Sorting

- Merge 2 Sorted Arrays
- merge sort
  - Partition Array (Sort 01, Sort 012)
- Quick Sort

Merge 2 Sorted Arrays

A1: 2 4 5 7 8 — O(N) Space

A2: 1 3 6 9 10 11 12 — O(M) space

res: 1 2 3 4 5 6 7 8 9 10 11 12

O(N+M) time → O(N+M) extra space

```java
public static int[] mergeTwoSortedArrays(int[] a, int[] b){
  int i = 0, j =0, k = 0;
  int[] ans = new int[a.length + b.length];
  while(i < a.length && j < b.length){
      if(a[i] <= b[j]){
        ans[k] = a[i];
        i++;
        k++;
      }else{
        ans[k] = b[j];
        j++;
        k++;
      }
  }

  while(i < a.length){
    ans[k] = a[i];
    k++;
    i++;
  }

  while(j < b.length){
    ans[k] = b[j];
    k++;
    j++;
  }

  return ans;
}
```

Tradeoff

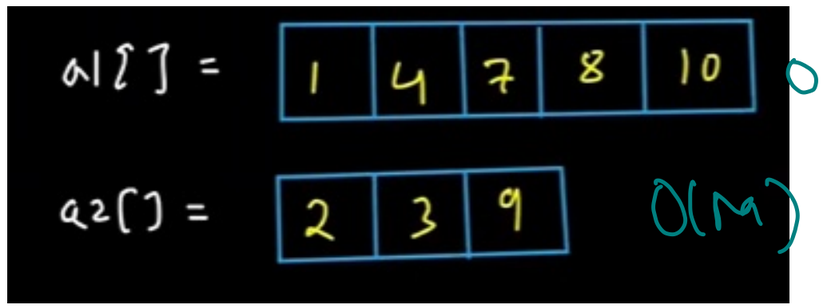
Time vs Space

2 Pointer O(N+M) - O(N+M)

Insertion Sort O(N*M + M^2) - O(1)

Shell Sort O((N+M) log2(N+M)) - O(1)

Without extra space



O(N)

O(M)

→ Using Insertion Sort

Redundant

1 4 7 8 10 2 3 9

1 4 7 8 10 2 3 9

1 4 7 8 10 2 3 9

1 4 7 8 10 2 3 9

0 1 2 3 4 5 6 7 8

1 4 7 8 10 2 3 9 O(N)

1 2 4 7 8 10 3 9 O(N+1)

1 2 3 4 7 8 10 9 O(N+2)

1 2 3 4 7 8 9 10 O(N+M)

$O(N + N+1 + N+2 + \dots N+M)$

$= O(N*M + M*\frac{(M+1)}{2})$

$= O(N*M + M^2)$ fine, $O(1)$ extra space

```java
public void swap(int[] nums1, int a, int b){
    int temp = nums1[a];
    nums1[a] = nums1[b];
    nums1[b] = temp;
}

public void merge(int[] nums1, int m, int[] nums2, int n) {
    for(int i = m; i < m+n; i++){
        nums1[i] = nums2[i - m];
    }

    for(int i=m; i<m+n; i++){
        for(int j=i-1; j>=0; j--){
            if(nums1[j] > nums1[j + 1]){
                swap(nums1, j, j + 1);
            }
            else break;
        }
    }
}
```

Shell Sort

gap sequence = $\lceil \frac{N+M}{2} \rceil, \lceil \frac{N+M}{4} \rceil, \lceil \frac{N+M}{8} \rceil, \ldots 1$

$= \frac{8}{2}, \frac{8}{4}, \frac{8}{8}$

$= 4, 2, 1$

a1[] = | 1 | 4 | 7 | 8 | 10 |

a2[] = | 2 | 3 | 9 |

1 4 7 8 10 2 3 9

1 4 7 8 10 2 3 9

1 2 7 8 10 4 3 9

1 2 3 8 10 4 7 9

gap = 4

n+m-4 comparisons

1 2 3 8 10 4 7 9

1 2 3 8 10 4 7 9

1 2 3 8 10 4 7 9

1 2 3 4 10 8 7 9

1 2 3 4 7 8 9 10

gap = 2

n+m-2 comparisons

1 2 3 4 7 8 10 9

1 2 3 4 7 8 9 10

gap = 1

n+m-1 comparisons

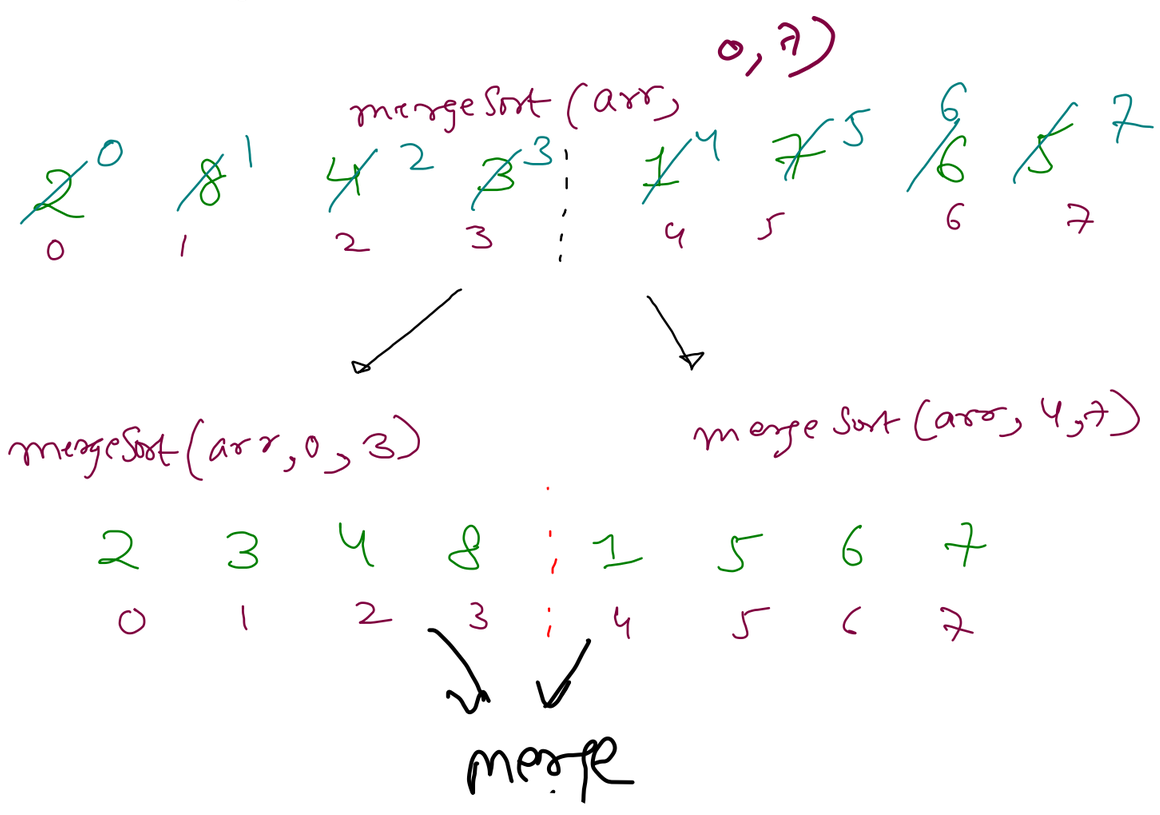
```java
public void merge(int[] nums1, int m, int[] nums2, int n) {
    for(int i = m; i < m+n; i++){
        nums1[i] = nums2[i - m];
    }

    for(double gap = (m + n) / 2.0; gap > 0; gap = gap/2.0){
        int jStart = (int)Math.ceil(gap);
        for(int i=0, j=jStart; j<m+n; i++, j++){
            if(nums1[i] > nums1[j]){
                swap(nums1, i, j);
            }
        }
        if(jStart == 1) break;
    }
}
```

gaps	comparison
$\frac{N}{2}$	$N - \frac{N}{2}$
$\frac{N}{4}$	$N - \frac{N}{4}$
$\frac{N}{8}$	$N - \frac{N}{8}$
$\vdots$	$\vdots$
1	$N - 1$

