Two pointer approach

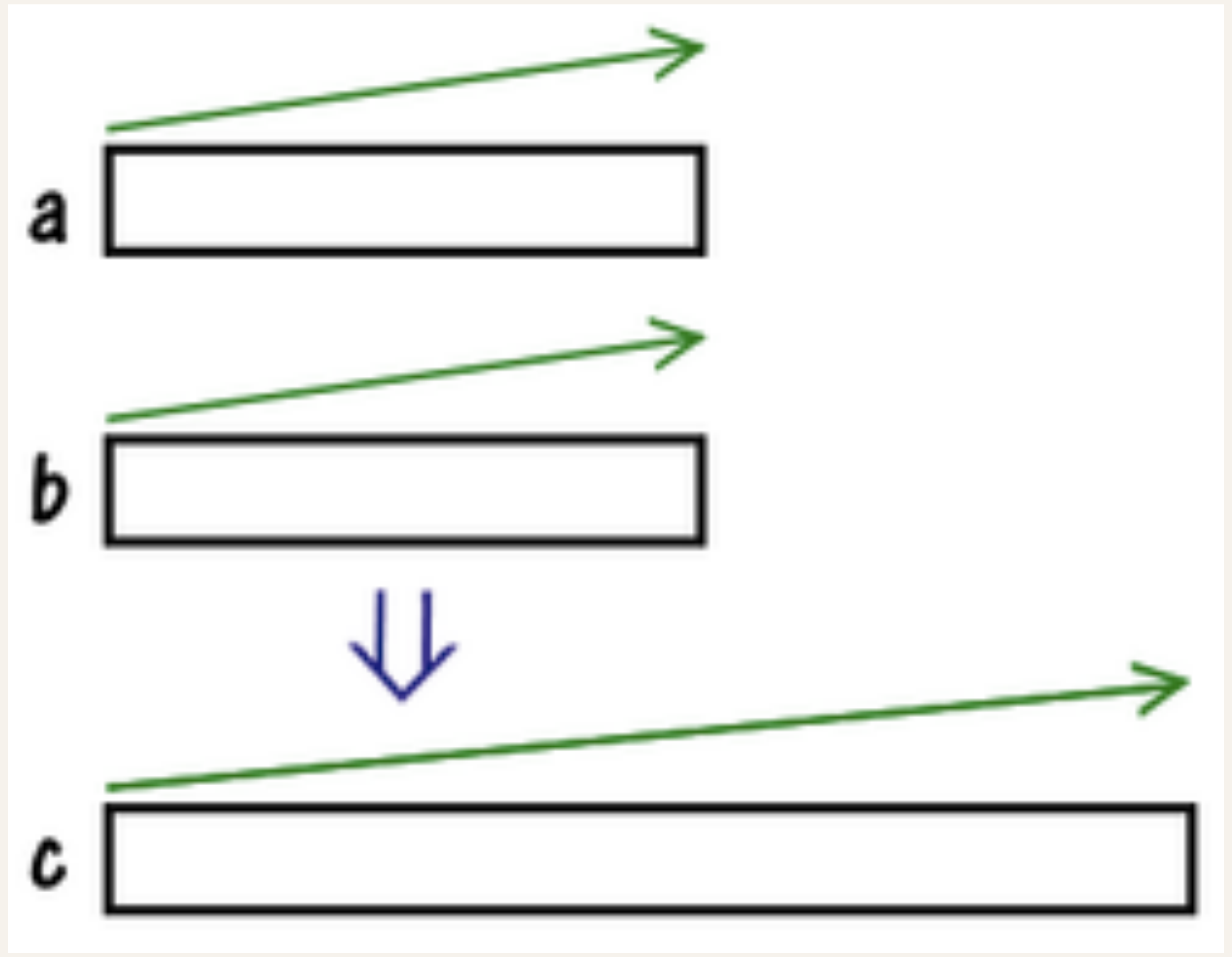
Ajay Kumar

Date

Content

Two Pointer Approach

- Merging Two Arrays
- We are given two sorted array a and b we want to find the third array which is formed by merging the two array in ascending order
- Naive method O(NlogN) where N= (a+b) by using binary search
- Where as two pointer approach gives the result in O(n+m) time



Code

```
while(i<a.size()||j < b.size())
    if a[i] < b[j]{
       c[i + j] = a[i]
       i++
    }
    else:
       c[i + j] = b[j]
       j++
```

- a=[1 3 5 8 10]
- b=[2 6 7 9 13]
- c=[1 2 3 5 6 7 8 9 10 13]
- And add the +INF to last of the both array

Number of smaller element

Problem

1. We have two arrays a and b. We want to know for each element b_j how many element such i such that $a_i<b_j$

How to solve?

