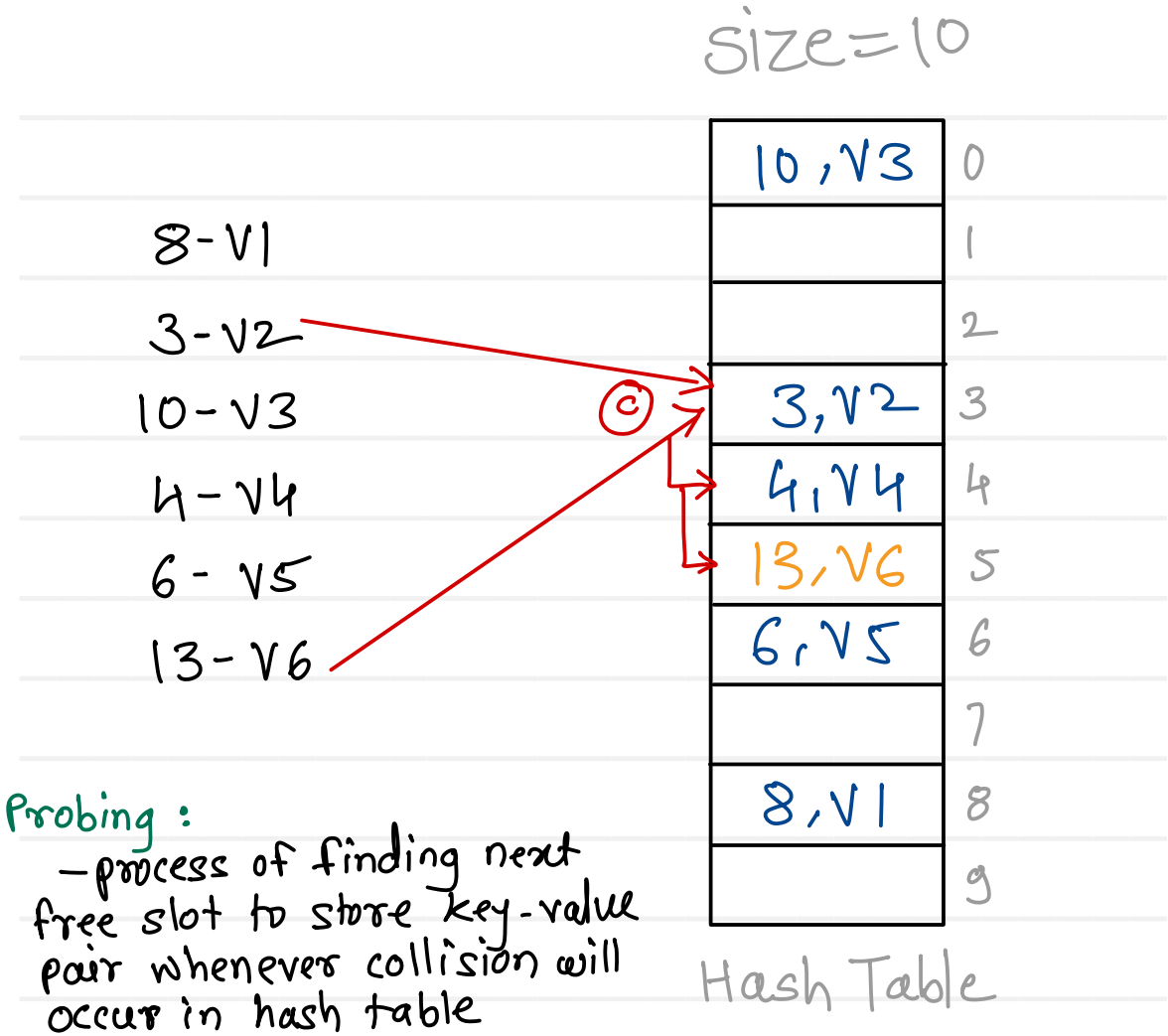
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Open addressing - Linear probing



Probing :
- process of finding next free slot to store key-value pair whenever collision will occur in hash table

