Sorting Algorithms

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- Formally
  - Input: A sequence of n numbers <a1,a2,…,an>
  - Output: A reordering <a'1,a'2,…,a'n> of the sequence such that $a'1 \leq a'2 \leq \ldots \leq a'n$
  - Given the input <6, 3, 1, 7>, the algorithm should produce <1, 3, 6, 7>
    - Called an instance of the problem

Sorting Algorithms

- Some of sorting algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
- These are among the most fundamental sorting algorithms

Bubble Sort

- Elements of the array are stored in a[0 .. n-1]
- Repeatedly
  - Compare every pair of adjacent elements.
  - swap adjacent elements that are out of order.
- After one iteration, maximum/minimum element moves to the last position of the list like a bubble moving up to the top in boiling water.
- Repeat the process n-1 times to sort entire list.
- In each iteration n-1 pairs of adjacent elements are to be compared and swapped whenever necessary.
- Can we reduce the number of pairs elements to be compared?

Bubble Sort

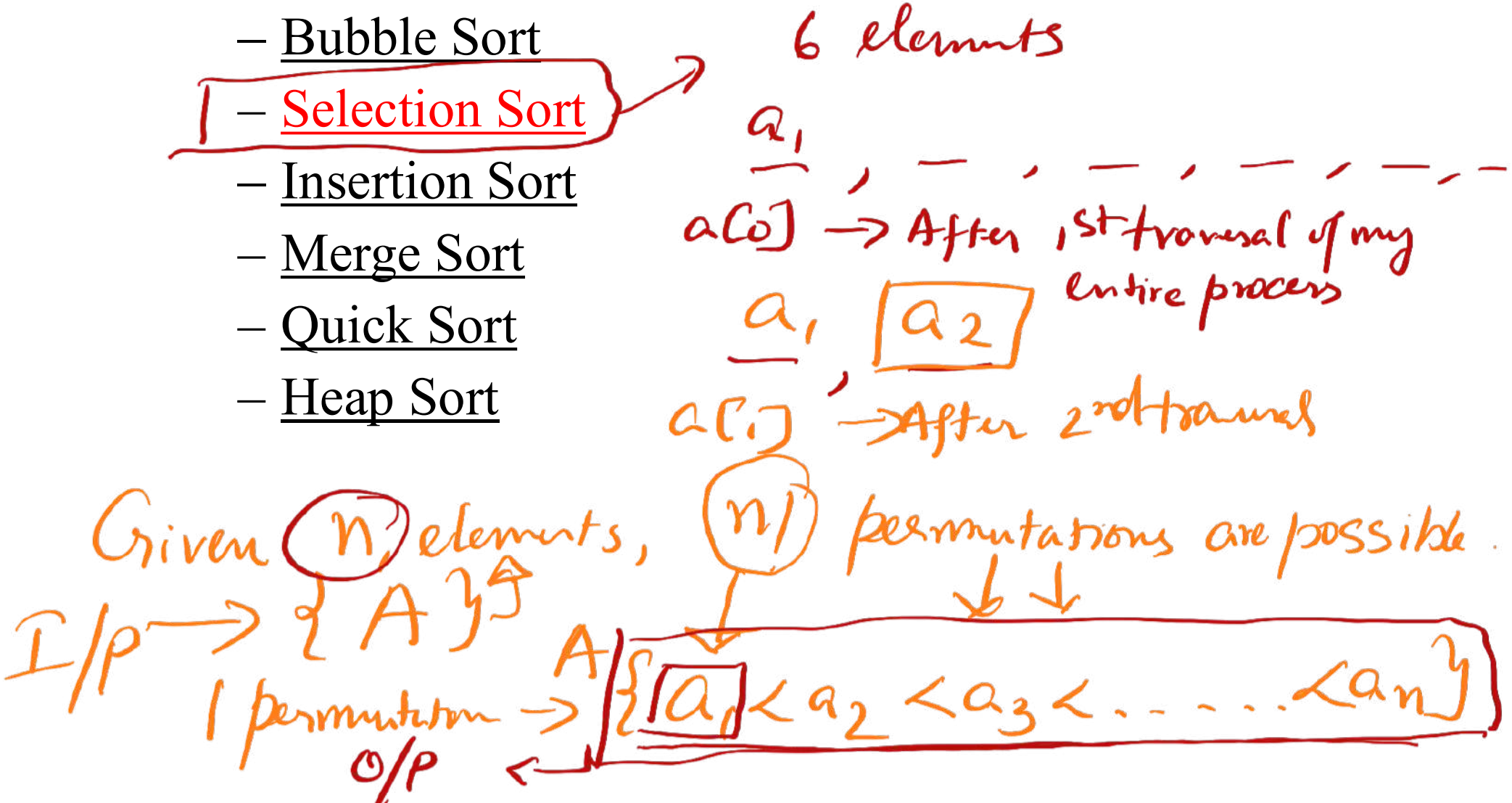
23	78	45	8	32	56
23	45	78	8	32	56
23	45	8	78	32	56
23	45	8	32	78	56
23	45	8	32	56	78
23	8	45	32	56	78
23	8	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78

Bubble Sort Algorithm -1

```
template <class Item>
void bubleSort(Item a[], int n)
{
      for (int i = 0; i < n; i++)
      {
            for (int j = 0; j < n; j++)
            if (a[j] > a[j+1])
              swap(a[j],a[j+1]);
      }
}
```