First way

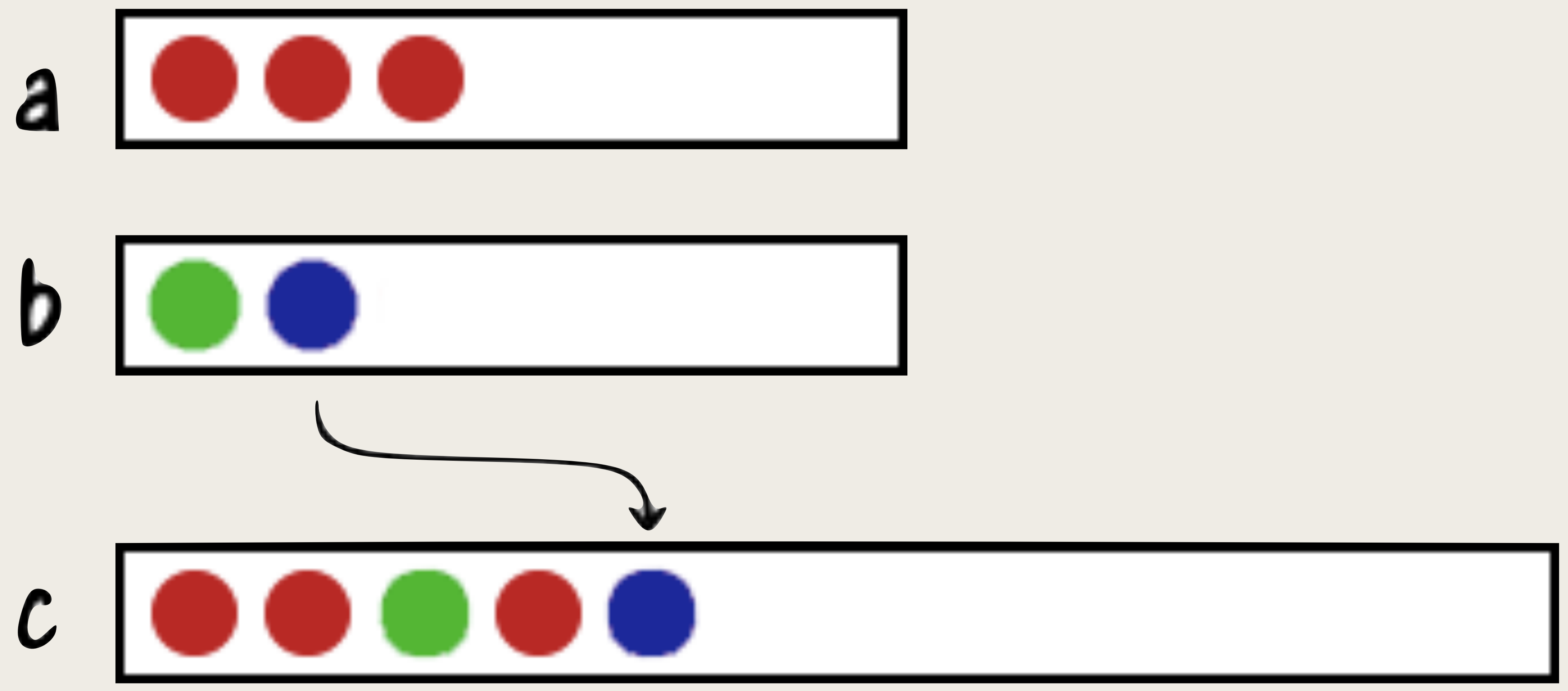
1) Sort the array(if they are unsorted) and