$$TC = O\left(N\log_2 N - N\left\{\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots 1\right\}\right)$$

$$= O(N\log_2 N) = O((N+M)\log_2(N+M))$$

Merge Sort (with extra space) {Inplace} {Stable}




```java
public void mergeSort(int[] nums, int left, int right){
    if(left >= right){
        return; // 0 or 1 elements in array
    }

    int mid = left + (right - left) / 2;
    mergeSort(nums, left, mid); // left Sorted
    mergeSort(nums, mid + 1, right); // right Sorted
    merge(nums, left, mid, mid + 1, right);
}
```

```java
public void merge(int[] nums, int a1l, int a1r, int a2l, int a2r){
    int totalRes = (a1r - a1l + 1) + (a2r - a2l + 1);
    int[] res = new int[totalRes];

    int ptr1 = a1l, ptr2 = a2l, ptr3 = 0;
    while(ptr1 <= a1r && ptr2 <= a2r){
        if(nums[ptr1] <= nums[ptr2]){
            res[ptr3] = nums[ptr1];
            ptr3++; ptr1++;
        }
        else {
            res[ptr3] = nums[ptr2];
            ptr3++; ptr2++;
        }
    }

    while(ptr1 <= a1r){
       res[ptr3] = nums[ptr1];
       ptr3++; ptr1++;
    }

    while(ptr2 <= a2r){
       res[ptr3] = nums[ptr2];
       ptr3++; ptr2++;
    }

    for(int i=a1l; i<=a2r; i++){
        nums[i] = res[i - a1l];
    }
}
```

Two pointer

$2^0 T(N) = 2T(N/2) + O(\frac{N}{2} + \frac{N}{2})$

$2^1 T(N/2) = 2^2 T(N/4) + 2O(\frac{N}{4} + \frac{N}{4})$

$2^2 T(N/4) = 2^3 T(N/8) + 2^2 O(\frac{N}{8} + \frac{N}{8})$

$2^3 T(N/8) = 2^4 T(N/16) + 2^3 O(N/16 + N/16)$

$\vdots \quad \vdots \quad \vdots \quad \vdots$

$2^x T(1) = 2T(0) + O(1)$

$T(N) = O(N) + 2 * O(\frac{N}{2}) + 2^2 * O(\frac{N}{4}) + 2^x * O(1)$

$= N + N + N + \cdots \quad x \text{ times}$

$= N * x = O(N * \log_2 N)$

$N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8} \cdots 1$

$x = no\ of\ eqns$

$= \log_2 N$

$2^a = 2^{\log_2 N}$

$= N$

$ar^{n-1} = 1$

$a = N$

$r = 1/2$

$N(\frac{1}{2})^x = 1$

$2^x = N$

$\log_2(2^x) = \log_2 N$

$x = \log_2 N$

$5^{-}\ 3^{x}\ 4^{p}\ 1^{a}\ 2\ \ 4^{q}\ 3^{y}\ 1^{b}$

$\{\ 1^{a}\ 3^{x}\ 4^{p}\ 5\}\ \{\ 1^{b}\ 2\ \ 3^{y}\ 4^{q}\}$

$\{\ \boxed{1^{a}\ 1^{b}}\ 2\ \ \boxed{3^{x}\ 3^{y}}\ \boxed{4^{p}\ 4^{q}}\ 5^{-}\}$

Stable

↓

preserving the original order of elements are equal

(without extra space)

Shell sort

$x = \log_2 N$

$2^0 \left(T(N) = 2T(N/2) + O(N \log N) \right)$

$2^1 \left(T(N/2) = 2T(N/4) + O\left(\frac{N}{2} \log \frac{N}{2}\right) \right)$

$2^2 \left(T(N/4) = 2T(N/8) + O\left(\frac{N}{4} \log \frac{N}{4}\right) \right)$

$\vdots$

$2^x \left(T(1) = 2T(0) + O(1) \right)$

$T(N) = N \log N + 2^1 * \frac{N}{2} \log \frac{N}{2} + 2^2 * \frac{N}{2^2} \log \frac{N}{2^2}$

$+ \cdots \; 2^x * \frac{N}{2^x} \log \frac{N}{2^x}$

$= N \log N + N \log \frac{N}{2} + N \log \frac{N}{2^2} + N \log \frac{N}{2^3} + \cdots N \log \frac{N}{N}$

$= N \left\{ \log N + \log \frac{N}{2} + \log \frac{N}{2^3} + \log \frac{N}{2^4} + \cdots \log \frac{N}{N} \right\}$

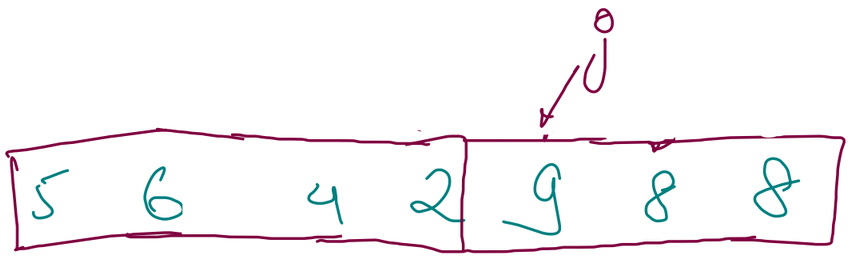
$= N \left\{ \log \left\{ \frac{N * N * N \cdots x \text{ times}}{2^0 * 2^1 * 2^2 * \cdots 2^x} \right\} \right\} = N \left(\log \frac{N^x}{2^{\log_2^2 N}} \right)$

$= N \log_2^2 N$

Partition Array

~~Stable~~ ⇒ quickSort
↳ ~~Stable~~

pivot = 6



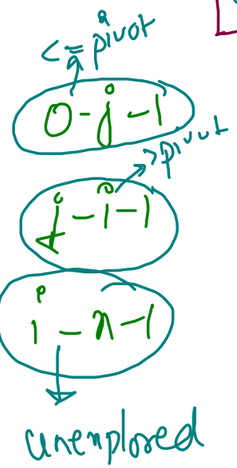
→ -ve, +ve

→ 0, 1 {Sort 01}

→ 3 env, non-zero (+, -)

→ odd, even

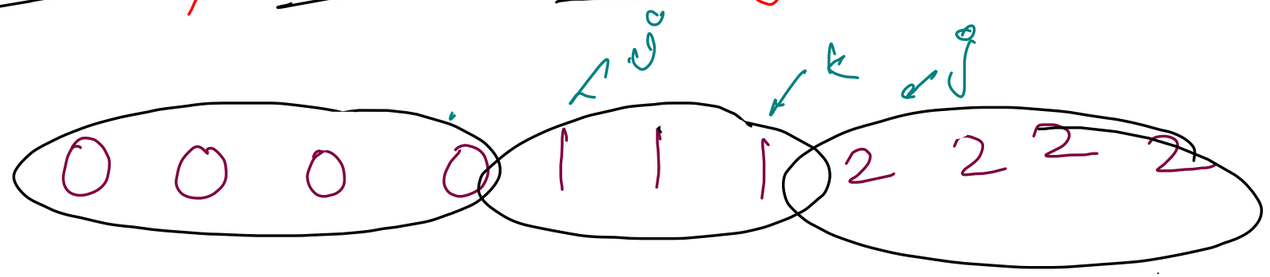
→ <=pivot, >pivot



Partition
O(n)

```
int i = 0, j = 0;
while(i < arr.length)
{
    if(arr[i] <= pivot)
    {
        swap(arr, i, j);
        i++; j++;
    }
    else i++;
}
```

DNF Sort / Sort 0'1'2' / Three way Partitioning



Single traversal

=0's (0 - i-1)

=1's (i - j-1)

=2's (k+1) (n-1)

unexplored (j - k)

```
int i = 0 , j = 0 , k = arr.length - 1;
while(j <= k)
{
    if(arr[j] == 0)
    {
        swap(arr,j,i);
        i++;
        j++;
    }
    else if(arr[j] == 2)
    {
        swap(arr,j,k);
        k--;
    }
    else j++;
}
```

0 , 1 , 2

%3 == 0 , %3 == 1 , %3 == 2

< pivot , = pivot , > pivot

< a , [a,b] , > b { Dual pivot partitioning }

Sorting - lecture → 2

(IMPORTANT)