Sorting Algorithms

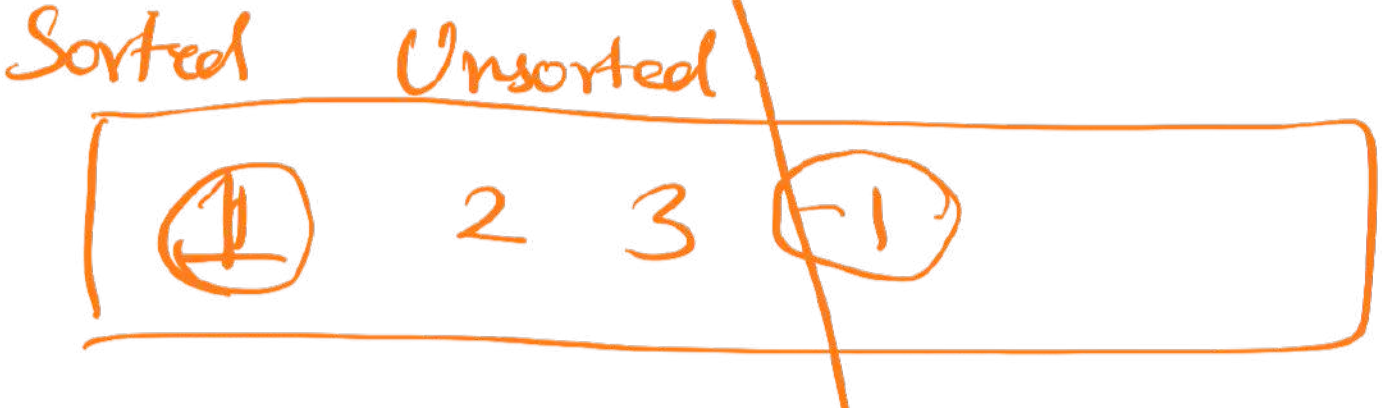
- Some of sorting algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort

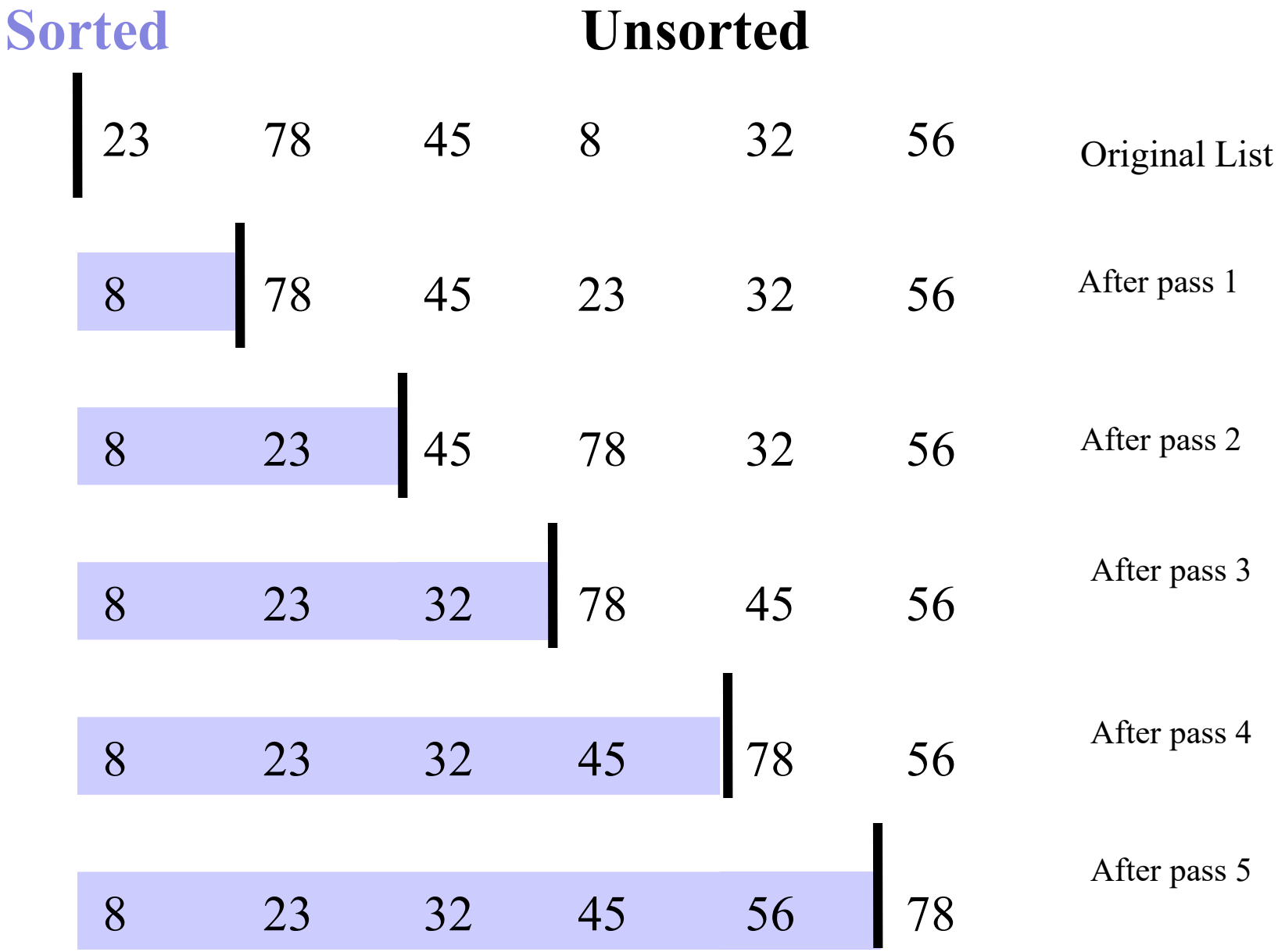


Selection Sort

- Partition the input list into a sorted and unsorted part (initially sorted part is empty)
- **Select** the smallest element in the unsorted list and put it to the end of the sorted part

- Increase the size of the sorted part by one
- Repeat this n-1 times to sort a list of n elements




Sorted
Unsorted
23 78 45 8 32 56
Original List
8 78 45 23 32 56
After pass 1
8 23 45 78 32 56
After pass 2
8 23 32 78 45 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5

