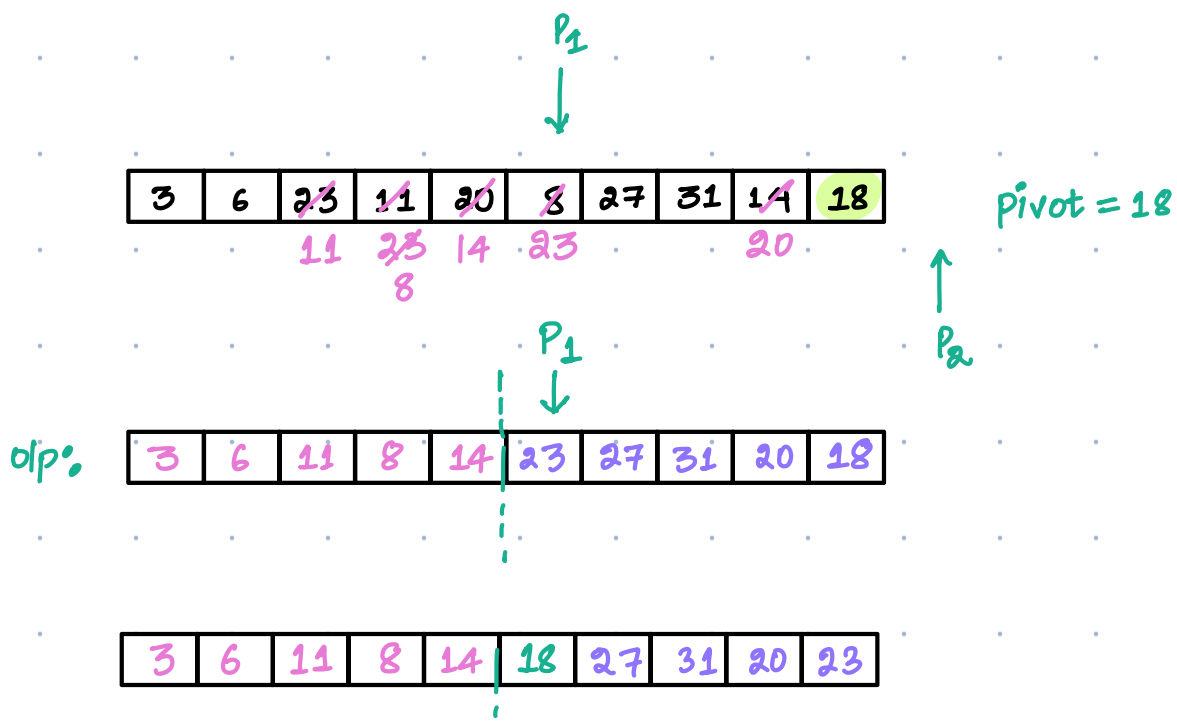
P_2 : to iterate over the array.