- Inversion Count → using Merge Sort
- Reverse Pairs
- Count of Smaller Elements After Self
- Count subarrays with 1s > 0s
- minimum Swaps to Sort - I & II

Count Inversions

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

(i, j) $i < j$ $a[i] > a[j]$

(8,5) (8,3) (8,4) (8,1) (8,6) (8,2)

(5,3) (5,4) (5,1) (5,2)

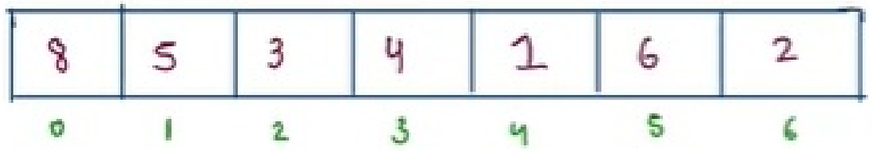
(3,1) (3,2) (4,1) (4,2) (6,2)

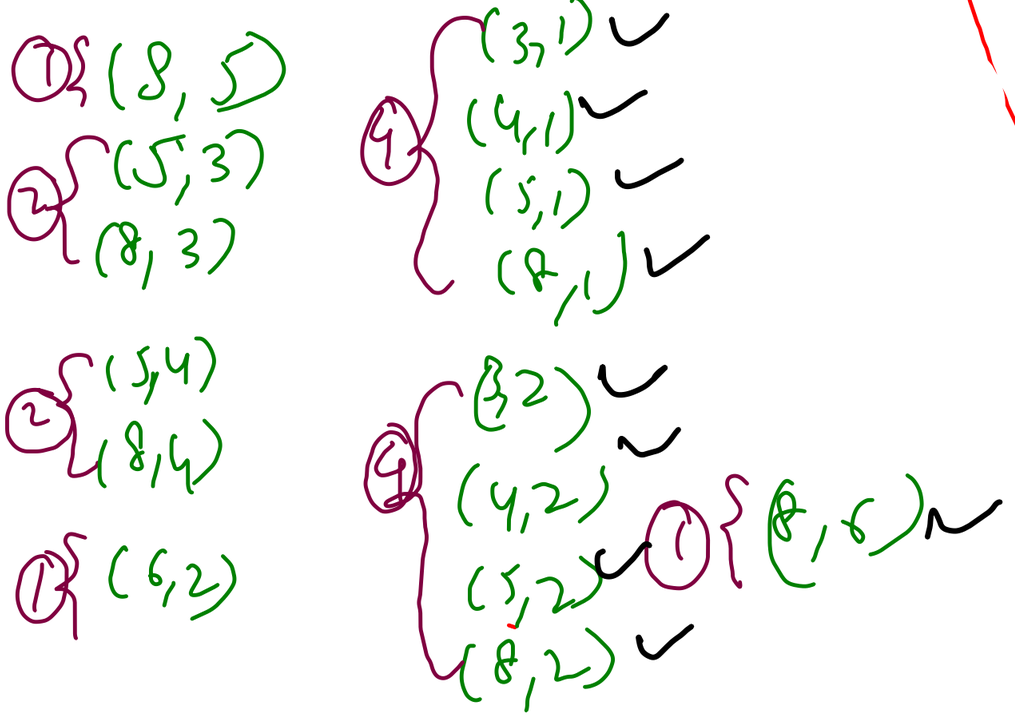
15 pairs

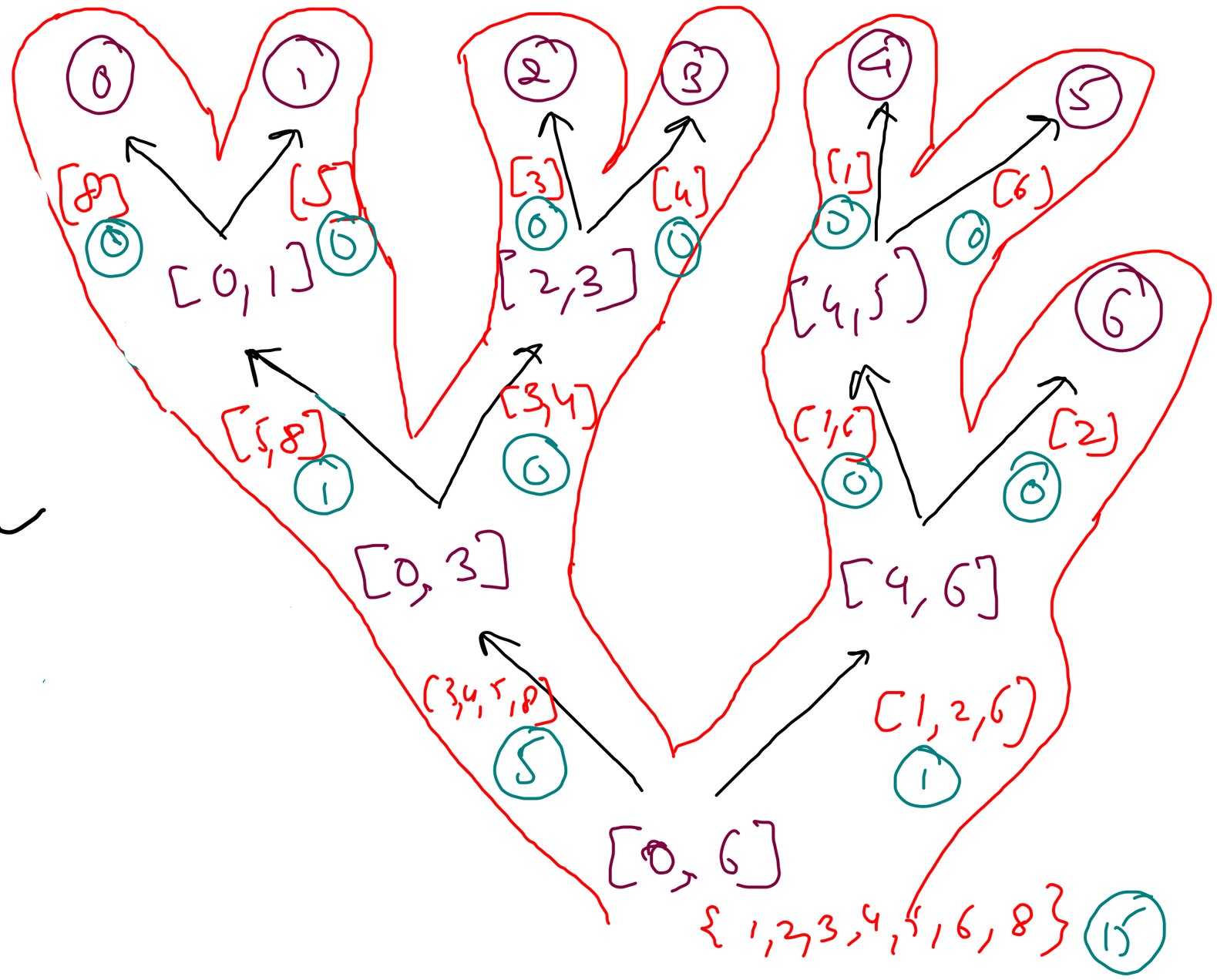
Brute force :→ $O(N^2)$

Optimization

- ⭐ → Merge Sort ($N \log N$)
- → Fenwick Tree
- → Policy Based Data Structure
- → Self Balancing BST








```java
public void mergeSort(int[] nums, int left, int right){
    if(left >= right){
        return; // 0 or 1 elements in array
    }

    int mid = left + (right - left) / 2;
    mergeSort(nums, left, mid); // left Sorted
    mergeSort(nums, mid + 1, right); // right Sorted
    merge(nums, left, mid, mid + 1, right);
}
```