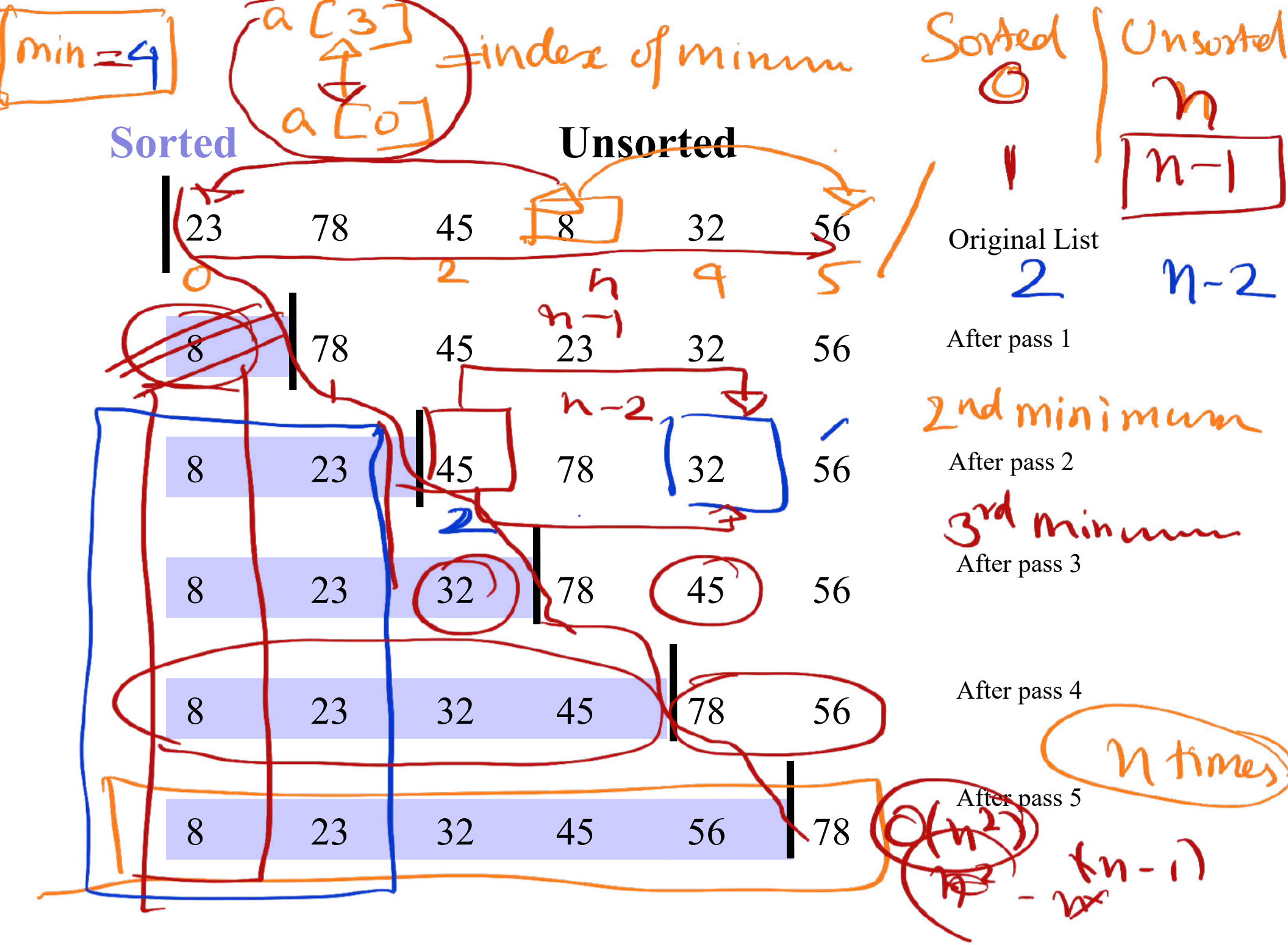
Selection Sort Algorithm

```
void selectionSort(Item a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
    {
       if (a[j] < a[min])
             min = j;
    }
    if (min!=i)
    {
    swap(a[i], a[min]);
    }
  }
}
```

Selection Sort -- Complexity

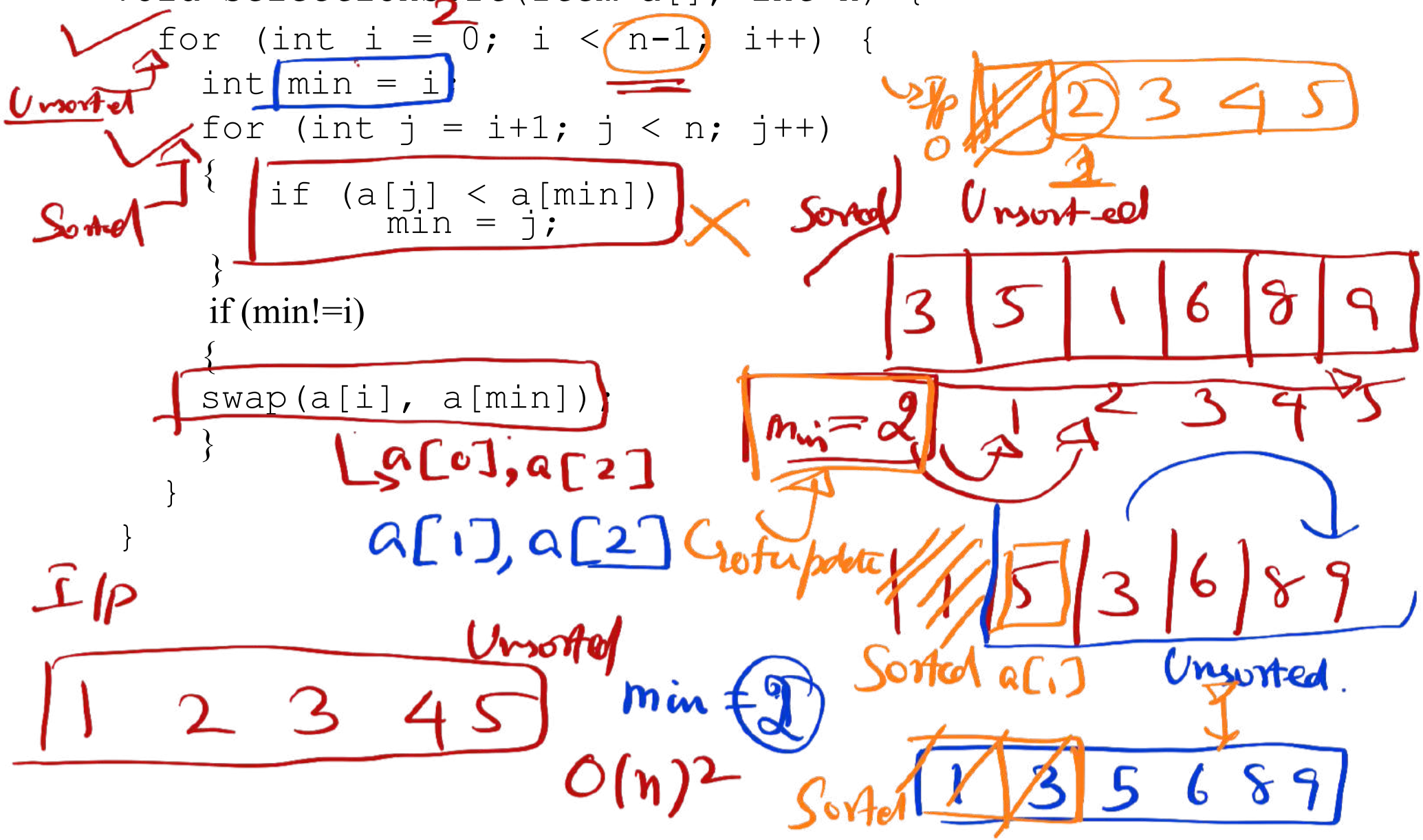
- What is the complexity of selection sort?
- What are best, average, and worst case complexities?