h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i ← probe number

where i = 1, 2, 3, …

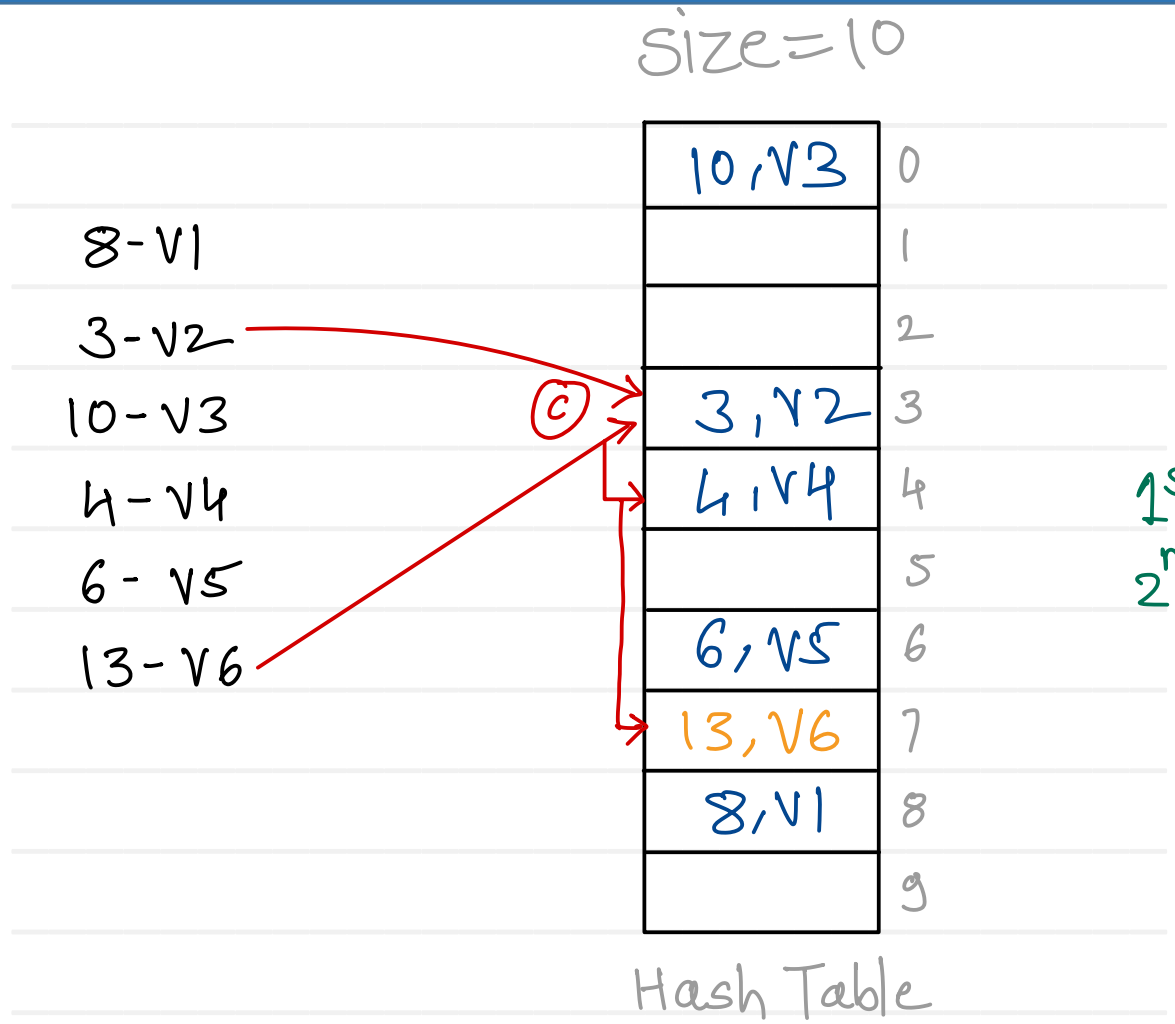
h(13) = 13 % 10 = 3 (c)

1st probe : h(13,1) = [3+1] % 10 = 4 (c)

2nd probe : h(13,2) = [3+2] % 10 = 5

Primary clustering :
- near key position table becomes crowded
- to find next empty when collision is occured need to take long run of filled slots "near" key position

Open addressing - Quadratic probing



h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i^2

where i = 1, 2, 3, ...

h(13) = 13 % 10 = 3 (c)

1st probe : h(13,1) = [3+1] % 10 = 4 (c)

2nd probe : h(13,2) = [3+4] % 10 = 7

- there is no guarantee of getting empty slot for key value pair

Secondary clustering :

- to find next empty when collision is occured need to take long run of filled slots "away" key position