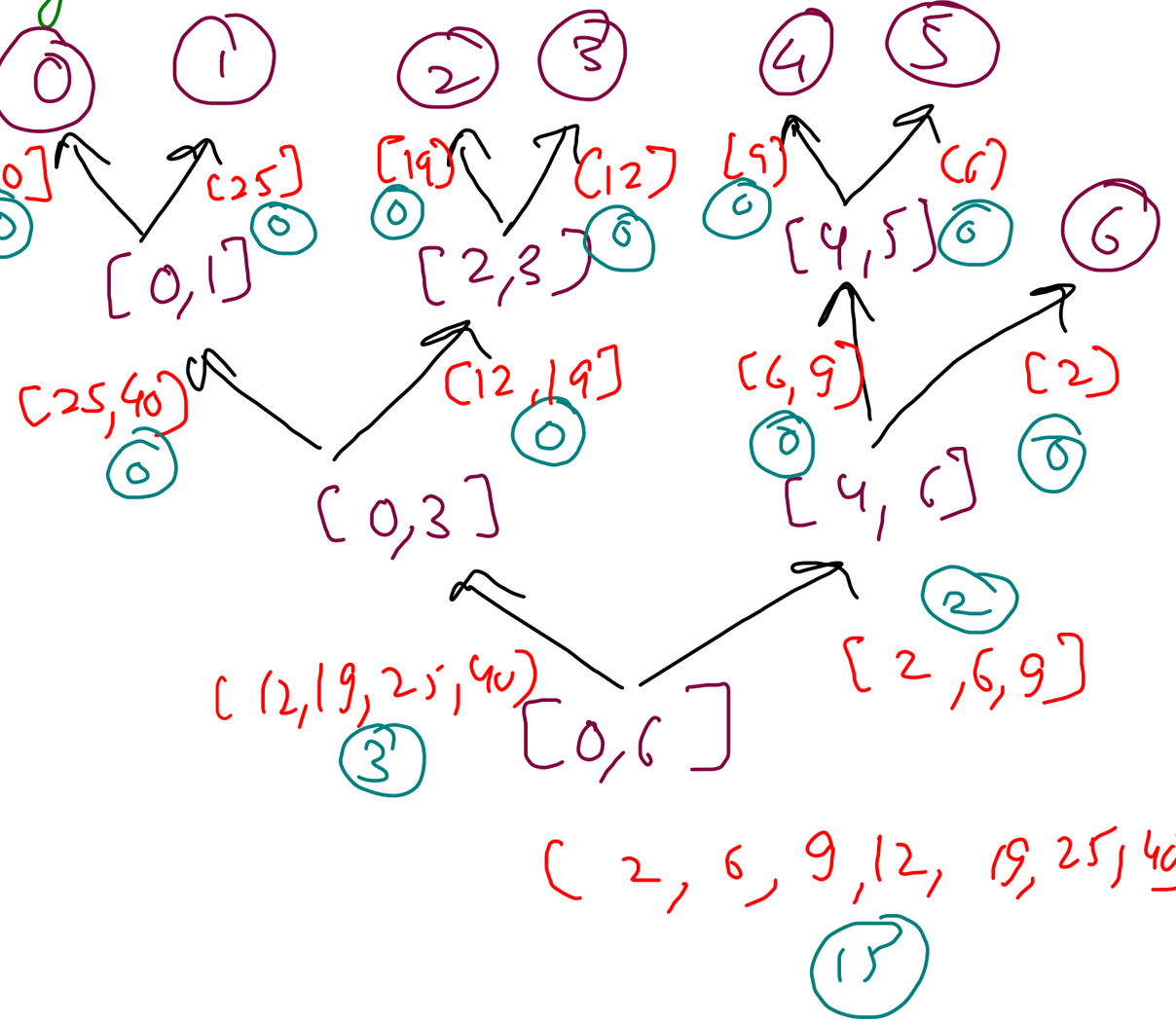
Reverse Pairs

$i < j$ but $a[i] > 2 * a[j]$

(25,12) (40,12) (40,19)

(6,2) (9,2) (12,2) (19,2) (25,2)
(40,2)

(19,6) (25,6) (40,6) (19,9) (25,9) (40,9)



```java
public int findInversionCount(int[] nums, int a1l, int a1r, int a2l, int a2r){
    int invCount = 0;
    int ptr1 = a1l, ptr2 = a2l, ptr3 = 0;
    while(ptr1 <= a1r && ptr2 <= a2r){
        if((long)nums[ptr1] <= 2l * (long)nums[ptr2]){
            ptr3++; ptr1++;
        }
        else {
            invCount += (a1r - ptr1 + 1);
            ptr3++; ptr2++;
        }
    }
    return invCount;
}
```

GOOGLE Count winning Streaks (NADAS)

(Premium) leetcode 2031 Count subarrays with more 1s than 0s

$N^3 \rightarrow N^2 \rightarrow N \log N$

3 loops

optimised subarray (2 loops)

for count {prefix}

0 1 2 3 4 5 6 7

[0, 1, 1, 0, 1, 1, 1, 0]

(0,1,1) (1,1,0,1)

(1,1,0) (1,0,1)

- - -

- - -

```java
public static int winningStreak(int[] arr){
    int ans = 0;

    for(int st=0; st<arr.length; st++){
        int countof0s = 0, countof1s = 0;
        for(int end=st; end<arr.length; end++){
            if(arr[end] == 0){
                countof0s++;
            } else {
                countof1s++;
            }

            if(countof1s > countof0s){
                ans++;
            }
        }
    }

    return ans;
}
```

$O(N^2)$ (TLE)

"consider all subarrays"

[0, 1, 1, 0, 1, 1, 1]

[-1, 0, 1, 0, 1, 2, 3]

1s > 0s

↓ ↓

+1 −1

4 {0,1,1}

{0,1,1,0,1}

{0,1,1,0,1,2}

{0,1,1,0,1,1,1}

[1]

[1,1]

[1,1,1]

[1,1,1,0,1]

(1)

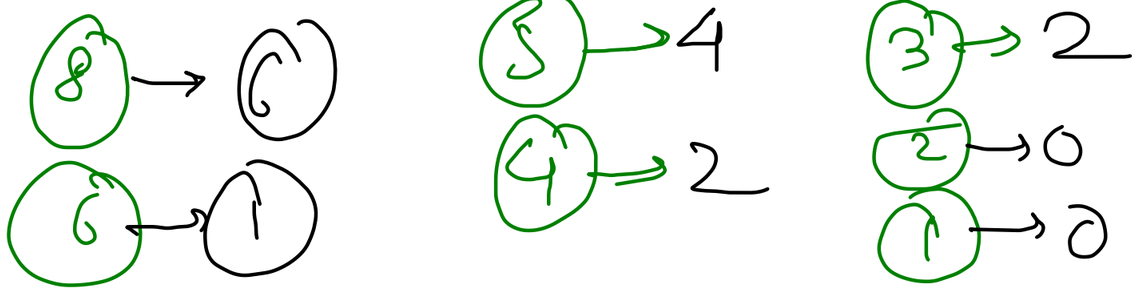
(1,1)

(0,1,1)

(1,1,0)

```cpp
int subarraysWithMoreZerosThanOnes(vector<int> &nums)
{
    int n = nums.size();
    vector<int> pref(n);
    int sum = 0;

    for (int i = 0; i < n; i++) {
        if (nums[i] == 0)
            nums[i] = -1;
        sum += nums[i];
        pref[i] = sum;
    }

    int inversions = 0;
    for (int i = 0; i < n; i++) {
        if (pref[i] > 0)
            inversions = (inversions + 1) % mod;
    }

    reverse(pref.begin(), pref.end());
    mergeSort(pref, 0, n - 1, inversions);
    return inversions;
}
```