- It is $O(n^2)$ for all cases!


min = 4
a[3]
a[0]
= index of minm
Sorted
Unsorted
Sorted
Unsorted
0
n
1
n-1
23 78 45 8 32 56
Original List
0 2 4 4 5
2
n-2
8 78 45 23 32 56
After pass 1
n-1
2nd minimum
n-2
8 23 45 78 32 56
After pass 2
2
3rd minimum
8 23 32 78 45 56
After pass 3
8 23 32 45 78 56
After pass 4
n times
8 23 32 45 56 78
After pass 5
O(n²)
n²
(n-1)
- n

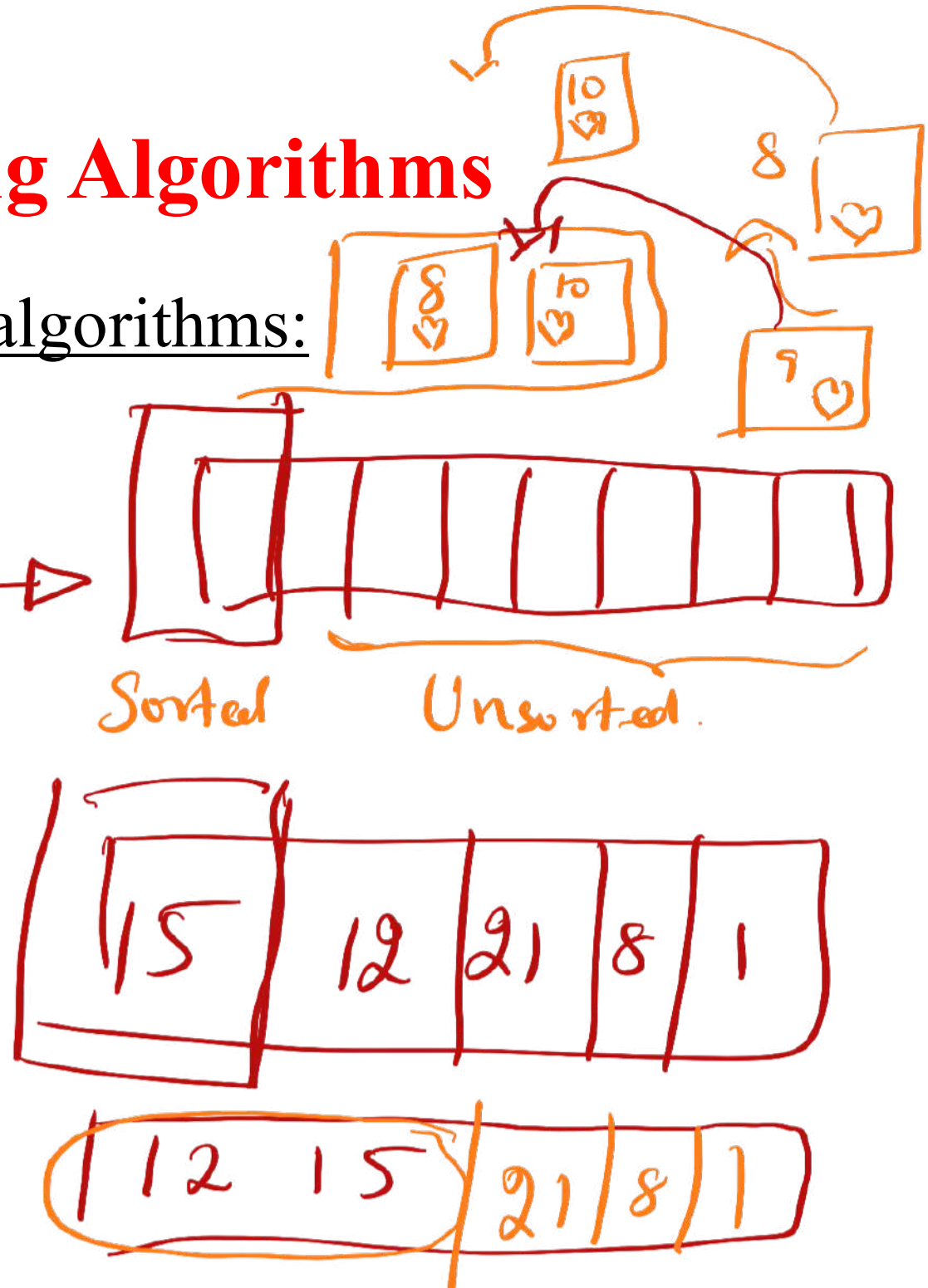
Selection Sort Algorithm

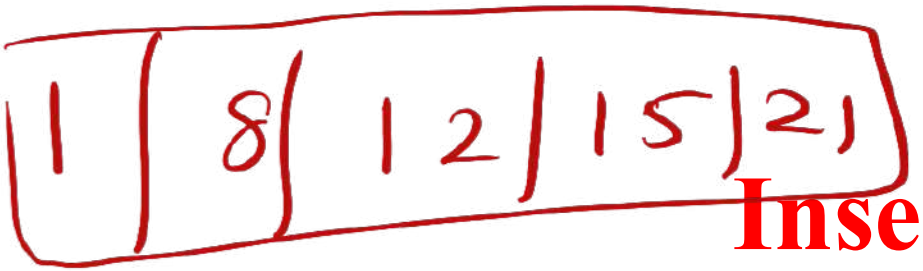
```
void selectionSort(Item a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
    {
      if (a[j] < a[min])
        min = j;
    }
    if (min!=i)
    {
    swap(a[i], a[min]);
    }
  }
}
```