Open addressing - Quadratic probing

size=10

8-V1
3-V2
10-V3
4-V4
6-V5
13-V6
23,V7
33,V8

10,V3	0
	1
23,V7	2
3,V2	3
4,V4	4
	5
6,V5	6
13,V6	7
8,V1	8
33,V8	9

Hash Table

h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i^2

where i = 1, 2, 3, …

h(23) = 23 % 10 = 3 ©

1st : h(23,1) = [3 + 1] % 10 = 4 ©

2nd : h(23,2) = [3 + 4] % 10 = 7 ©

3rd : h(23,3) = [3 + 9] % 10 = 2

h(33) = 23 % 10 = 3 ©

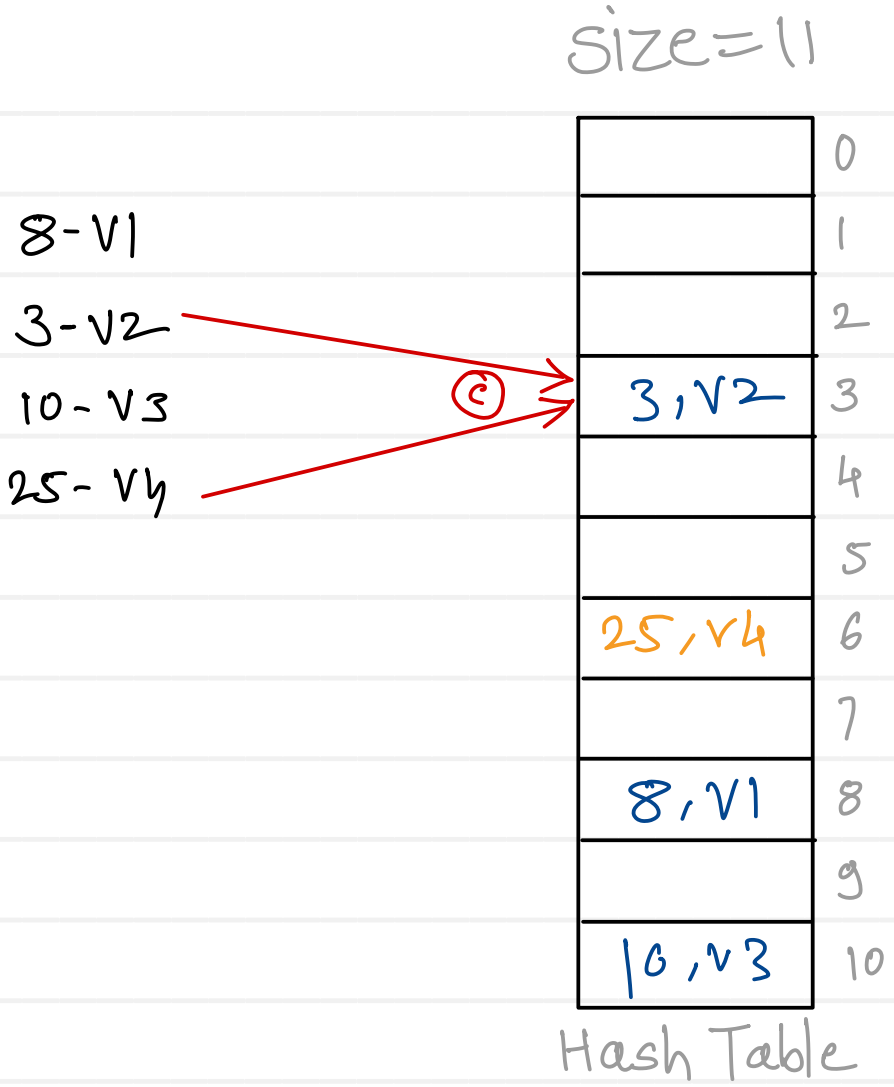
1st : h(33,1) = [3 + 1] % 10 = 4 ©

2nd : h(33,2) = [3 + 4] % 10 = 7 ©

3rd : h(33,3) = [3 + 9] % 10 = 2 ©

4th : h(33,4) = [3 + 16] % 10 = 9

Open addressing - Double hashing



h1(k) = k % size

h2(k) = 7 - (k % 7)

h(k, i) = [h1(k) + i * h2(k)] % size

h1(8) = 8 % 11 = 8

h1(3) = 3 % 11 = 3

h1(10) = 10 % 11 = 10

h1(25) = 25 % 11 = 3 (c)

h2(25) = 7 - (25 % 7) = 3

1st: h(25,1) = [3 + 1 * 3] % 11 = 6

h1(36) = 36 % 11 = 3 (c)

h2(36) = 7 - (36 % 7) = 6

1st: h(36,1) = [3 + 1 * 6] % 11 = 9

Rehashing

$$\text{Load factor} = \frac{n}{N}$$

e.g. $N = 10, n = 7$

$\lambda = \frac{7}{10} = 0.7$

hash table is 70 % filled

n - number of elements (key-value) present in hash table
N - number of total slots in hash table

- Load factor ranges from 0 to 1.
- If n < N Load factor < 1 - free slots are available
- If n = N Load factor = 1 - free slots are not available

- In rehashing, whenever hash table will be filled more than 60 or 70 % size of hash table is increased by twice
- Existing key value pairs are remapped according to new size

Two sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6

Output: [0,1]

$T(n) = O(n^2)$

$S(n) = O(1)$

```
int[] twoSum(int[] nums, int target) {
    for(i=0; i<nums.length-1; i++){
        for(j=i+1; j<nums.length; j++){
            if(nums[i]+nums[j]==target)
                return new int[]{i,j};
        }
    }
    return new int[]{};
}
```

Two sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

HashMap

key	value
2	0

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2]

HashMap

key	value
3	0
2	1

Example 3:

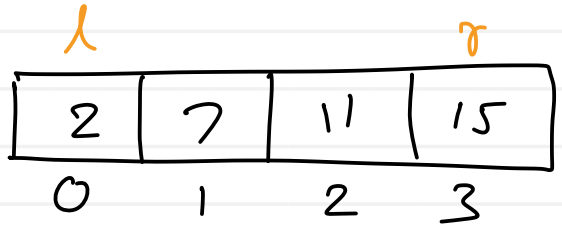
Input: nums = [3,3], target = 6

Output: [0,1]

```
int[] twoSum(int[] nums, int target){
    Map<Integer,Integer> tbl = new HashMap<>();
    for(int i=0; i<nums.length; i++){
        if(tbl.containsKey(target-nums[i]))
            return new int[]{tbl.get(target-nums[i]),i};
        tbl.put(nums[i],i);
    }
    return new int[]{};
}
```

Two pointers Technique

- The two-pointer technique is a widely used approach to solving problems efficiently, which involves arrays or linked lists.
- This method involves traversing arrays or lists with two pointers moving at different speeds or in different directions.
- This technique is used to solve problems more efficiently than using a single pointer or nested loops.



target = 9

l	r	sum
0	3	17
0	2	13
0	1	9

- Find a pair of elements that sum up to a target.
  - Array: [2, 7, 11, 15]
  - Target Sum: 9
- Use two index variables **left** and **right** to traverse from both corners.
  - Initialize: left = 0, right = n – 1
  - Run a loop while **left < right,** do the following inside the loop
- Compute current sum, **sum** = arr[left] + arr[right]
- If the **sum** equals the **target**, we've found the pair.
- If the **sum** is less than the **target**, move the **left** pointer to the right to increase the **sum**.
- If the **sum** is greater than the **target**, move the **right** pointer to the left to decrease the **sum**.

Two sum

sorted

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]

Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]

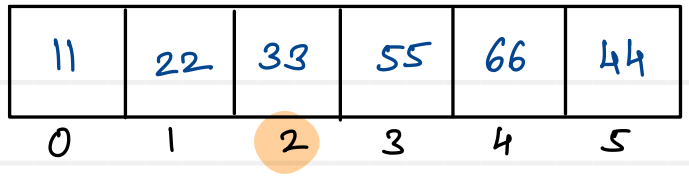
Example 3:

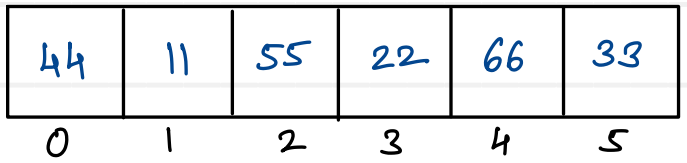
Input: nums = [3,3], target = 6
Output: [0,1]

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length-1;
    while(left < right) {
        sum = nums[left] + nums[right];
        if(sum == target)
            return new int[] {left, right};
        else if(sum < target)
            left++;
        else
            right++;
    }
    return new int[] { };
}
```

Selection sort

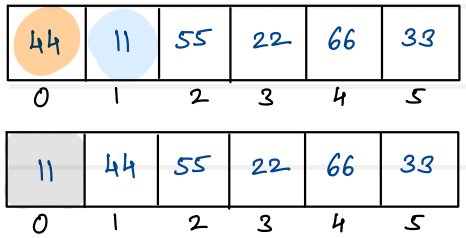
1. Select one position of the array
2. Find smallest element out of remaining elements
3. Swap selected position element and smallest element
4. Repeat above steps until array is sorted (N-1)



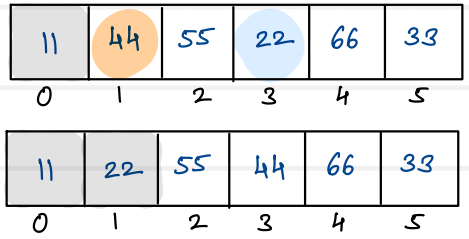


i minIndex j
3 3 4
5
6

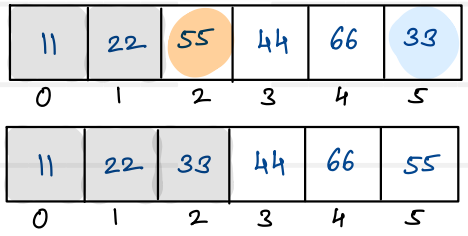
Pass 1



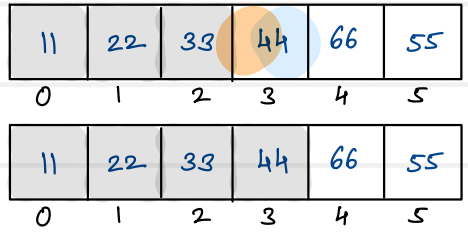
Pass 2



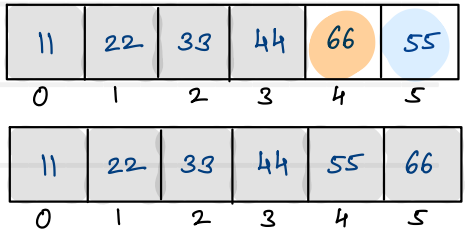
Pass 3



Pass 4



Pass 5

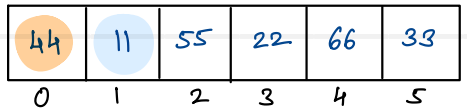


to select positions one by one : $i = 0 \rightarrow N-2$ $(i < N-1)$

to find smallest element : $j = i+1 \rightarrow N-1$ $(j < N)$