Minimum Swaps to Sort (II)



bubble sort-

Min adjacent swaps to Sort

⇓

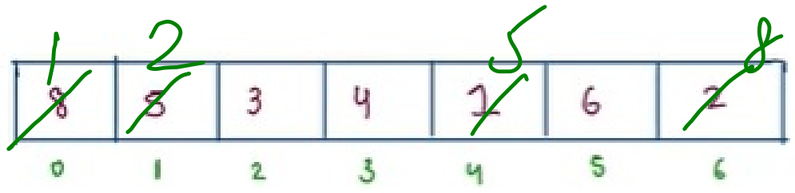
Inversion Count

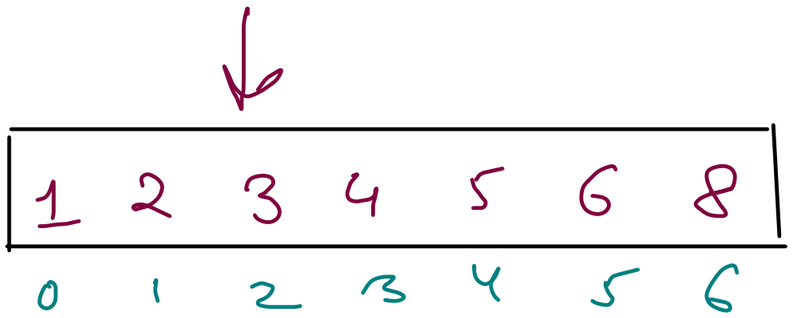
$O(N \log N)$ Merge sort

Minimum swaps to Sort (I)

↳ Adjacent ✗

val → (initial, final)





8 → (0, 6)

5 → (1, 4)

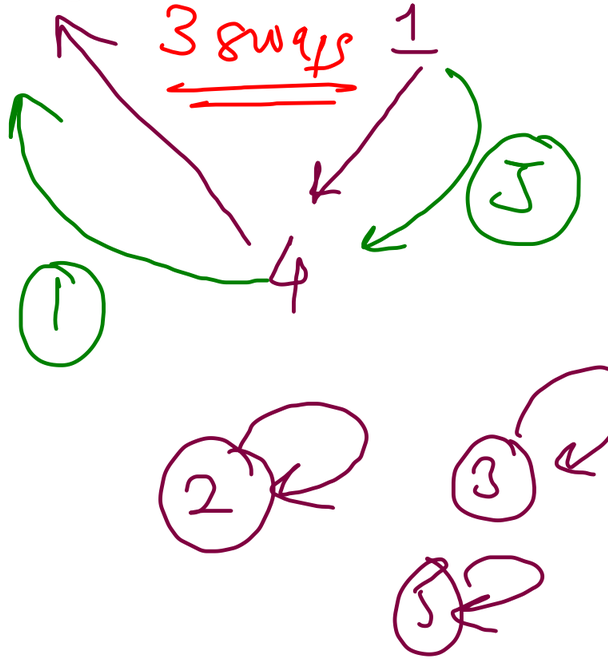
3 → (2, 2)

4 → (3, 3)

1 → (4, 0)

6 → (5, 5)

2 → (6, 1)



Corner case

2 3 4 5 6 7 8 1

0 1 2 3 4 5 6 7

1 2 3 4 5 6 7 8

0 → 1

1 → 2

2 → 3

3 → 4

4 → 5

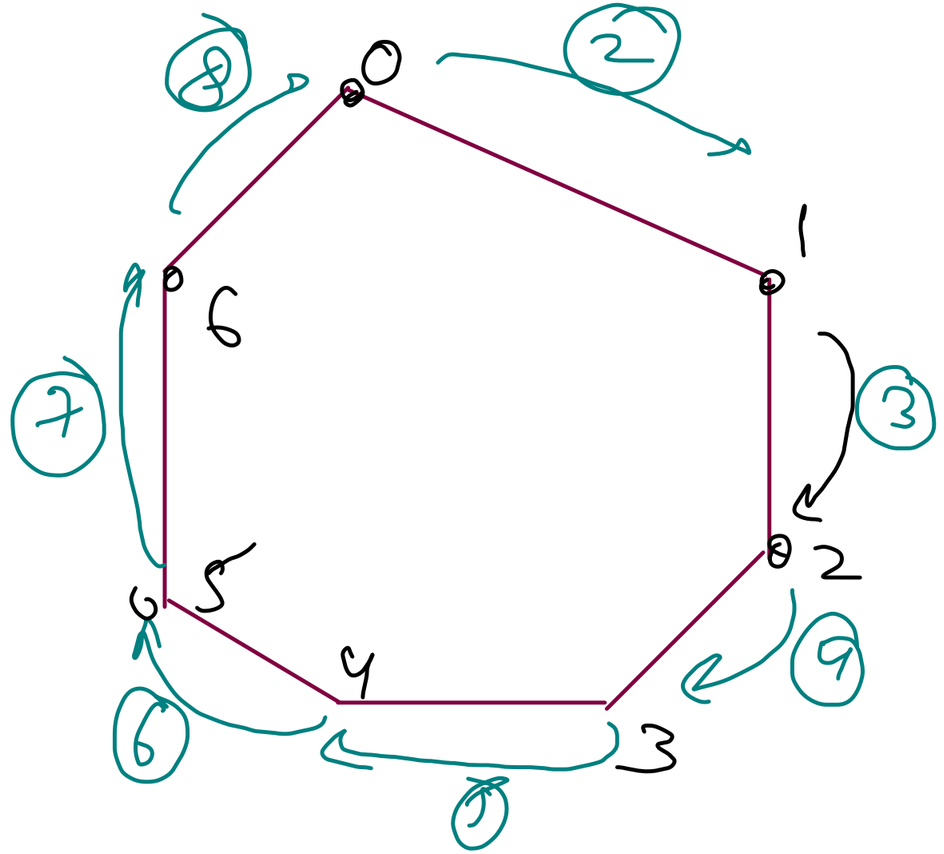
5 → 6

6 → 7

7 → 0

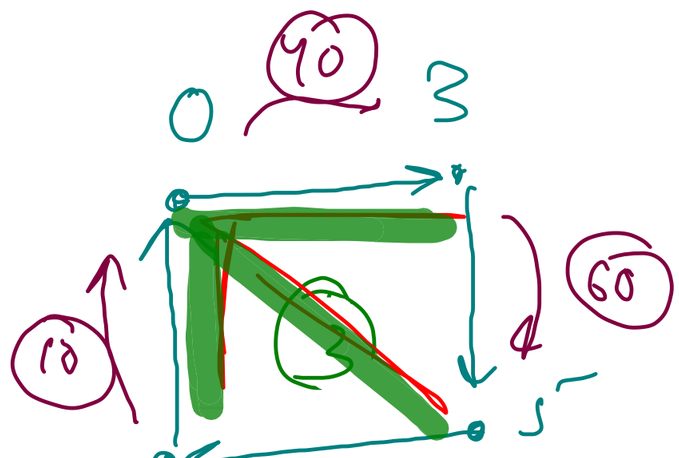
6 swaps

cycle_length - 1



40	50	30	60	20	70	10	80
0	1	2	3	4	5	6	7
10	20	30	40	50	60	70	80

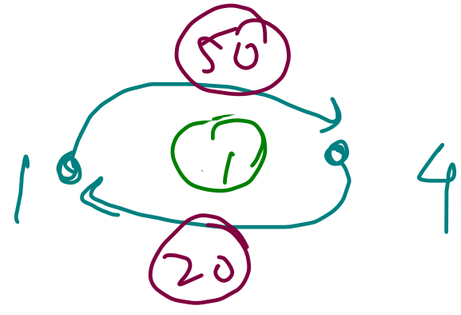
distinct

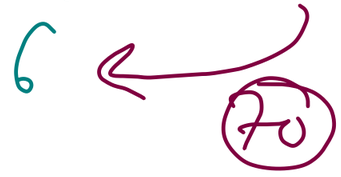


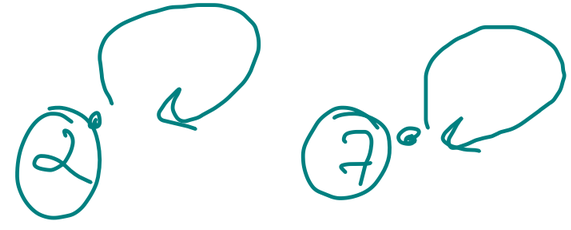
40 $0 \to 3$ 60 $3 \to 5$ 10 $6 \to 0$

50 $1 \to 4$ 20 $4 \to 1$ 80 $7 \to 7$

30 $2 \to 2$ 70 $5 \to 6$







{Total nodes - no of components}

```cpp
int cycle_size(vector<pair<int,int>> &arr, int start, vector<bool> &vis)
{
    int res = 0;
    int j = start;
    while (!vis[j])
    {
        vis[j] = true;
        j = arr[j].second;
        res++;
    }
    return res - 1;
}
int minSwaps(int a[], int n){
    vector<pair<int, int>> arr(n);
    for(int i=0;i<n;i++)
        arr[i] = make_pair(a[i], i);
    sort(arr.begin(), arr.end());
    vector<bool> vis(n, false);
    int res = 0;
    for(int i=0;i<n;i++)
    {
        if(vis[i] == false)
            res += cycle_size(arr, i, vis);
    }
    return res;
}
```