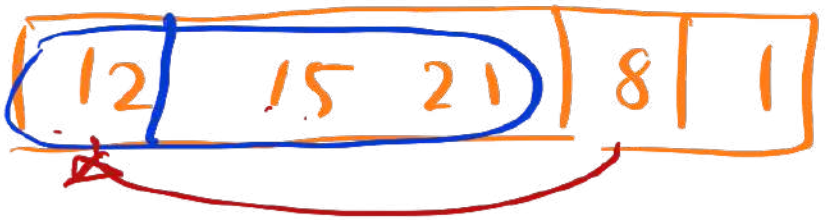


Sorting Algorithms

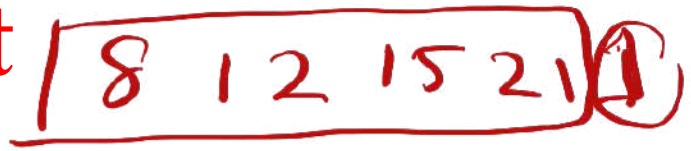
- Some of sorting algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort



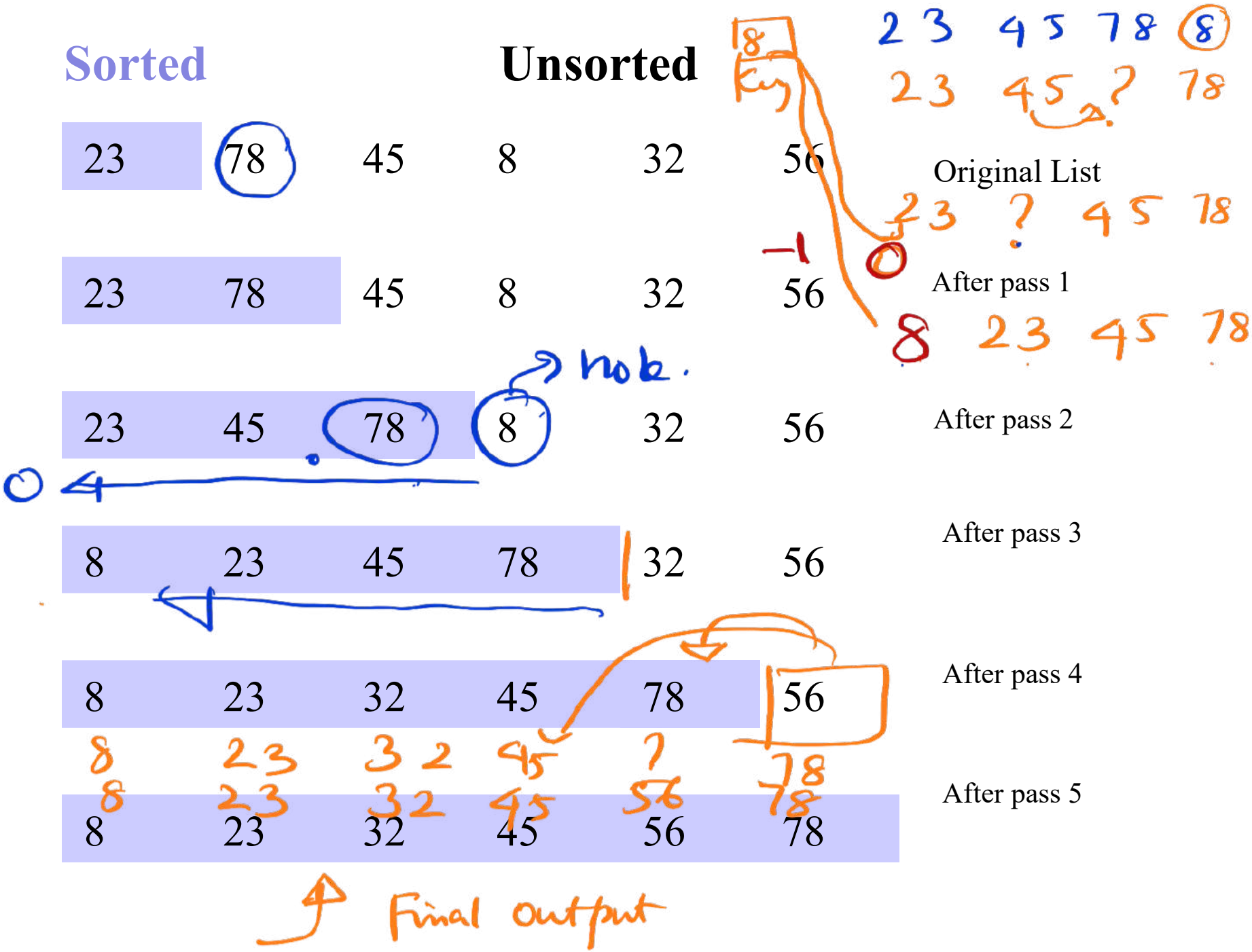




Insertion Sort



- Insertion sort is a simple sorting algorithm that is appropriate for small size inputs.
- Again, the list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and **inserted** at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

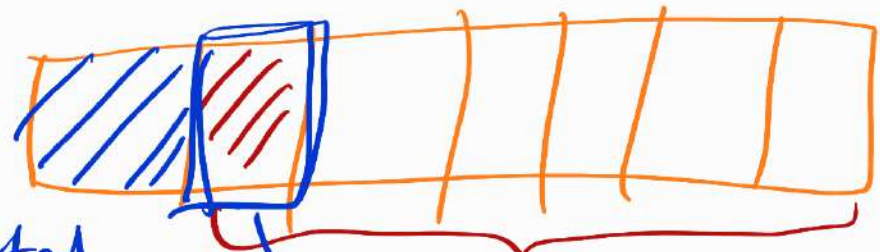

Sorted
Unsorted
23 78 45 8 32 56
Original List
23 78 45 8 32 56
After pass 1
23 45 78 8 32 56
After pass 2
8 23 45 78 32 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5
Final output

Insertion Sort – Complexity

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - [redacted].
- ***Worst-case:*** ➔ **O(n^2)**
  - [redacted]
- ***Average-case:*** ➔ **O(n^2)**
  - [redacted].