2) Apply the binary search to get the result

Second way

1) use two-pointer approach and

2) Merge the array to get the third array



```
i = 0
for j = 0..b.size()-1: //iterate for each j in b
    while i < a.size() && a[i] < b[j]: //move the counter I till the condition is
valid
    i++
    res[j] = i    //store the value of I’s in the resulutant vector
```

No of Inversion Count

- Problem statement: Given an array **a[]**. The task is to find the inversion count of **a[]**. Where two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

- **Input:** arr[] = {8, 4, 2, 1}
  Output: 6
  Explanation: Given array has six inversions: (8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1).

- **Input:** arr[] = {1, 20, 6, 4, 5}
  Output: 5
  Explanation: Given array has five inversions: (20, 6), (20, 4), (20, 5), (6, 4), (6, 5).

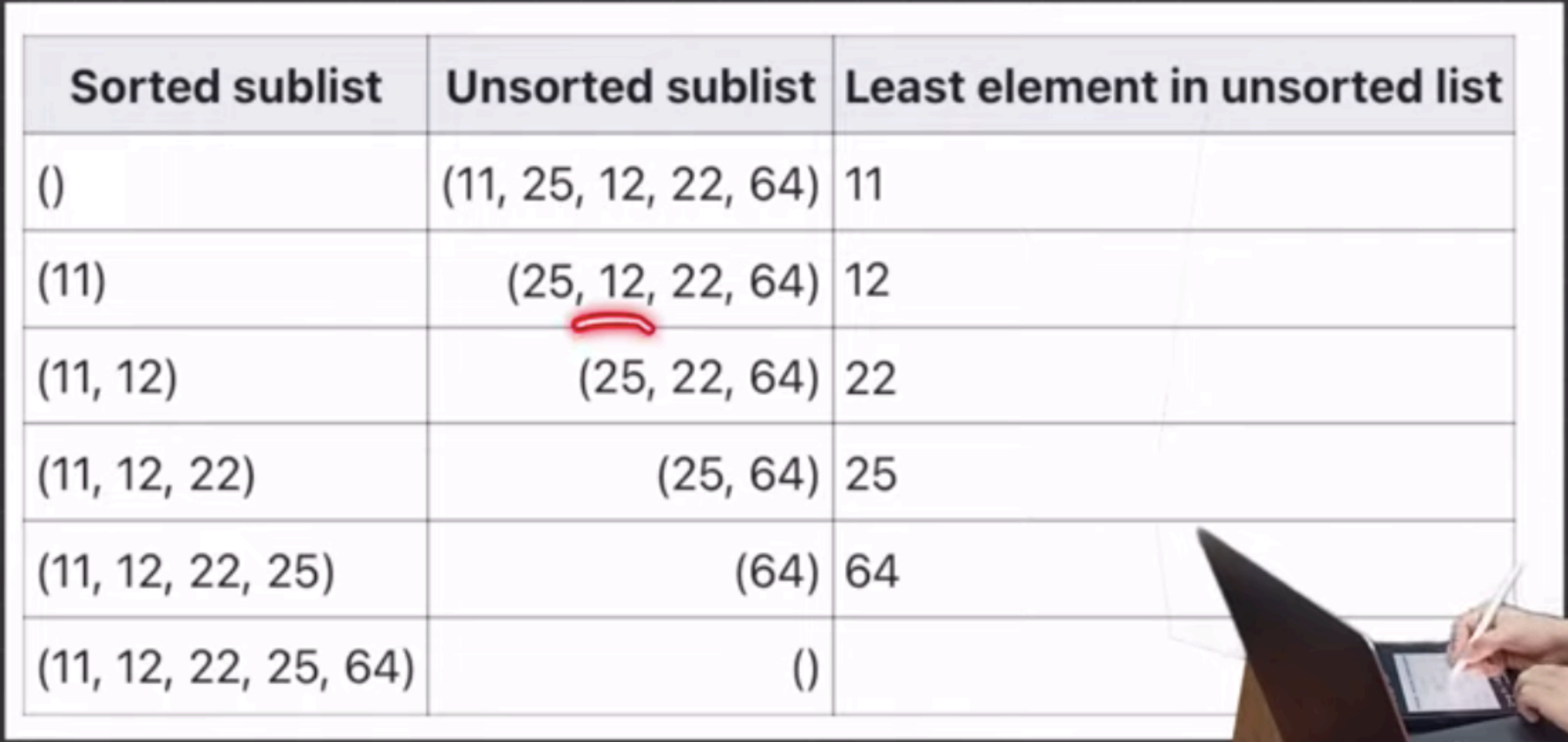
-

Selection sort

- Pick the minimum element from the array
- For each step find the minimum element from the remaining array
- And update the ith element by swapping with the minimum element from the array

```
arr = {11 25 12 13 64};
for(int i=0;i<n;i++){
    int minIndex=i;
    for(int j=i+1;j<n;j++){
        if(arr[j]<arr[minIndex]){
            minIndex=i;
        }
    }
    swap(arr[i],arr[minIndex]);
}
```

