Merge Sort

//1. divide array into two parts
//2. sort both partitions individually
//3. merge sorted partitions into temp array in such way that, temp arr is sorted
//4. overwrite temp array into original array

no. of elements = n
no. of divisons / levels = $\log n$
no. of comparisions per level = n
Total comps = $n * \log n$
Time $\propto n \log n$

$$\boxed{T(n) = O(n \log n)}$$

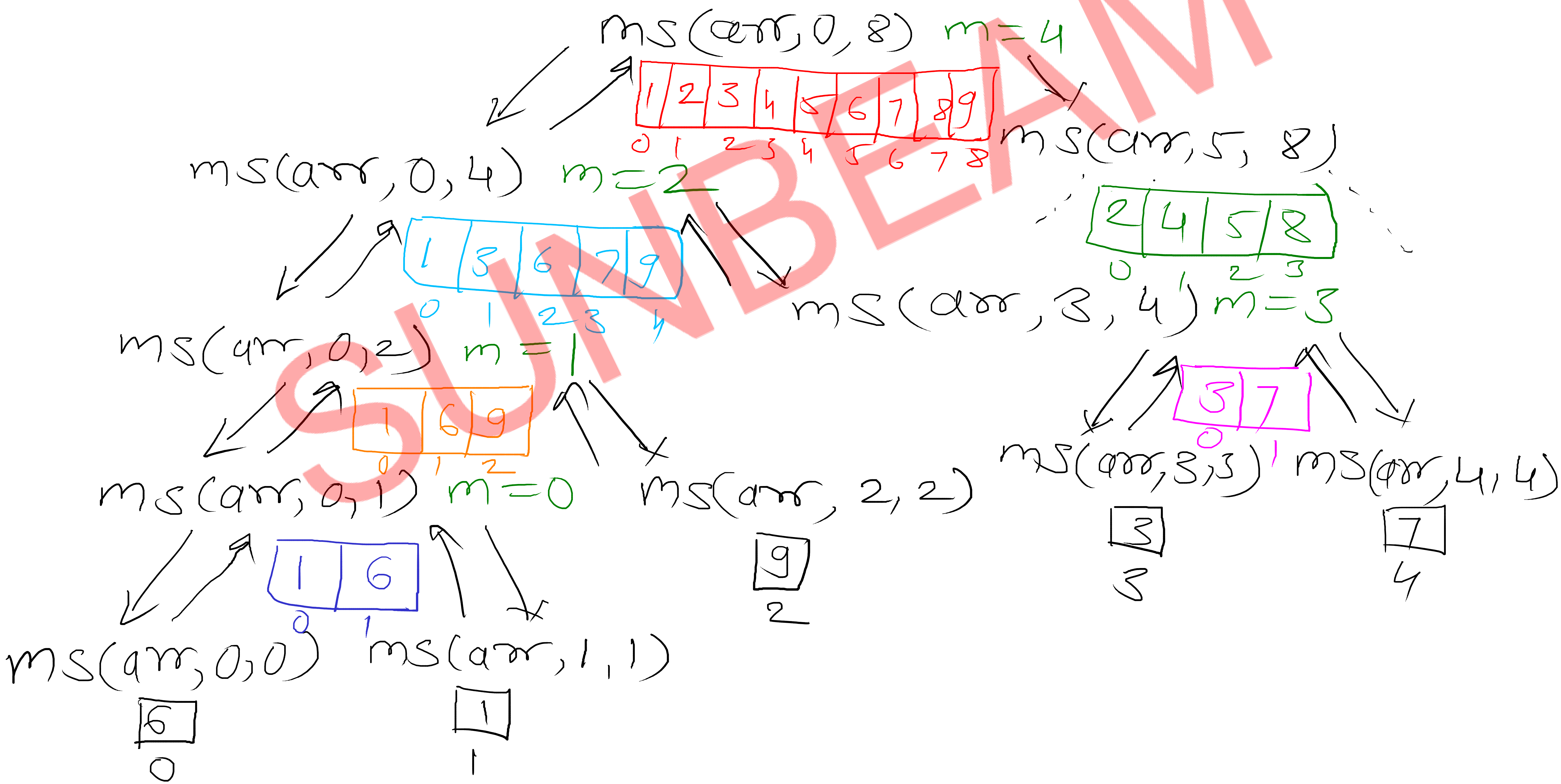
Best
Avg
Worst

processing variable – temp[]
Auxiliary space – n
$AS(n) = O(n)$

Merge Sort

1 2 3 4 5 6 7 8 9

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



Quick Sort

//1. select one referance/axis/pivot element from an array (left, right, mid)
//2. arrage all smaller elements than pivot on left side of pivot
//3. arrange all greater elements than pivot on right side of pivot
//4. sort left and right partitions of pivot individually by applying same quick sort algorithm

no. of elements = n
no. of levels = $\log n$
comps. per levels = n
Total comps = $n \log n$