```
public static void selectionSort(int arr[], int N){
    //1. select positions one by one
    for( int i=0 ; i<N-1 ; i++){
        //2. find min element from remaining elements
        int minIndex = i;
        for(int j=i+1 ; j<N ; j++){
            if(arr[j] < arr[minIndex])
                minIndex = j;
        }
        //3. swap selected position element & smallest element
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

Selection sort



i	minIndex	j
0	0	1
	1	2
		3
		4
		5
		6

```
minIndex = i
for(j=i+1; j<N; j++)
    if(arr[j] < arr[minIndex])
        minIndex = j;
```

n – no. of elements
n-1 : no. of passes

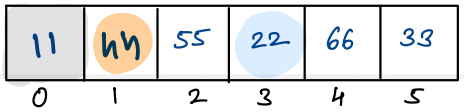
pass	comps
1	n-1
2	n-2
⋮	⋮
n-1	1

Total comps $\approx 1+2+3+\dots+(n-2)+(n-1)$

$\approx \frac{(n-1)(n-1+1)}{2}$

$\approx \frac{n^2-n}{2}$

Time $\propto \frac{n^2-n}{2}$ $\quad \frac{n^2+n}{2}$



i	minIndex	j
1	1	2
	3	3
		4
		5
		6

Mathematical polynomial:

Degree – highest power of variable

- While writing complexities, we always consider only degree term. because it is highest growing term in polynomial

n	n^2
1	1
10	100
100	10000
1000	1000000

$T(n) = O(n^2)$ Best Avg worst

$S(n) = O(1)$

Bubble sort

1. Compare all pairs of consecutive elements of the array one by one
2. If left element is greater than right element , then swap both
3. Repeat above steps until array is sorted (N-1)