→ O(N log N)

} ⇒ O(N)

Sorting - lecture (3)

Revision
↳ Count Sort, Radix Sort

Bucket Sort

(Q) Top K frequent Elements

(Q) Frequency Sort

Coordinate Compression

Extra Questions

(Q) Maximum Gap → Radix Sort

(Q) Relative Sort Array

(Q) H Index

```cpp
// Karenge to Sorting hi, but tareeka hoga 'Radix Sort'
// Time Complexity - O(N * digits) = O(N * 5) -> Linear
// Space Complexity - O(N + Digits) = O(N) -> Linear

    void countSort(vector<int> &arr, int n, int exp)
    {
        int output[n];
        vector<int> count(10, 0);

        for (int i = 0; i < n; i++)
            count[(arr[i] / exp) % 10]++;

        for (int i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for (int i = n - 1; i >= 0; i--)
        {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }

        for (int i = 0; i < n; i++)
            arr[i] = output[i];
    }

int maximumGap(vector<int>& nums)
{
    if(nums.size() <= 1) return 0;

    // Radix Sort
    int n = nums.size();
    int m = *max_element(nums.begin(), nums.end());
    for(int exp = 1; m/exp > 0; exp *= 10)
        countSort(nums, n, exp);

    int ans = 0;
    for(int i=0; i<n-1; i++)
        ans = max(ans, nums[i+1] - nums[i]);
    return ans;
}
```

Maximum Gap

↓

Radix Sort

Count Sort

Elements in Small Range

9 6 3 5 3 4 3 9 6 4 6 5 8 2 9

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

1	3	2	2	3	0	1	3

0 1 2 3 4 5 6 7

[2] [3] [4] [5] [6] [7] [8] [9]

min

max

$O(max - min + 1)$

Extra space

Time $\Rightarrow O(N)$

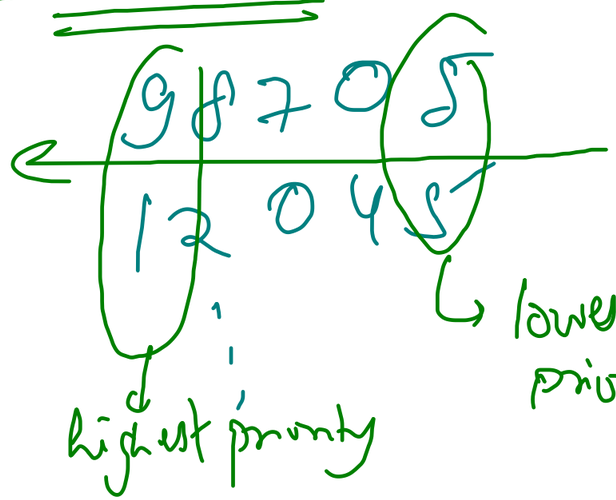
2 3 3 3 4 4 5 5 6 6 6 8 9 9 9

digits

TC :→ O(10*N)

SC :→ O(N)

Radix Sort



Bucket Sort

① Characters → String Sort

26, lowercase / uppercase

52 → uppercase + lowercase

256 → maximum characters

"pepcoding"

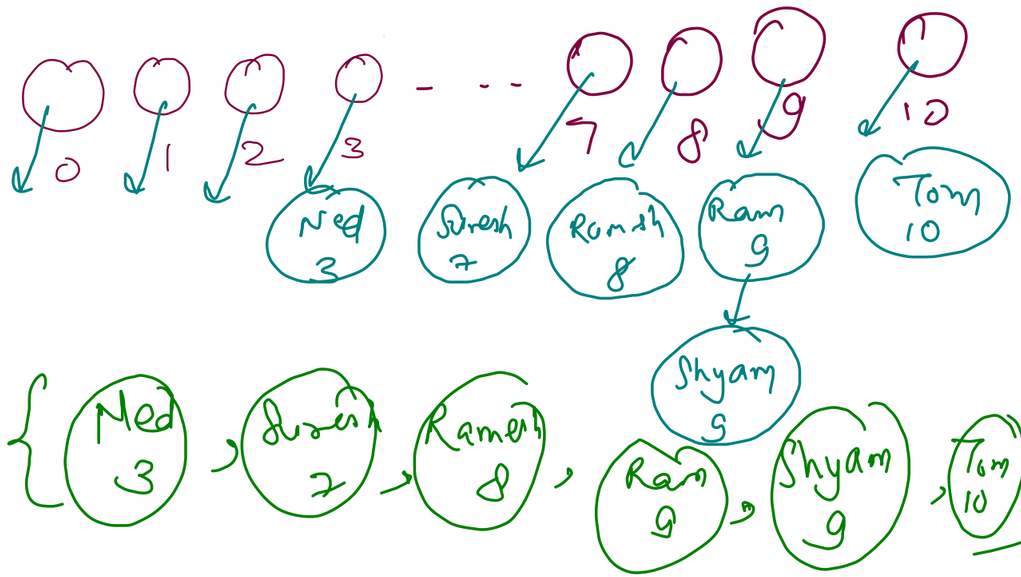
a 1 c ---- z

0 0 1 ----

② Marks → (0, 100)

Ram → 9
Ramesh → 8
Shyam → 9
Suresh → 7
Tom → 10
Ned → 3

Sorting objects based



No of buckets = no of distinct values = 11

O(no of buckets + no of elements)

= O(11 + N) => O(N)

Time => O(N)

③ Values => N elements => values [0, N-1] or [1, N]

↳ N buckets

1636 Frequency Sort

```java
public int[] frequencySort(int[] s) {
    HashMap<Integer, Integer> freq = new HashMap<>();
    for(int i=0; i<s.length; i++){
        freq.put(s[i], freq.getOrDefault(s[i], 0) + 1);
    }

    TreeSet<Integer>[] buckets = new TreeSet[s.length + 1];
    for(int i=0; i<buckets.length; i++){
        buckets[i] = new TreeSet<>();
    }

    for(Integer ch: freq.keySet()){
        int val = freq.get(ch);
        buckets[val].add(ch);
    }

    int[] res = new int[s.length];
    int idx = 0;

    for(int b=0; b<buckets.length; b++){
        for(Integer ch: buckets[b].descendingSet()){
            for(int f=0; f<b; f++){
                res[idx++] = ch;
            }
        }
    }
    return res;
}
```

Best Case O(N)

TreeSet (Java) O(logN) ← Insert & Removal

Ordered set (C++)

↳ Self Balancing BST (AVL)

{1,2,3,4,5,6,7,8,9} → {1,2,3 ... 9}

O(NlogN) worst case } all distinct keys

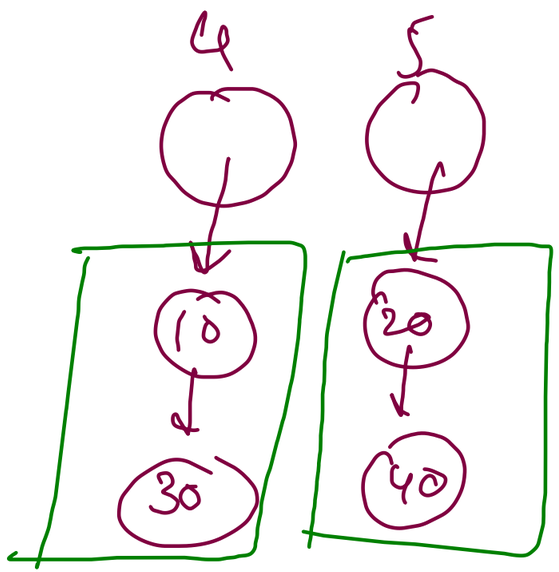
10 → 4

20 → 5

30 → 4

40 → 1

key, Value (freq)