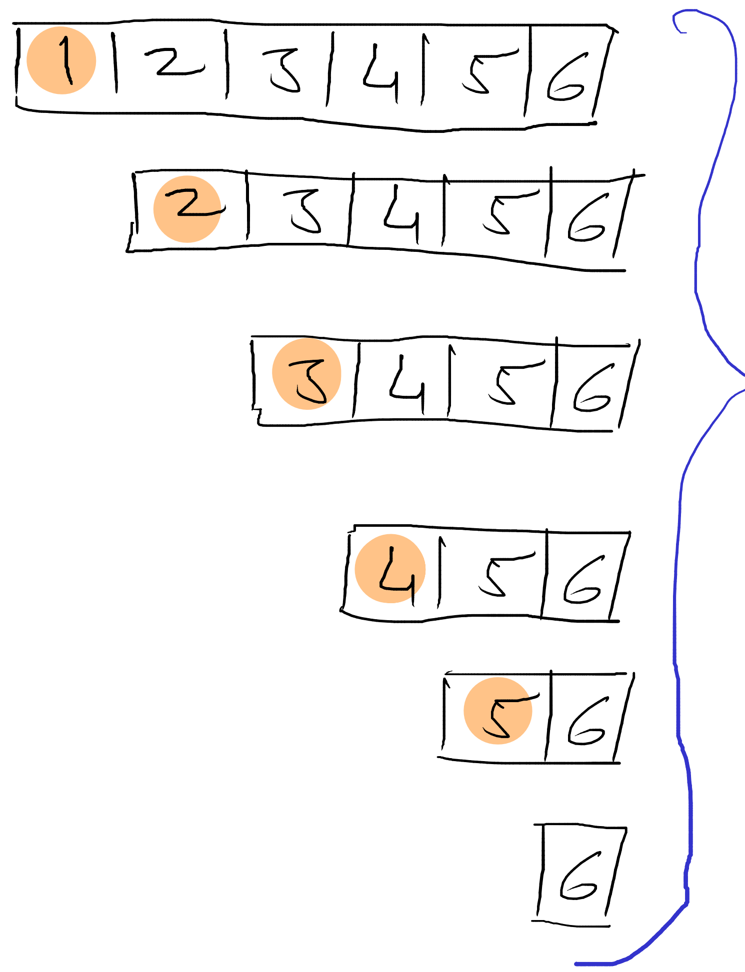
$\boxed{T(n) = O(n \log n)}$ — Best, — Avg

$\boxed{T(n) = O(n^2)}$ — worst

$\boxed{AS(n) = O(1)}$

- Time complexity is dependent on selection pivot.
- to improve time complexity of quick sort, pivot element is selected by any one of the method
  i) median three
  ii) dual pivot