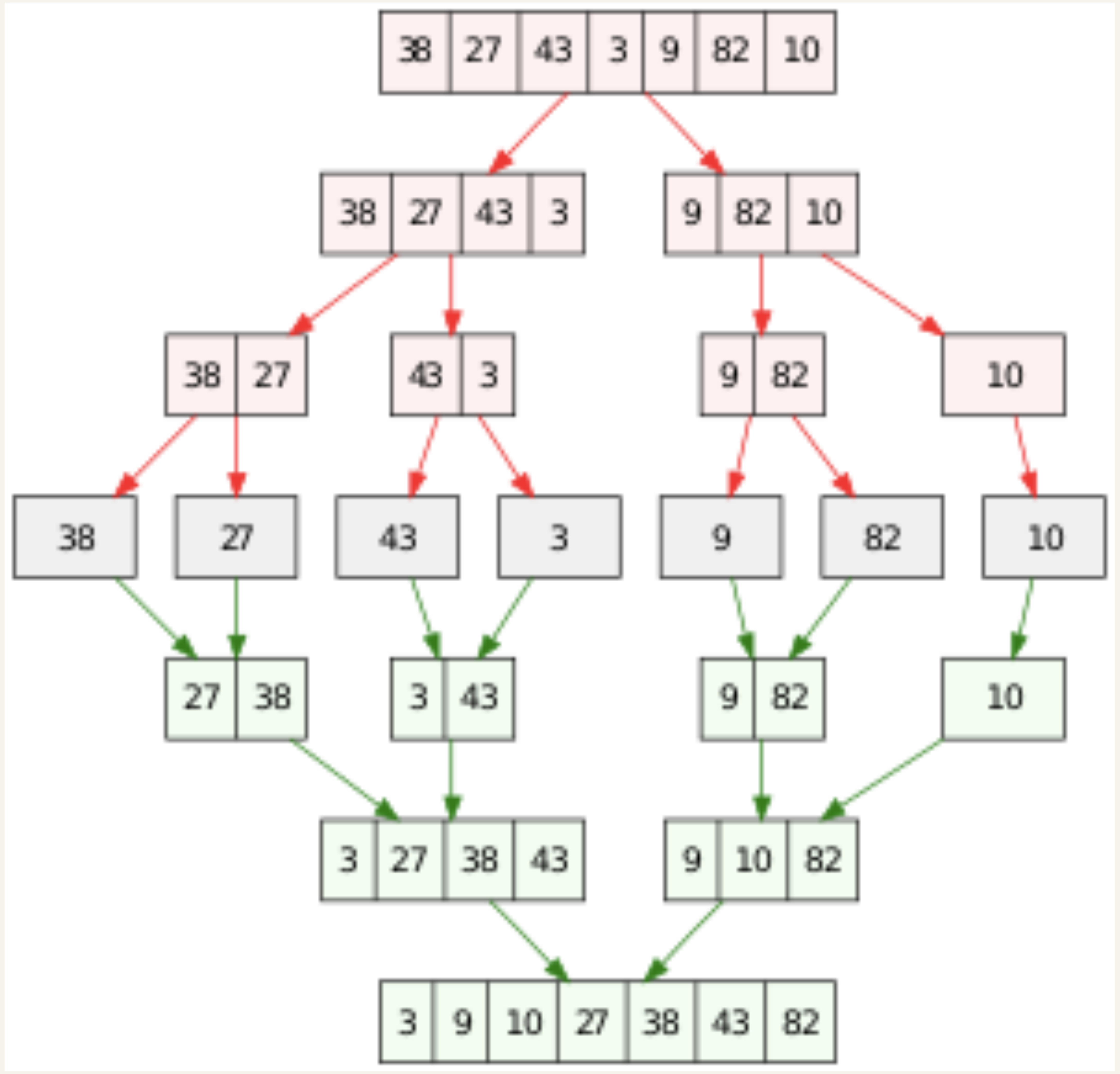
Selection Sort



Sorted sublist	Unsorted sublist	Least element in unsorted list
()	(11, 25, 12, 22, 64)	11
(11)	(25, 12, 22, 64)	12
(11, 12)	(25, 22, 64)	22
(11, 12, 22)	(25, 64)	25
(11, 12, 22, 25)	(64)	64
(11, 12, 22, 25, 64)	()	

Image src: Wikipedia

Merge sort



- Based on divide and conquer algorithm. Basically we have to divide the into subpart untill we got in two element and then sort
- Then recombine then using two pointer techniques and
- Overall complexity O(nlogn)

```
void merge(int l,int r,int mid){
    int len1 =mid-l+1;                    //here len1 is more than len2 if total length is odd
    int L[len1+1];
    int len2=r-mid;
    int R[len2 +1];
    for(int i=0;i<len1;i++){
        L[i]=a[i+l];
    }
    for(int i=0;i<len2;i++){
        R[i]=a[i+mid+1];
    }
    L[l_sz]=R[r_sz]=INT_MAX;
    int l_i=0;
    int r_i=0;
    for(int i=l;i<=r; ++i){
        if(L[l_i]<=R[r_i]){
            a[i]=L[l_i];
        }else {
            a[i]=R[r_i];
            r_i++;
        }
    }
}
```

```
void mergeSort(int l,int r){
    if(l==r) return;
    int mid=(l+r)/2;
    mergeSort(l,mid);
    mergeSort(mid+1,r);
    merge(l,r,mid);
}
```

Count the no of subarray with given xor

- Given the problem I face on this

Input : arr[] = {4, 2, 2, 6, 4}, m = 6
Output : 4
Explanation : The subarrays having XOR of
their elements as 6 are {4, 2},
{4, 2, 2, 6, 4}, {2, 2, 6},
and {6}