< / > Code

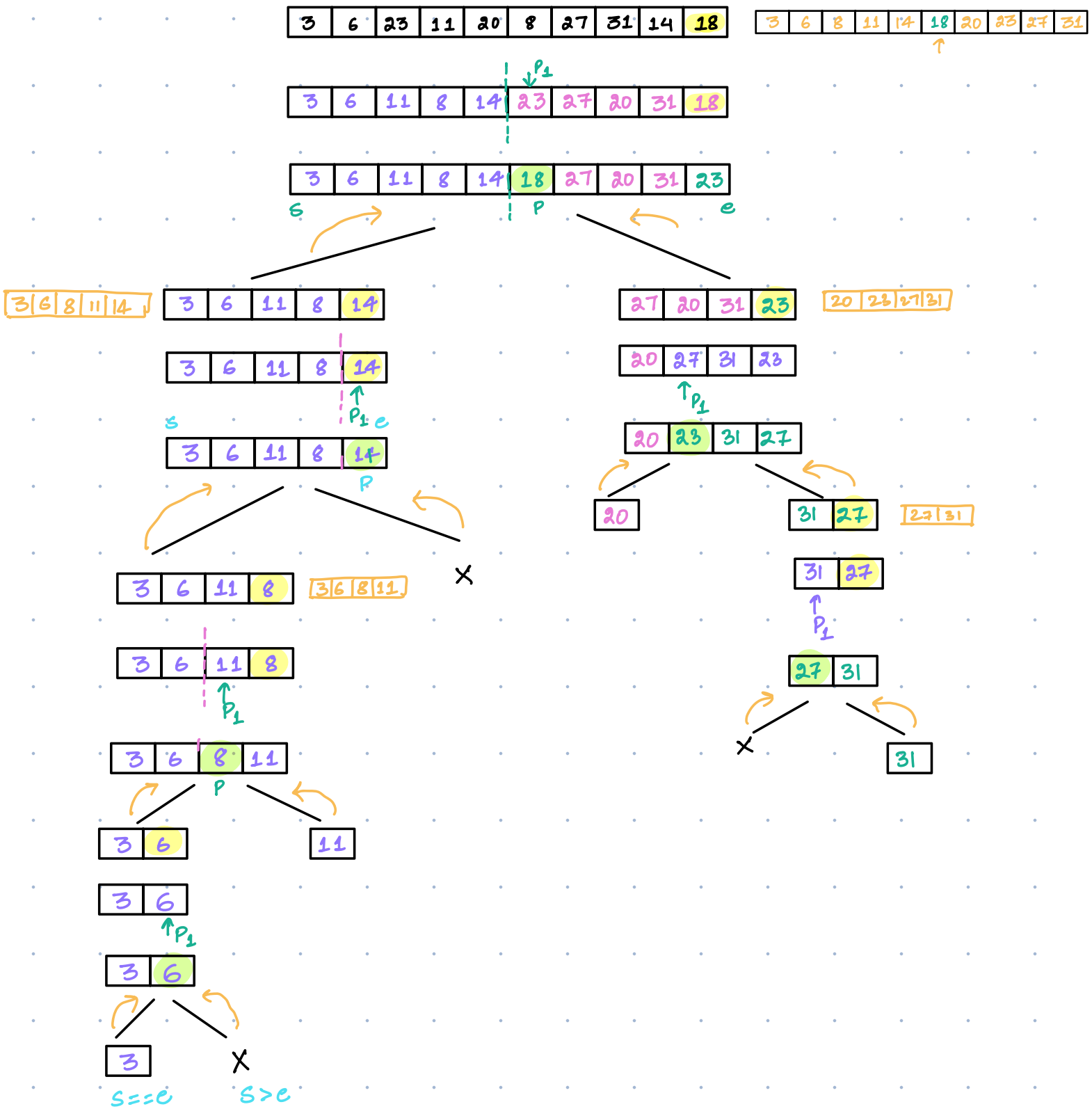
```
void pivotPartition (int [] arr) {
    pivot = arr[n-1];
    P1 = 0 , P2 = 0 ;

    while (P2 < n) {
        if (arr[P2] < pivot) {
            swap (arr[P2], arr[P1]) ;
            P1++;
        }
        P2++;
    }
}
```

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



Quick Sort



Approach:

- Choose a pivot & partition the array.

  smaller element on left side of array
  ↑ greater or equal elements on right side of array.

- Bring pivot element to its correct position in the sorted array (swap pivot element & arr[P,])

- Repeat the process on the 2 subarrays.

  [S, P-1]

  [P+1, e]

< / > Code

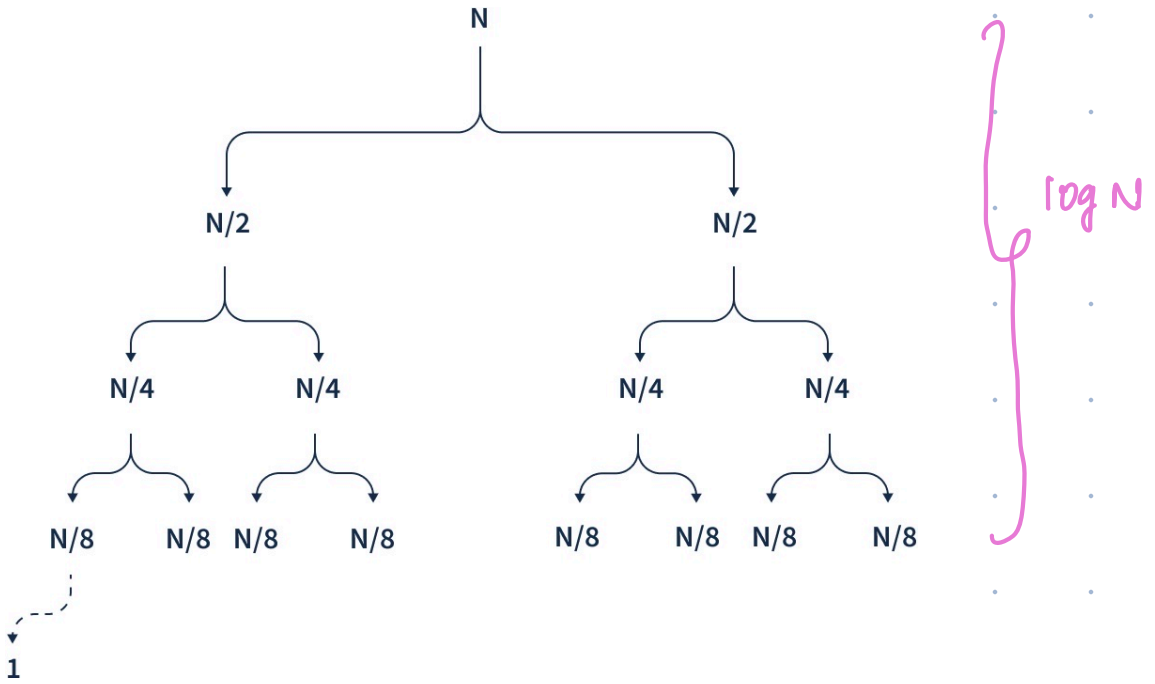
```
void QuickSort(int[] A, int s, int e) {
    if(s >= e) return;    // Base case.
    int pivotIndex = Partition(A, s, e);
    QuickSort(A, s, pivotIndex-1);
    QuickSort(A, pivotIndex+1, e);
}
```

```
int Partition(int[] arr, int s, int e) {
    P1 = s, P2 = s, pivot = arr[e];
    while(P2 <= e) {
        if(arr[P2] < pivot) {
            swap(arr[P1], arr[P2]);
            P1++;
        }
        P2++;
    }
    // Bring pivot element to its correct position
    swap(arr[P1], arr[e]);
    return P1;
}
```