11 22 33 44 55 66

11 22 33 44 55 66

11 22 33 44 55 66

11 22 33 44 55 66

11 22 33 44 55 66

No. of comps ≈ n-1

Time ∝ n-1

$T(n) = O(n)$ Best

$S(n) = O(1)$

No. of elements = n

No. of passes = n-1

Pass	Comps
1	n-1
2	n-2
⋮	⋮
n-2	2
n-1	1

Total comps = $1+2+3+\dots+(n-2)+(n-1)$

$= \frac{n(n+1)}{2}$

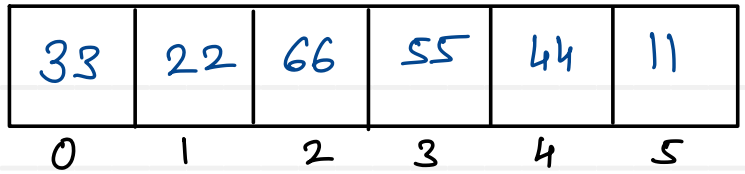
$= \frac{1}{2}(n^2+n)$

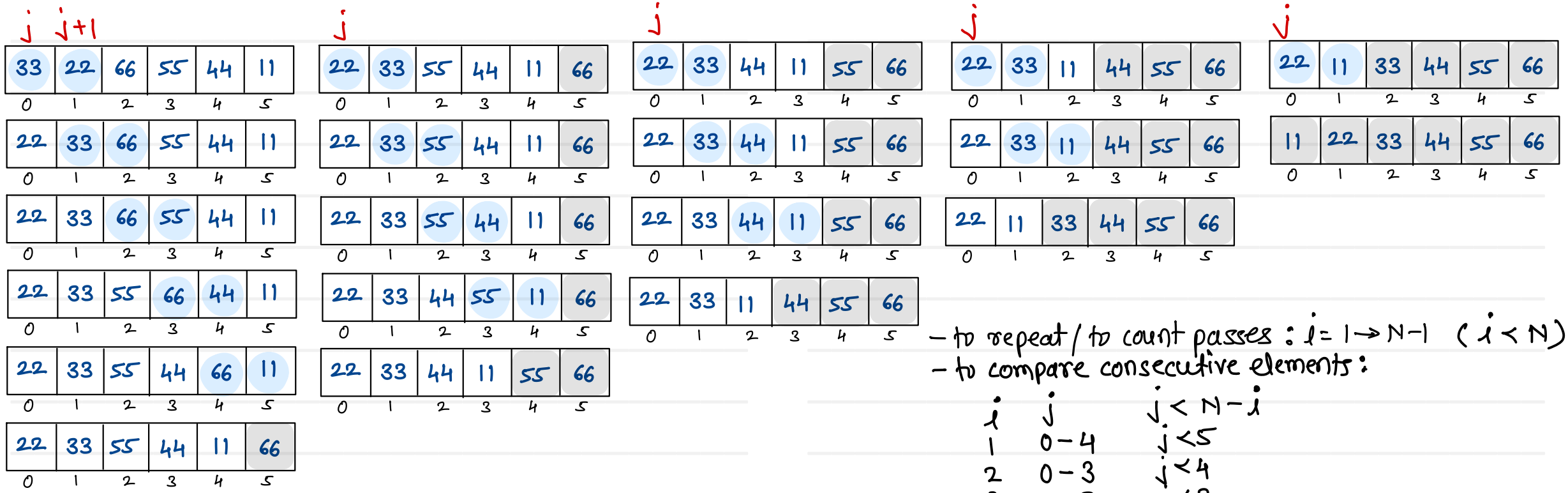
Time ∝ comps

Time ∝ $\frac{1}{2}(n^2+n)$

$T(n) = O(n^2)$ Avg, worst

Bubble sort





- to repeat / to count passes : $i = 1 \rightarrow N-1$ $(i < N)$
- to compare consecutive elements:

i	j	$j < N-i$
1	0 - 4	$j < 5$
2	0 - 3	$j < 4$
3	0 - 2	$j < 3$
4	0 - 1	$j < 2$
5	0 - 0	$j < 1$

Insertion sort

1. Pick one element of the array (start from 2nd element)
2. Compare picked element with all its left neighbours one by one
3. If left neighbour is greater, move it one position ahead
4. Insert picked element at its appropriate position
5. Repeat above steps until array is sorted

pass1: 11 22 33 44 55 66
pass2: 11 22 33 44 55 66
pass3: 11 22 33 44 55 66
pass4: 11 22 33 44 55 66
pass5: 11 22 33 44 55 66