Input : arr[] = {5, 6, 7, 8, 9}, m = 5
Output : 2
Explanation : The subarrays having XOR of
their elements as 5 are {5}
and {5, 6, 7, 8, 9}

Brute Force Approach (Try out every possible value of I,j)

```
long long subarrayXor(int arr[], int n, int m)
{
    long long ans = 0; // Initialize ans

    // Pick starting point i of subarrays
    for (int i = 0; i < n; i++) {
        int xorSum = 0; // Store XOR of current subarray

        // Pick ending point j of subarray for each i
        for (int j = i; j < n; j++) {
            // calculate xorSum
            xorSum = xorSum ^ arr[j];

            // If xorSum is equal to given value,
            // increase ans by 1.
            if (xorSum == m)
                ans++;
        }
    }
    return ans;
}
```

Time complexity : O(n^2)

- Efficient Approach: An **Efficient Solution** solves the above problem in O(n) time.
- Let us call the XOR of all elements in the range [i+1, j] as A, in the range [0, i] as B, and in the range [0, j] as C.
- If we do XOR of B with C, the overlapping elements in [0, i] from B and C zero out, and we get XOR of all elements in the range [i+1, j], i.e. A. Since A = B XOR C, we have B = A XOR C.
- Now, if we know the value of C and we take the value of A as m, we get the count of A as the count of all B satisfying this relation. Essentially, we get the count of all subarrays having XOR-sum m for each C. As we take the sum of this count overall C, we get our answer.