O(N)

Time Complexity Analysis of Quick Sort

Best - Case : When pivot divides the array into 2 equal size subarrays (approx.)



T.C : Total No of Calls * Time per call

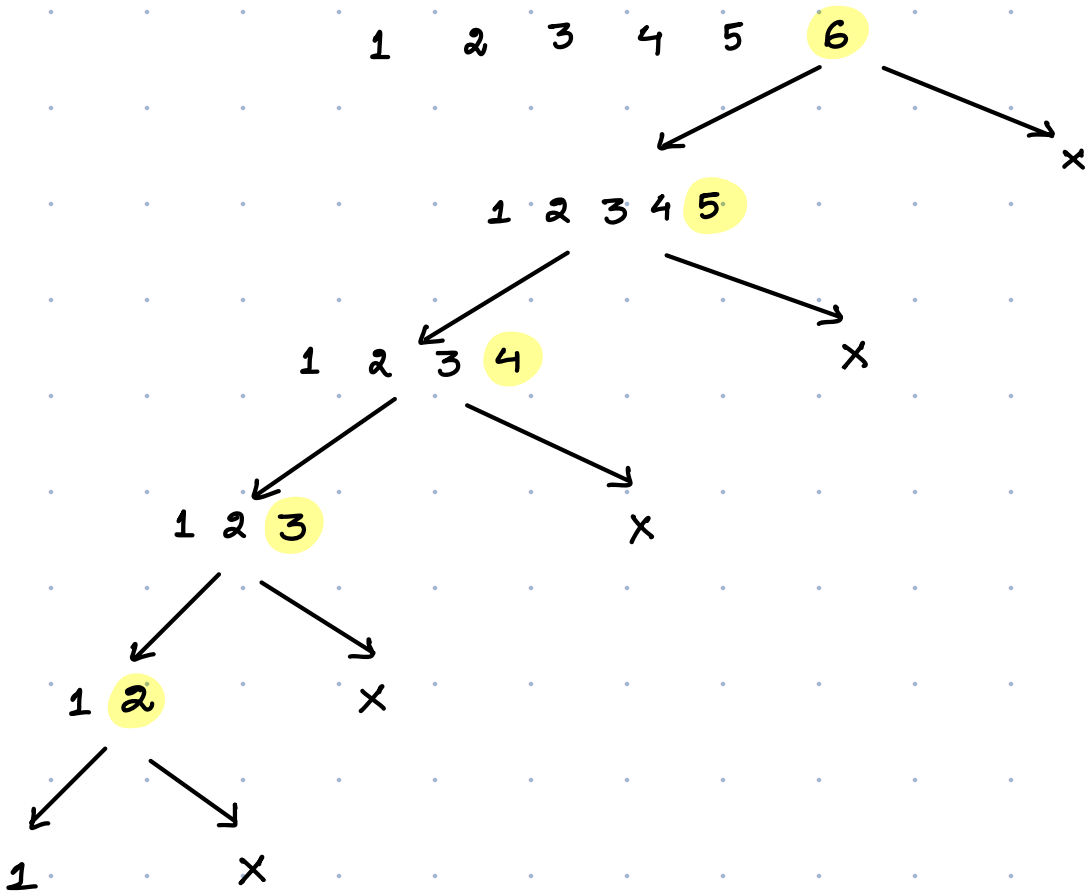
= log N * N

T.C : O(N log N)

S.C : Max Stack Space

S.C : O(log N)

Worst - Case



T.C : Total no of func calls * Time per call

= N * N

T.C : $O(N^2)$

S.C : Max Stack Space

S.C : $O(N)$

Randomised Quick Sort

→ Technique where we randomly select the pivot not necessarily 1st element or last element.