No. of comps = n-1

Time $\propto$ n-1

$T(n) = O(n)$ Best

$S(n) = O(1)$

No. of elements = n
No. of passes = n-1

passes	comps
1	1
2	2
3	3
:	:
n-1	n-1

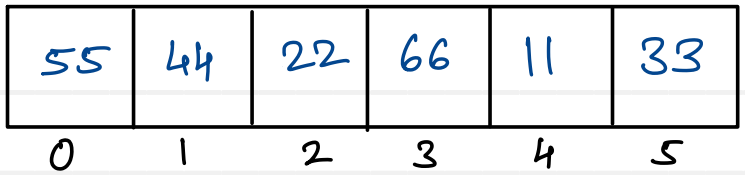
Total comps = $1+2+3+\dots+n$

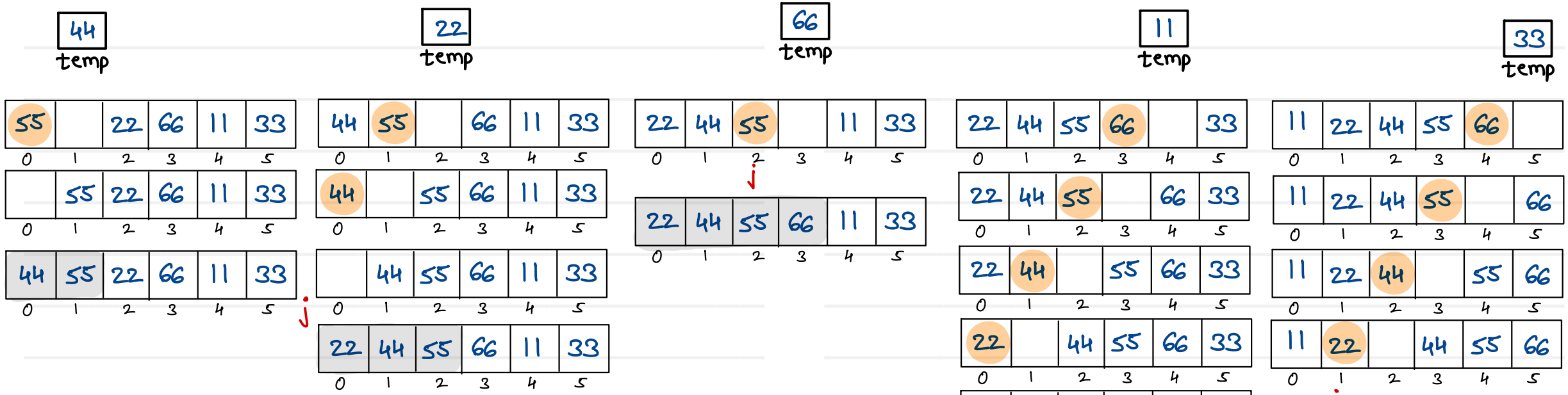
$= \frac{n(n+1)}{2}$

Time $\propto \frac{1}{2}(n^2+n)$

$T(n) = O(n^2)$ worst Avg

Insertion sort



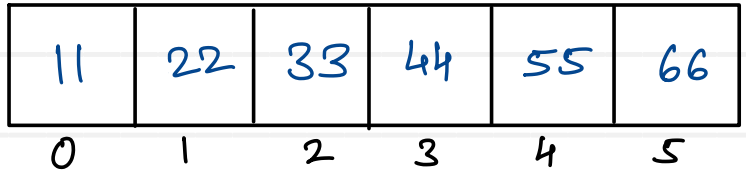


to pick elements : $i = 1 \rightarrow N-1$ $(i < N)$ $i++$

to compare with left neighbors : $j = i-1 \rightarrow 0$ $(j >= 0)$ $j--$

Insertion sort

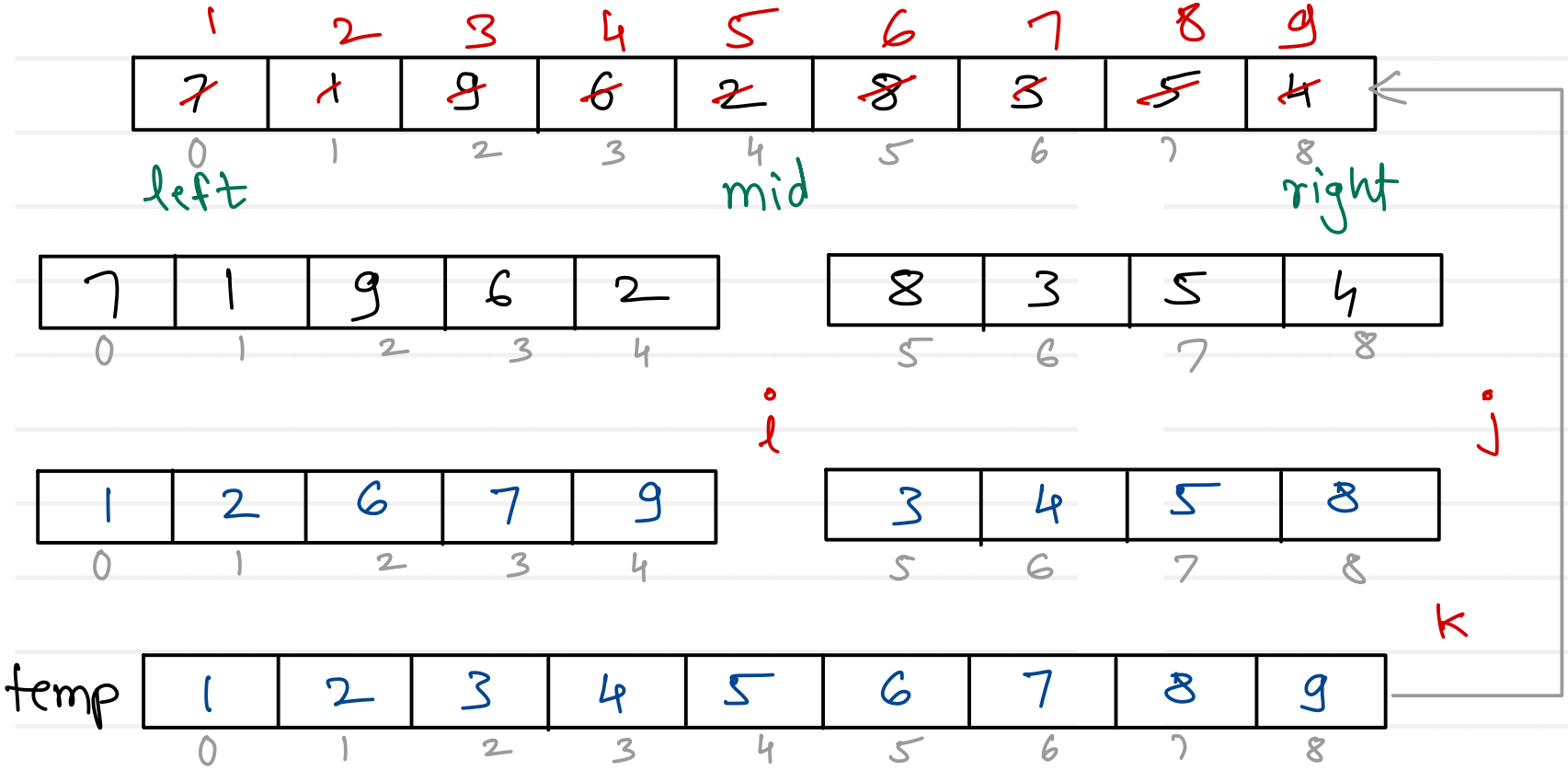
```
for( i=1 ; i<N ; i++) {
    temp = arr[i]
    j;
    for(j=i-1 ; j>=0; j--){
        if(arr[j] > temp)
            arr[j+1] = arr[j];
        else
            break;
    }
    arr[j+1] = temp;
}
```



i	i<6	temp	j
1	T	44	0,-1
2	T	22	1,0,-1
3	T	66	2
4	T	11	3,2,1,0,-1
5	T	33	4,3,2,1
6	F		