Insertion Sort – Complexity

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
- ***Average-case:*** ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.



4 ? 8

0 ? 4 8

3 4 8

Insertion Sort:-

```
for (i=1; i<n; i++)
{
    hole = a[i];
    j = i-1;
    while (j>=0 && a[j]>hole)
    {
        a[j+1] = a[j];
        j = j-1;
    }
    a[j+1] = hole;
}
```

a[j]

O(N)

1. 2. 3. 4. 5

I/p (9) | 6 5 0 8 2 ←

I^St Sorted | Unsorted.

hole

(0)

2 3 ? 4 5

I/o 0 2 3 4 5

hole = 6

9 | (?) 5 0 8 2

6 9

II^nd (?) 9 5 0 8 2

Reached j = 0; then j = -1, while loop fails.

6 9 | (5) 0 8 2

hole = 5

6 ? 9 0 8 2

III^rd ? 6 9 0 8 2

Reached j = 0; then j = -1;

5 6 9 | (0) 8 2

IV^th Same process like III^rd hole = 0

0 5 6 9 | (8) 2

hole = 8

0 5 6 ? 9 2

0 5 6 8 9 2

Vth When 6 > 8 ; Condition fails.

a[j] > hole

So no need to check for preceding elements.

0 5 6 8 9 2

hole = 2

0 5 6 8 ? 9

0 5 6 ? 8 9

VIth 0 5 ? 6 8 9

0 ? 5 6 8 9.

Now 0 > 2 ; Condition fails, So no need to

a[j] > hole

Check for preceding elements.

0 2 5 6 8 9.

0 2 5 6 8 9

Merge Sort

* External Sorting

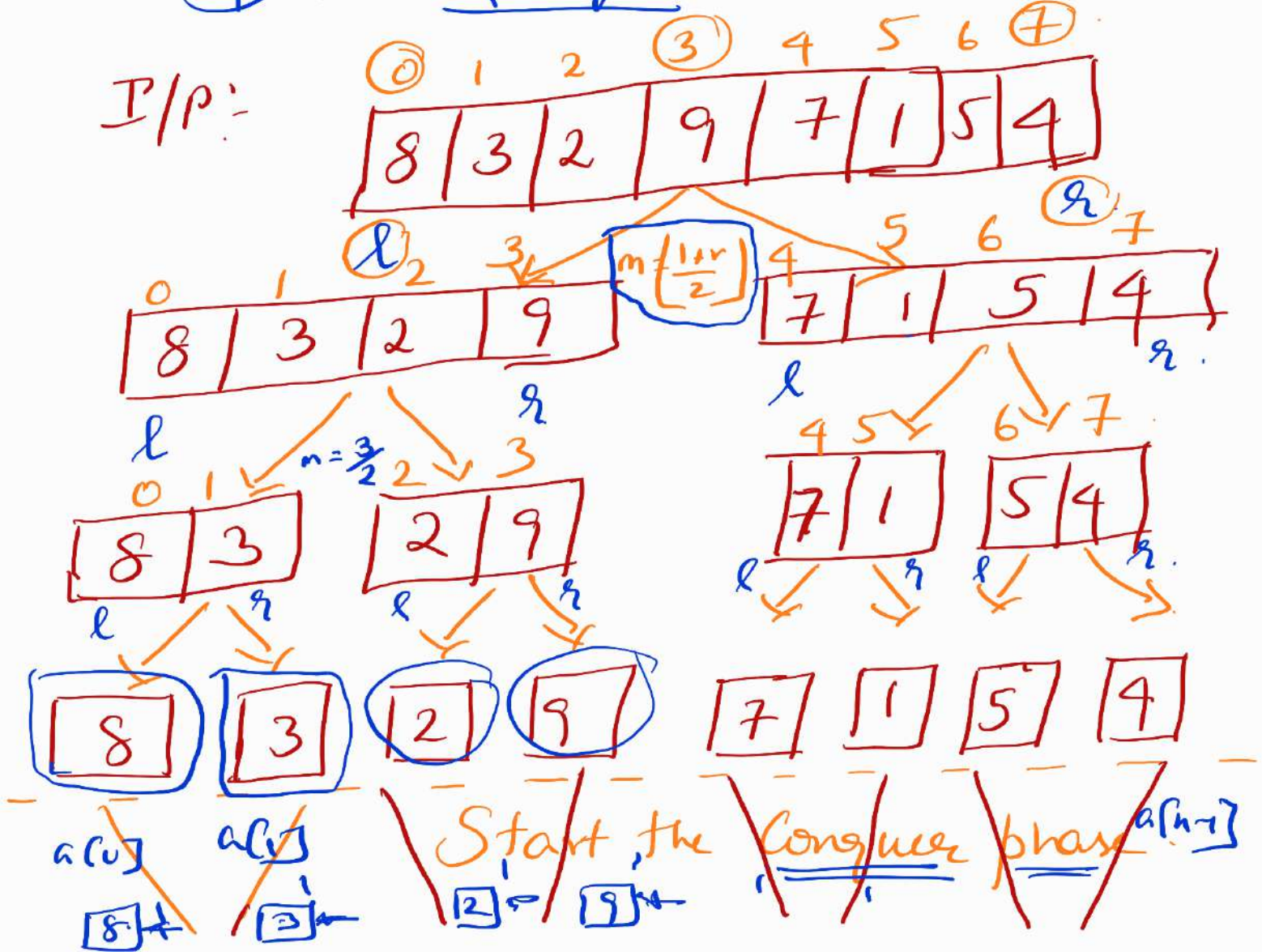
```
MergeSort ( Arr[], int l, int r )
{
    if ( l < r )
    {
        int m = ⌊(l + r) / 2⌋;

        MergeSort ( Arr, l, m )      ← left
        MergeSort ( Arr, m+1, r )    ← right
        Merge ( Arr, l, m, r );      ← Conquer
    }
}
```