→ Helps to get away with worst time complexity. as probability of always the random element being MIN/MAX is very low.

Given N elements,

- Probability that a random element is minimum :- $\frac{1}{N}$

- Next time with remaining N-1 elements,

  Probability that a random element is minimum :- $1/N-1$

- Next time with remaining N-2 elements,

  Probability that a random element is minimum :- $1/N-2$

- Next time with remaining N-3 elements,

  Probability that a random element is minimum :- $1/N-3$

$$\frac{1}{N} * \frac{1}{N-1} * \frac{1}{N-2} \text{ - - - - - } 1 = \frac{1}{N!}$$

(very small)

Randomised quick sort helps us to achieve avg. T.C of O(N logN) most of the times.

Comparator

- Function that compares 2 values & returns a result indicating whether the values are equal less than or greater than each other.

- Used in Sorting Algorithm to compare elements in a datastructure & arrange them in specified order.

Java, Python, JS, Ruby

- If we want first argument to come before second : **return -ve**
- If we want second argument to come before first : **return +ve**
- If both are same : **return 0**

C++

- If we want first argument to come before second : **return true**

  else : **return false**

Sorting based on Factors

< *Question* > : Given arr[N], sort the data in ascending order of count of factors.

If count of factors are equal, then sort the elements on the basis of their values.

arr - [9 , 3 , 10 , 6 , 4]

0 1 2 3 4

1 <= N <= 10^5

1 <= A[i] <= 10^4

9 → 1, 3, 9 — 3

3 → 1, 3 — 2

10 → 1, 2, 5, 10 — 4

6 → 1, 2, 3, 6 — 4

4 → 1, 2, 4 — 3

o/p : [3, 4, 9, 6, 10]

QUIZ!

10	4	5	13	1

10 → 1, 2, 5, 10 — 4

4 → 1, 2, 4 — 3

5 → 1, 5 — 2

13 → 1, 13 — 2

1 → 1 — 1

o/p : [1, 5, 13, 4, 10]

< / > Code

C++

```
bool compare(int val1, int val2) {

    int f1 = count_of_factors(val1);

    int f2 = count_of_factors(val2);

    if (f1 == f2) {

        if (val1 < val2)  // val1 should come before val2
            return true;
        else
            return false;
    }

    else if (f1 < f2)  // val1 should come before val2
        return true;

    return false;  // val2 should come before val1
}
```

```
vector<int> FactorBasedSorting(vector<int> A) {

    sort(A.begin(), A.end(), compare);
    return A;
}
```

Java

```java
public ArrayList<Integer> solve(ArrayList<Integer> A) {
    Collections.sort(A, new Comparator<Integer>(){
        @Override
        public int comp(Integer v1, Integer v2){
            if(factors(v1) == factors(v2)){
                if(v1<v2) return -1;
                else if(v2<v1) return 1;
                return 0;
            }
            else if(factors(v1)<factors(v2)){
                return -1;
            }
            return 1;
        }
    });
    return A;
}
```

Python

```python
def compare(v1, v2):
    if(factors(v1) == factors(v2)):
        if(v1<v2):
            return -1;
        if(v2<v1):
            return 1;
        else
            return 0;
    elif (factors(v1)<factors(v2)):
        return -1;
    else
        return 1;


class Solution:
    def solve(self, A):
        A = sorted(A, key = functools.cmp_to_key(compare))
        return A
```

Largest Number

Given a list of non-negative integers, arrange them such that they form the largest number and return it. Since the result may be very large, so you need to return a string instead of an integer.

[10, 2]

o/p : "210"

[3, 30, 34, 5, 9]

o/p : "9534330"

sort in des order X

[10, 5, 2, 8, 200]

o/p : "852200 10"

Approach

X	Y
3	30

string X+Y "330" > Y+X "303"

9 "95" > 5 "59"

```
bool compare(int a, int b) {
    if (to_string(a) + to_string(b) > to_string(b) + to_string(a))
        return true;
    else
        return false;
}
```

```
String LargestNumber(int[] arr) {
    sort(arr.begin(), arr.end(), compare);
    if(arr[0] == '0') return "0";
    string ans = "";
    for(i=0; i<arr.length; i++) {
        ans += to_string(arr[i]);
    }
    return ans;
}
```

0, 9, 2, 3, 4 → 9, 4, 3, 2, 0

0, 0, 0, 0, 0 → 0, 0, 0, 0, 0

[3, 30, 34, 5, 9] → [9 5 34 3 30]

ans: "953430"

3_ 34 30

33 334 330