Merge sort

1. Divide array in two parts
2. Sort both partitions individually (by merge sort only)
3. Merge sorted partitions into temporary array
4. Overwrite temporary array into original array



mid = (left + right) / 2

left partition : left → mid

right partition : mid+1 → right

size = right - left + 1

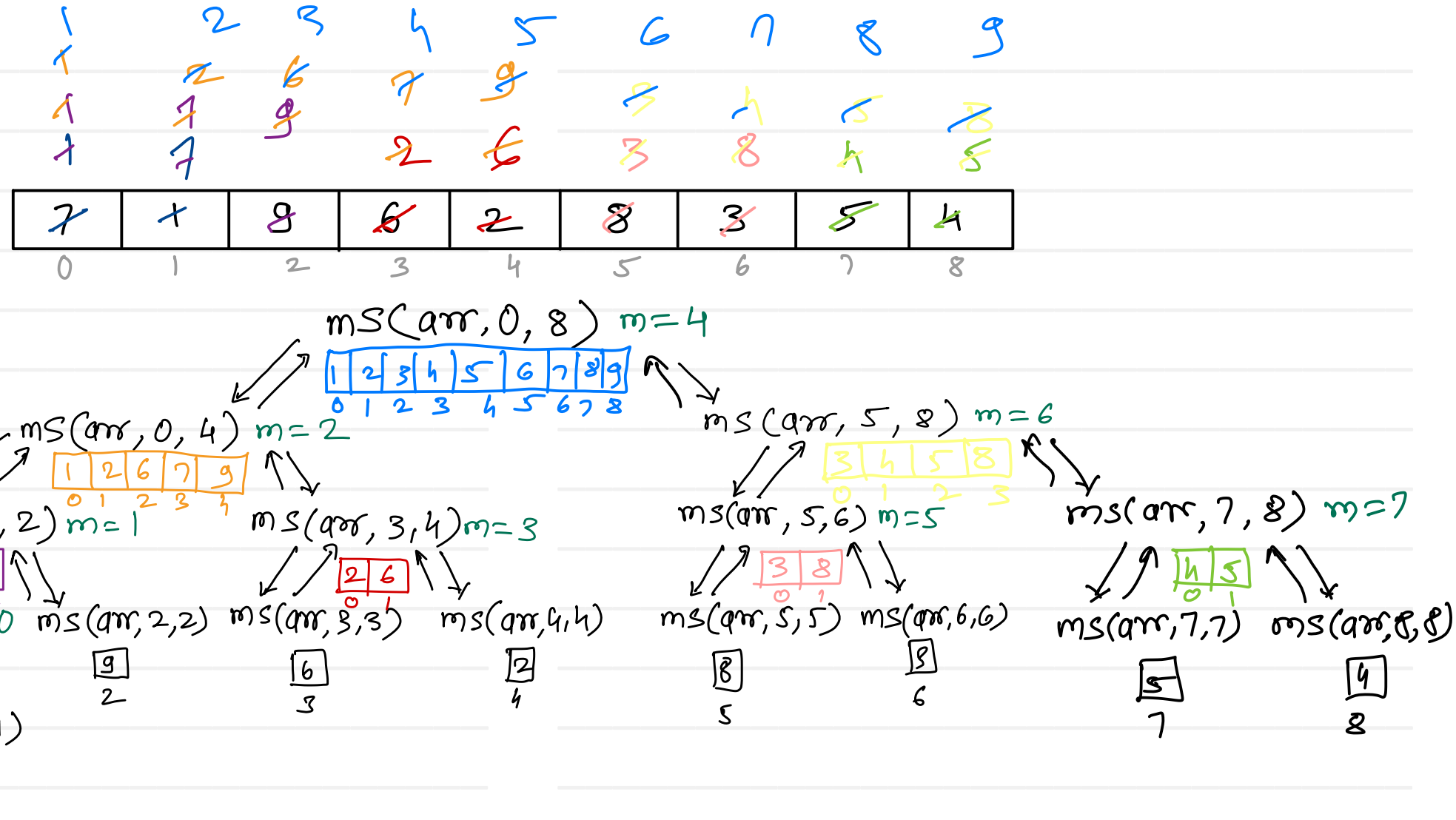
temp array

i = left → mid (i<=mid)

j = mid+1 → right (j<=right)

k = 0 → size-1

Merge sort


ms(arr, 0, 8) m=4
ms(arr, 0, 4) m=2
ms(arr, 5, 8) m=6
ms(arr, 0, 2) m=1
ms(arr, 3, 4) m=3
ms(arr, 5, 6) m=5
ms(arr, 7, 8) m=7
ms(arr, 0, 1) m=0
ms(arr, 2, 2)
ms(arr, 3, 3)
ms(arr, 4, 4)
ms(arr, 5, 5)
ms(arr, 6, 6)
ms(arr, 7, 7)
ms(arr, 8, 8)
ms(arr, 0, 0)
ms(arr, 1, 1)

no. of levels = log n

comps per level = n

Total comps = n log n

Best
Avg
worst

$$T(n) = O(n \log n)$$

temp array is auxiliary space

$$S(n) = O(n)$$

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com