C++ Code

```cpp
vector<int> frequencySort(vector<int>& nums)
{
    vector<int> res;
    int n = nums.size();
    if (n == 0) return res;

    unordered_map<int, int> freq;
    for (auto i : nums) freq[i]++;

    vector<set<int, greater<int>>> bucket(n + 1);
    for (auto &p: freq)
        bucket[p.second].insert(p.first);


    for (int i=1; i<=n; i++)
        for (auto &e: bucket[i])
            for(int j=0; j<i; j++)
                res.push_back(e);

    return res;
}
```

451 Sort Characters by Frequency

```java
public String frequencySort(String s) {
    HashMap<Character, Integer> freq = new HashMap<>();
    for(int i=0; i<s.length(); i++){
        char ch = s.charAt(i);
        freq.put(ch, freq.getOrDefault(ch, 0) + 1);
    }

    ArrayList<Character>[] buckets = new ArrayList[s.length() + 1];
    for(int i=0; i<buckets.length; i++){
        buckets[i] = new ArrayList<>();
    }

    for(Character ch: freq.keySet()){
        int val = freq.get(ch);
        buckets[val].add(ch);
    }

    StringBuilder sb = new StringBuilder("");
    for(int b=buckets.length-1; b>=0; b--){
        for(Character ch: buckets[b]){
            for(int f=0; f<b; f++){
                sb.append(ch);
            }
        }
    }
    return sb.toString();
}
```

hashmap fill

bucket init

buckets fill

sort on basis of freq (Dec)

① O(N)

② O(N)

③ O(62)

④ O(N)

"bbbaaaccc"

Hashmap {O(62)}

b → 1 2 3 4

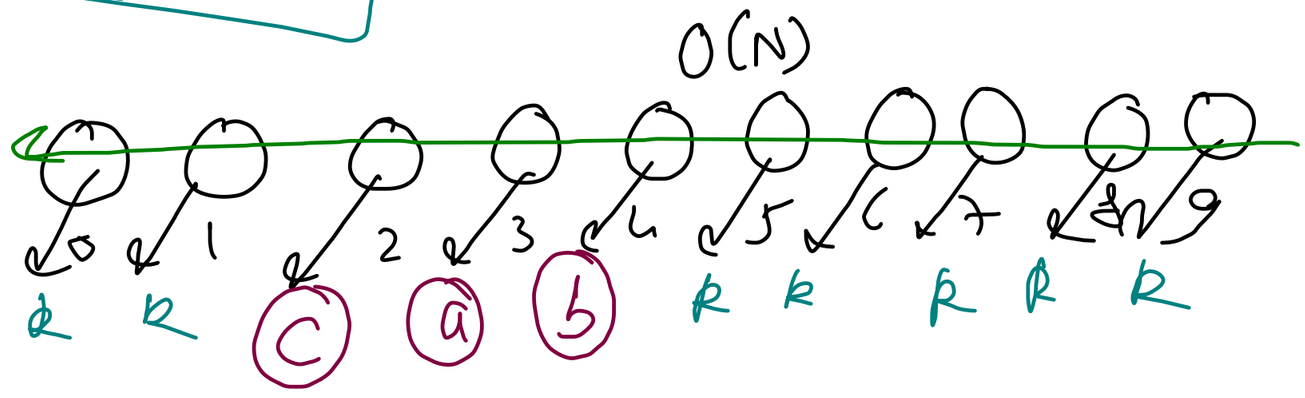
a → 1 2 3

c → 1 2

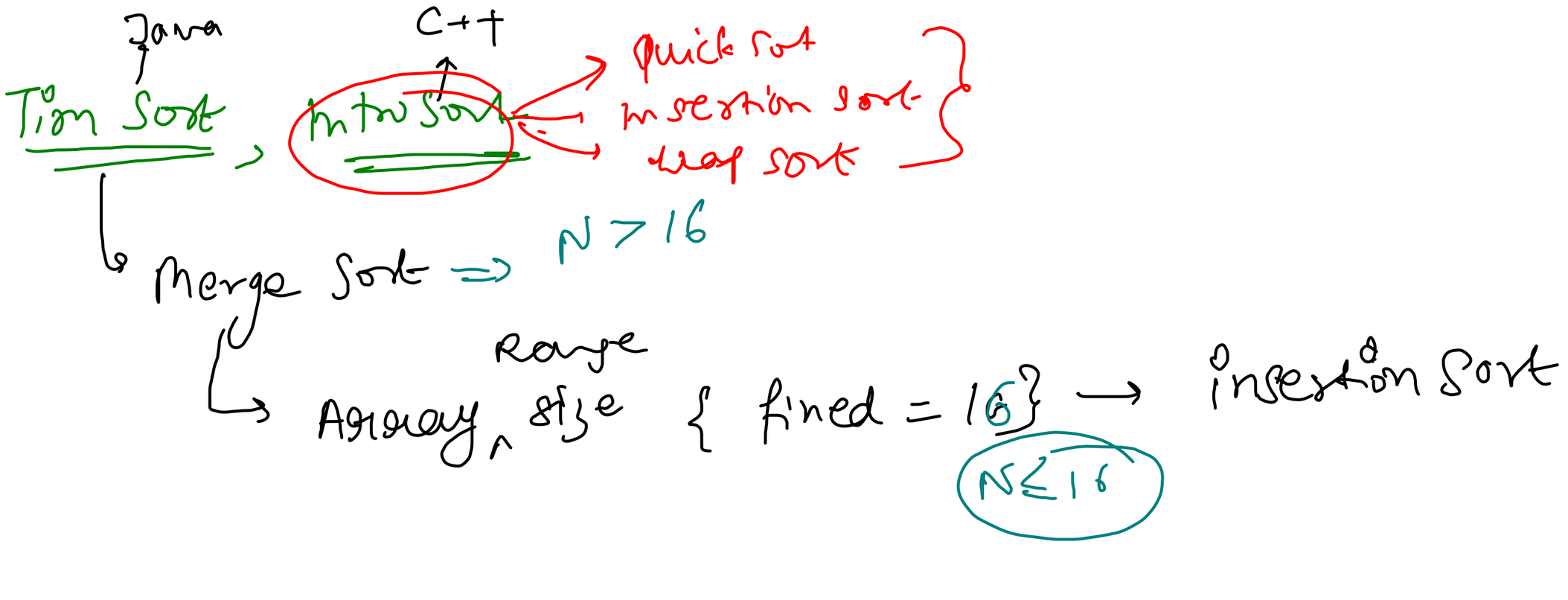
[0, 9]

no of buckets = sb.length + 1

O(N)



"bbbaaaccc"


Java
C++
Tim Sort
Intro Sort
Quick sort
insertion sort
Heap sort
Merge Sort ⇒ N > 16
Range
Array size { fixed = 16 } → insertion Sort
N ≤ 16

Top K Frequent Elements

(347, 692)

Given an integer array `nums` and an integer `k`, return *the* `k` *most frequent elements*. You may return the answer in **any order**.

Constraints:

- `1 <= nums.length <= 10^5`
- `k` is in the range `[1, the number of unique elements in the array]`.
- It is **guaranteed** that the answer is **unique**.

Follow up: Your algorithm's time complexity must be better than `O(n log n)`, where n is the array's size.