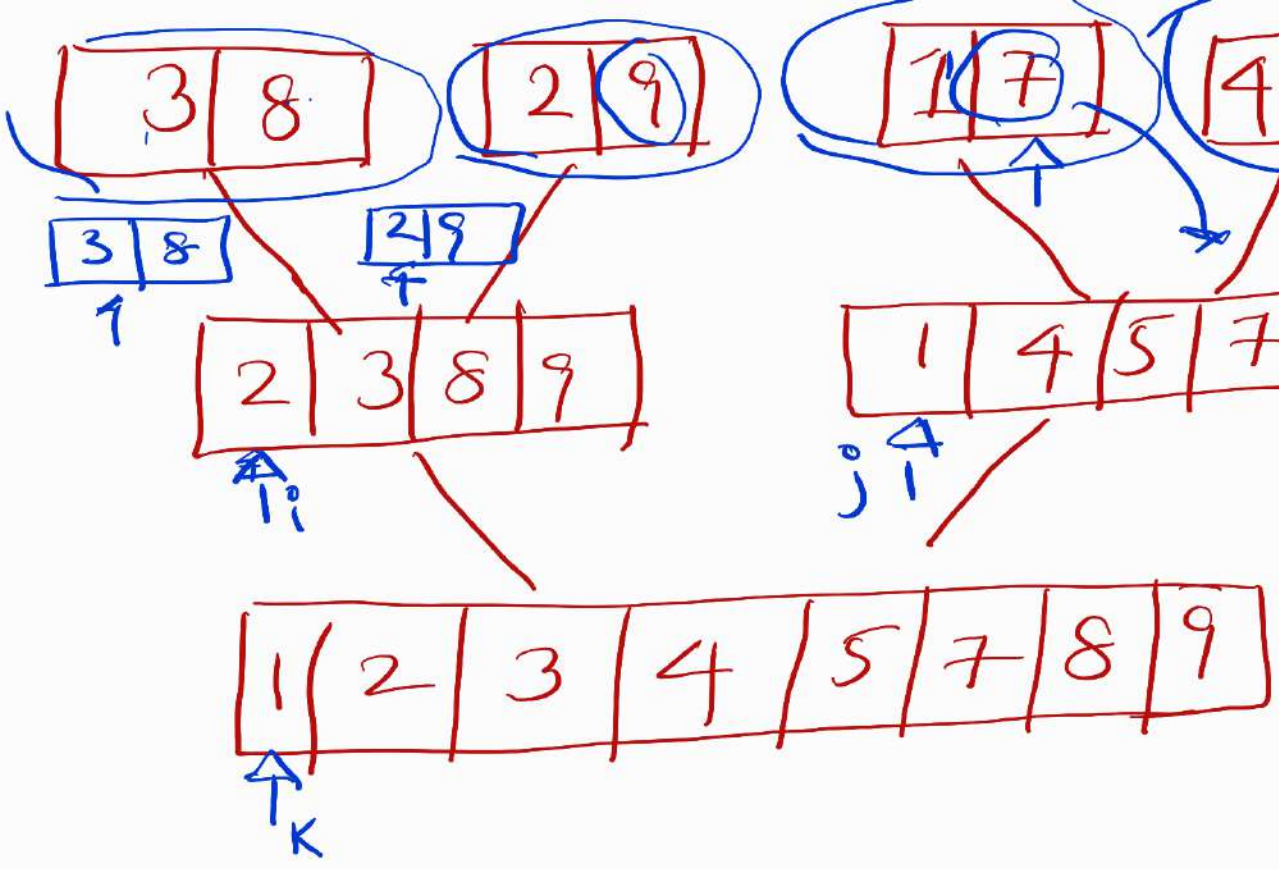
Divide

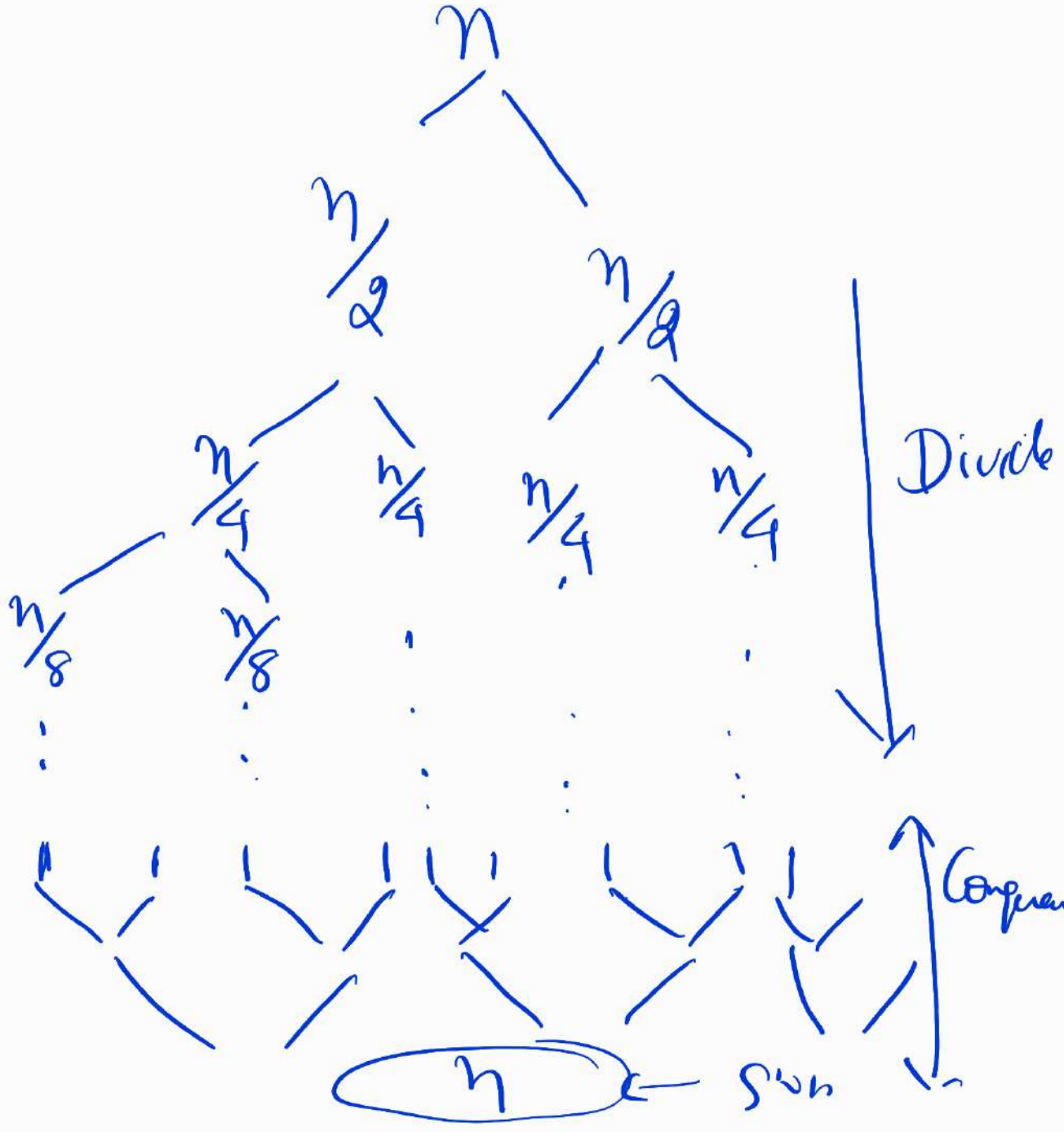
Conquer

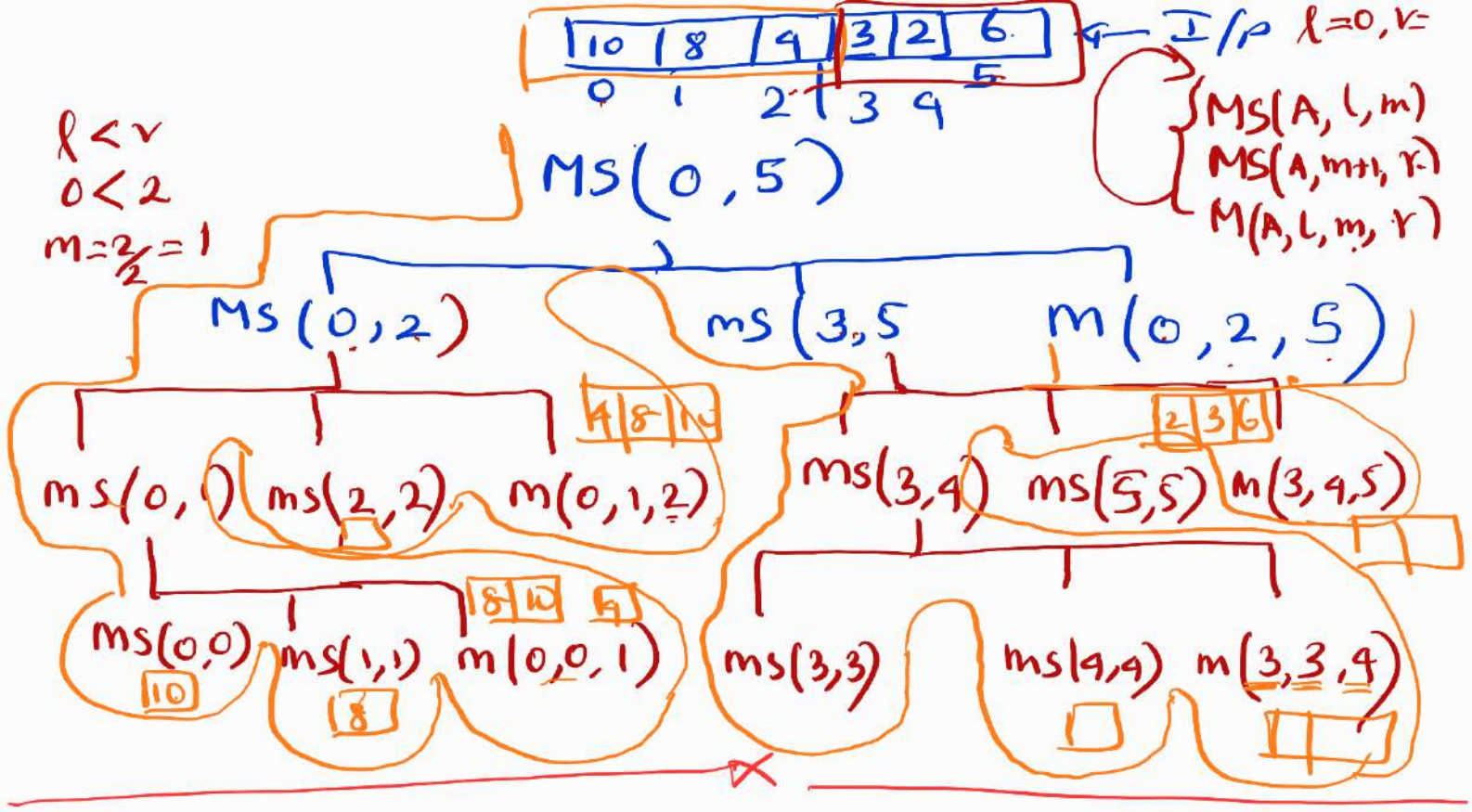
Divide & Conquer Strategy

I/P :-




3 8
2 9
1 7
4 5
3 8
2 9
i
2 3 8 9
j
1 4 5 7
1 2 3 4 5 7 8 9
k


n
n/2
n/2
n/4
n/4
n/4
n/4
n/8
n/8
Divide
Conquer
n
Sum




```
Void merge (int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1; //Size of left temp. array
    int n2 = r - m; //Size of right temp array.

From main array, put it into two temporary array L[n1], R[n2]

    for (i = 0; i < n1; i++)
        L[i] = Arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = Arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] < R[j]) {
            arr[k] = L[i];
            i++;
        }
        k++;
    }
```

```
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

Void mergesort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = ⌊ (l+r)/2 ⌋;
```

```
        mergesort(arr, l, m);
        mergesort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```