Pseudo code

```
1) Initialize ans as 0.
2) Compute xorArr, the prefix xor-sum array.
3) Create a map mp in which we store count of all prefixes with XOR as a particular value.
4) Traverse xorArr and for each element in xorArr
    (A) If m^xorArr[i] XOR exists in map, then there is another previous prefix with same XOR,
    i.e., there is a subarray ending at i with XOR equal to m. We add count of all such subarrays to
    result.
   (B) If xorArr[i] is equal to m, increment ans by 1.
   (C) Increment count of elements having XOR-sum
      xorArr[i] in map by 1.
5) Return ans.
```

```
long long subarrayXor(int arr[], int n, int m)
{
    long long ans = 0; // Initialize answer to be returned
    int* xorArr = new int[n];
    unordered_map<int, int> mp;
    // Initialize first element of prefix array
    xorArr[0] = arr[0];
    // Computing the prefix array.
    for (int i = 1; i < n; i++)
        xorArr[i] = xorArr[i - 1] ^ arr[i];

    // Calculate the answer
    for (int i = 0; i < n; i++) {

        // Find XOR of current prefix with m.
        int tmp = m ^ xorArr[i];
        ans = ans + ((long long)mp[tmp]);

        // If this subarray has XOR equal to m itself.
        if (xorArr[i] == m)
            ans++;

        // Add the XOR of this subarray to the map
        mp[xorArr[i]]++;
    }
    return ans;
}
```

2D prefix sum technique

Now, what if we wanted to process Q queries for the sum over a subrecatngle of a 2D matrix with N rows and M coloumns. Let’s assume both rows and coloumns are 1-indexed and we use the following matrix as an example

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

Naively,each sum query would then take O(NM) time. for a total of O(QNM). This is too slow

Let's take the following example region. which we want to sum.Manually summing all the cells, we have a submatrix sum of 7 + 11+ 9+ 6+ 1+ 3=37

0	0	0	0	0	0
0	1	5	6	11	8
0	1	**7**	**11**	**9**	4
0	4	**6**	**1**	**3**	2
0	7	5	4	2	3

- The **first logical optimization** would be to do one-dimensional prefix sum of each row. Then we'd have the following row-prefix sum matrix.
- The desired subarray sum of each row in our desired region is simple the green cell minus the red cell in that respective row. We do this for each row to get (28-1)+ (14-4)= 37

0	0	0	0	0	0
0	1	6	12	23	31
0	🟥 1	8	19	🟩 28	32
0	🟥 4	10	11	🟩 14	16
0	7	12	16	18	21

- Now, if we wanted to find a submatrix sum. we could break up the submatrix into a subarray for each row, and then add their sums.which would be calculated using the prefix sums method described earlier. Siche the matrix has N row, the time compleixty of this is O(QN). this might be fast enough for the Q=10^5 and N= 10^3, but we can do better
- In fact,we can do two-dimesional prefix sums. In our two dimesional prefix sum array, we have

$$prefix[a][b] = \sum_{i=1}^{a} \sum_{j=1}^{b} arr[i][j]$$

This can be calculate as follow for row index 1≤i≤n and column index 1≤j≤m

$$prefix[i][j] = prefix[i-1][j] + prefix[i][j-1] - prefix[i-1][j-1] + arr[i][j]$$

- Let’s calculate prefix[2][3]

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

- The submatrix sum between rows a and A and coloumns b and B, can thus be expressed as follows:
- $$\sum_{i=a}^{A}\sum_{j=b}^{B} arr[i][j] = prefix[A][B] - prefix[a-1][B] - prefix[A][b-1] + prefix[a-1][b-1]$$

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

add prefix[A][B]

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

Subtract prefix[a-1][B]

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

Subtract prefix[A][b-1]

0	0	0	0	0	0
0	1	5	6	11	8
0	1	7	11	9	4
0	4	6	1	3	2
0	7	5	4	2	3

Add prefix[a-1][b-1]

- Since no matter the size of the submatrix we are summing, we only need to access four values of the 2D prefix sum array, this run O(1) query after the an O(NM) preprocessing.
- Warning : we need to be cautious of off-by-one errors, an intervals can be inclusive, exclusive , 1-indexed, etc.
-