① Priority Queue ⇒ O(N log N)

② Bucket Sort
↳ O(N)

{1, 1, 1, 2, 2, 3}

1 → 3 2 → 2 3 → 1

0 1 2 3 4 5 6 7

3 2 1

Java Code

```java
public int[] topKFrequent(int[] s, int k) {
    HashMap<Integer, Integer> freq = new HashMap<>();
    for(int i=0; i<s.length; i++){
        freq.put(s[i], freq.getOrDefault(s[i], 0) + 1);
    }

    ArrayList<Integer>[] buckets = new ArrayList[s.length + 1];
    for(int i=0; i<buckets.length; i++){
        buckets[i] = new ArrayList<>();
    }

    for(Integer ch: freq.keySet()){
        int val = freq.get(ch);
        buckets[val].add(ch);
    }

    int[] res = new int[k];
    int idx = 0;

    for(int b=buckets.length-1; b>=0; b--){
        for(Integer ch: buckets[b]){
            res[idx++] = ch;
            if(idx == k) return res;
        }
    }
    return res;
}
```

C++ Code

```cpp
// Bucket Sort
// First Find Frequency of All Elements using Hashmap O(N) Extra Space.
// Now Create N + 1 buckets for where ith bucket will contain elements with freq = i
// Since Frequency can vary from 0 to N, hence N + 1 buckets.
// Hence, another O(N) Extra Space

// Now Starting with last bucket, keep inserting elements, until size becomes k.
// Hence Time Complexity = O(N) for Filling hashmap + O(N) for filling buckets
// + O(K) for taking out elements in decreasing frequency.
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        vector<int> res;
        int n = nums.size();
        if (n == 0) return res;

        unordered_map<int, int> freq;
        for (auto i : nums) freq[i]++;

        vector<vector<int>> bucket(n + 1);
        for (auto &p: freq)
            bucket[p.second].push_back(p.first);


        for (int i=n; i>=0; i--)
        {
            for (auto &e: bucket[i])
            {
                res.push_back(e);
                if (res.size() == k) return res;
            }
        }

        return res;
    }
};
```

692 Java code

```java
public List<String> topKFrequent(String[] s, int k) {
    HashMap<String, Integer> freq = new HashMap<>();
    for(int i=0; i<s.length; i++){
        freq.put(s[i], freq.getOrDefault(s[i], 0) + 1);
    }

    TreeSet<String>[] buckets = new TreeSet[s.length + 1];
    for(int i=0; i<buckets.length; i++){
        buckets[i] = new TreeSet<>();
    }

    for(String word: freq.keySet()){
        int val = freq.get(word);
        buckets[val].add(word);
    }

    List<String> res = new ArrayList<>();

    for(int b=buckets.length-1; b>=0; b--){
        for(String word: buckets[b]){
            res.add(word);
            if(res.size() == k) return res;
        }
    }
    return res;
}
```

C++ Code

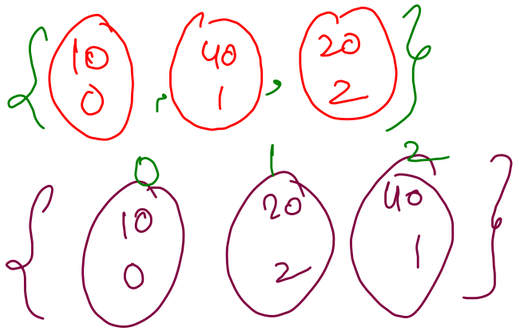
```cpp
static bool Compare(pair<string,int> p1, pair<string,int> p2)
{
    if(p1.second>p2.second || (p1.second==p2.second && p1.first<p2.first))
        return true;
    return false;
}
vector<string> topKFrequent(vector<string>& words, int k) {
    unordered_map<string,int> m;

    for(int i=0; i<words.size(); i++)
        m[words[i]]++;

    vector<pair<string,int>> arr;
    for(auto it=m.begin(); it!=m.end(); it++)
        arr.push_back({it->first,it->second});

    sort(arr.begin(),arr.end(),Compare);

    vector<string> ans;
    for(int i=0; i<k; i++)
        ans.push_back(arr[i].first);

    return ans;
}
```

Coordinate Compression

Given an array with **N** distinct elements, convert the given array to a reduced form where all elements are in range from **0** to **N-1**. The order of elements is same, i.e., **0** is placed in place of smallest element, **1** is placed for second smallest element, ... **N-1** is placed for largest element.

Expected Time Complexity: O(N*logN)
Expected Auxiliary Space: O(N)

Constraints:
$1 \le N \le 10^5$
$1 \le Arr[i] \le 10^6$

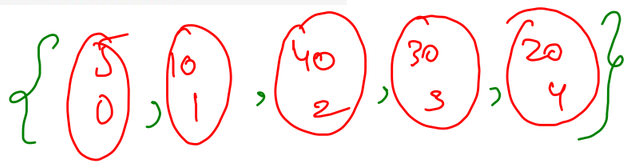


Example ①

Input:
N = 3
Arr[] = {10, 40, 20}
Output: 0 2 1
Explanation: 10 is the least element so it is replaced by 0. 40 is the largest element so it is replaced by 3-1 = 2. And 20 is the 2nd smallest element so it is replaced by 1.

Example ②

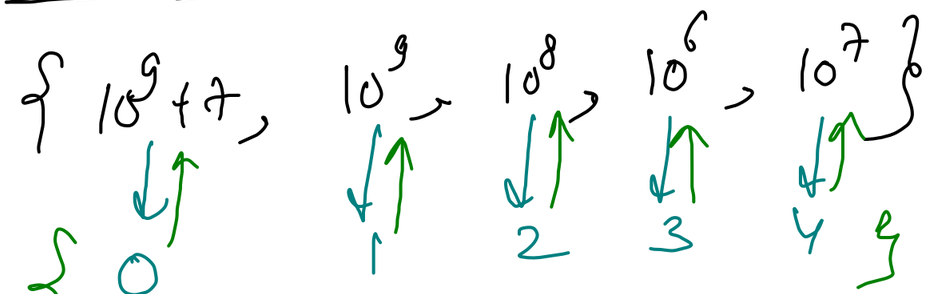
Input:
N = 5
Arr[] = {5, 10, 40, 30, 20}
Output: 0 1 4 3 2
Explanation: As 5 is smallest element, it's replaced by 0. Then 10 is 2nd smallest element, so it's replaced by 1. Then 20 is 3rd smallest element, so it's replaced by 2. And so on.




```
static class Pair implements Comparable<Pair>{
    int val;
    int idx;

    Pair(int val, int idx){
        this.val = val;
        this.idx = idx;
    }

    public int compareTo(Pair other){
        return this.val - other.val;
    }
}
```

```
void convert(int[] arr, int n) {
    Pair[] sorted = new Pair[n];
    for(int i=0; i<n; i++){
        sorted[i] = new Pair(arr[i], i);
    }
    Arrays.sort(sorted);

    for(int i=0; i<n; i++){
        arr[sorted[i].idx] = i;
    }
}
```

★Prime factors



H Index (274)

Given an array of integers `citations` where `citations[i]` is the number of citations a researcher received for their `i^th` paper, return compute the researcher's `h` **-index**.

According to the definition of h-index on Wikipedia: A scientist has an index `h` if `h` of their `n` papers have at least `h` citations each, and the other `n - h` papers have no more than `h` citations each.

If there are several possible values for `h`, the maximum one is taken as the `h` **-index**.

{ 0, 1, 2, 3, 4 }
{ 3, 0, 6, 2, 5 }

0, 2, 3, 5, 6

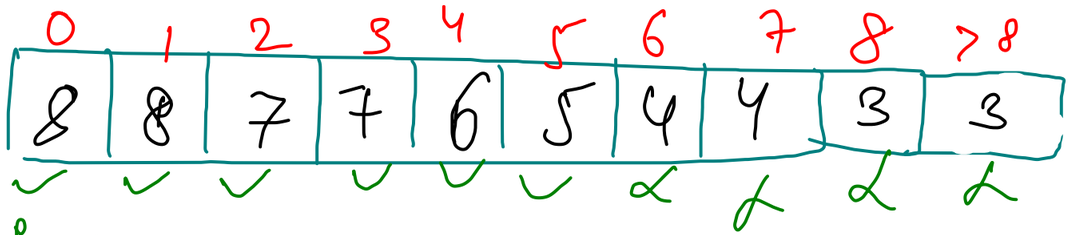
ans = 3

h index ⇒ maximum { atleast h papers having value ≥h }

example ②

{ 1, 3, 4, 5, 7, 9, 10, 11 } 8 elements

T T T T F F F F



h index

① >n extra bucket.
② Suffix array

TC ⇒ O(N)
SC ⇒ O(N)

```java
public int hIndex(int[] citations) {
    int[] buckets = new int[citations.length + 2];
    for(int val: citations){
        if(val > citations.length){
            buckets[buckets.length - 1]++;
        } else {
            buckets[val]++;
        }
    }

    for(int i=buckets.length-2; i>=0; i--){
        buckets[i] += buckets[i + 1];
        if(buckets[i] >= i){
            return i;
        }
    }
    return 0;
}
```

O(N)

O(N)