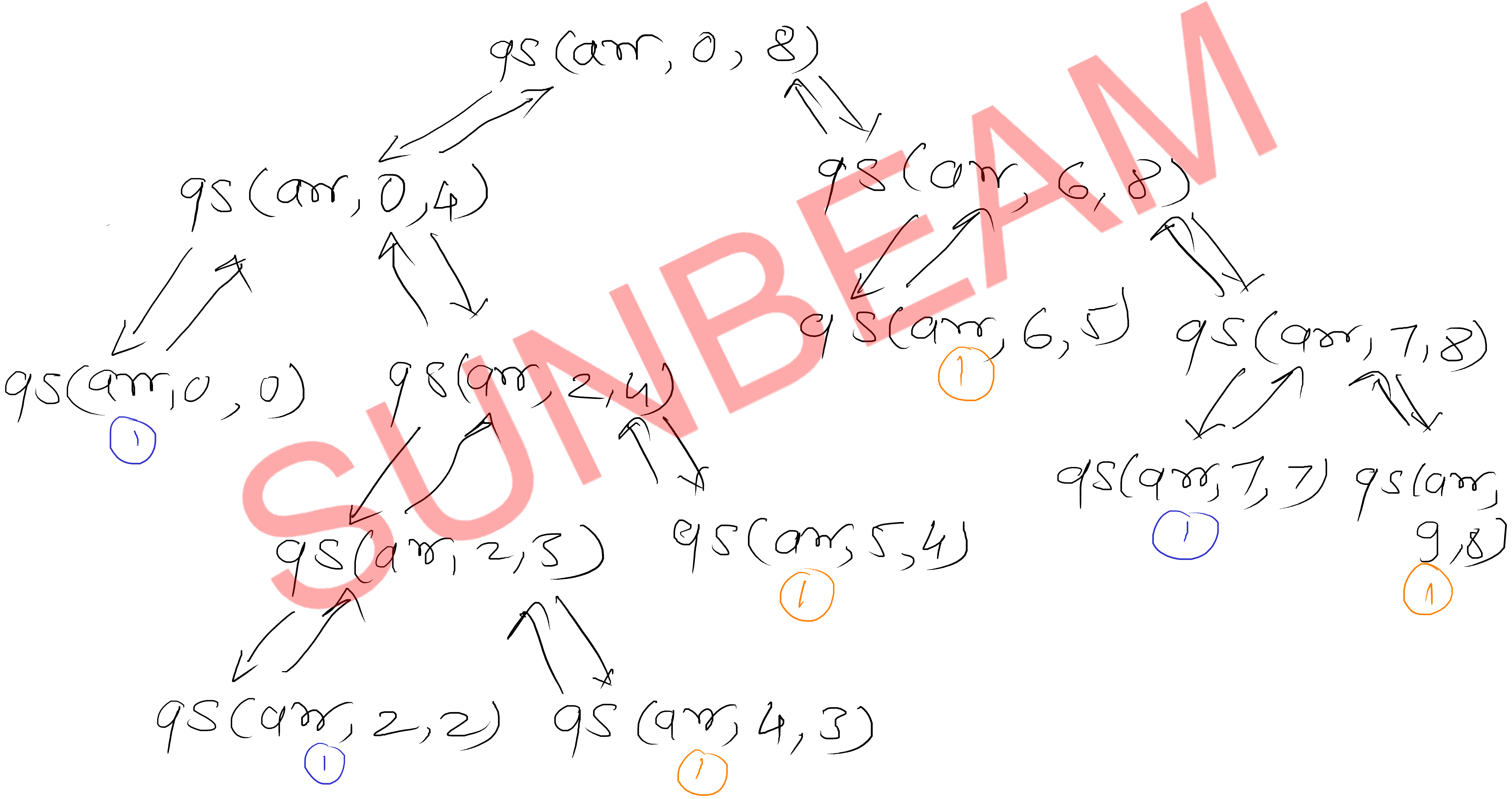
Worst case



no. of levels = n
comps = n^2
Time $\propto n^2$
$T(n) = O(n^2)$

Quick Sort

66	33	99	11	77	22	55	66	88
0	1	2	3	4	5	6	7	8



Stable sort vs Unstable sort

* Array: [{A, 65}, {B, 90}, {C, 55}, {D, 85}, {E, 55}, {F, 65}]

* Stable sort:
  - Equal elements maintains their relative order as in original array -- Guaranteed.
  [{C, 55}, {E, 55}, {A, 65}, {F, 65}, {D, 85}, {B, 90}]
  e.g. Bubble, Insertion, ...

* UnStable sort:
  - Equal elements may not maintain their relative order as in original array.
  [{C, 55}, {E, 55}, {F, 65}, {A, 65}, {D, 85}, {B, 90}]
  e.g. Selection.

In-place sort vs Out-place sort

*** In-place sort**

- No additional space requires for holding array element.
- Aux Space complexity is O(1)
e.g. Selection, Bubble, Insertion, ...

*** Out-place sort**

- Additional space requires for holding sorted array element.
- Aux Space complexity is O(n) -- without stack space.
e.g. Merge

Searching of data

1. **Array - Linear search**
   $T(n) = O(n)$
2. **Array - Binary search**
   $T(n) = O(\log n)$
3. **Linked List - search**
   $T(n) = O(n)$
4. **Binary Tree - search**
   $T(n) = O(n)$
5. **BST - search**
   $T(n) = O(\log n)$

- in all data structures searching time complexity is variable, it depends on size/no of elements in data structure (n)
- in any of the data structure we are not able to search data in constant average time

- this can be done with the help of hashing technique
- implementation of this technique is known as "Hash Table".

Hashing

key value

8, v1

3, v2

10, v3

4, v4 collision

6, v5

13, v6

size = 10

0	10, v3
1	
2	
3	3, v2
4	4, v4
5	
6	6, v5
7	
8	8, v1
9	

Hash Table

$h(k) = k \% SIZE$

$h(8) = 8 \% 10 = 8$

$h(3) = 3 \% 10 = 3$

$h(10) = 10 \% 10 = 0$

$h(4) = 4 \% 10 = 4$

$h(6) = 6 \% 10 = 6$

$h(13) = 13 \% 10 = 3$

Collision :

- when two or more keys yield same slot.

- to handle collision we can use one of the below method.

1) closed addressing

2) open addressing

Add : - O(1)

1) find slot

2) arr[slot] = data

Search : - O(1)

1) find slot

2) return arr[slot]

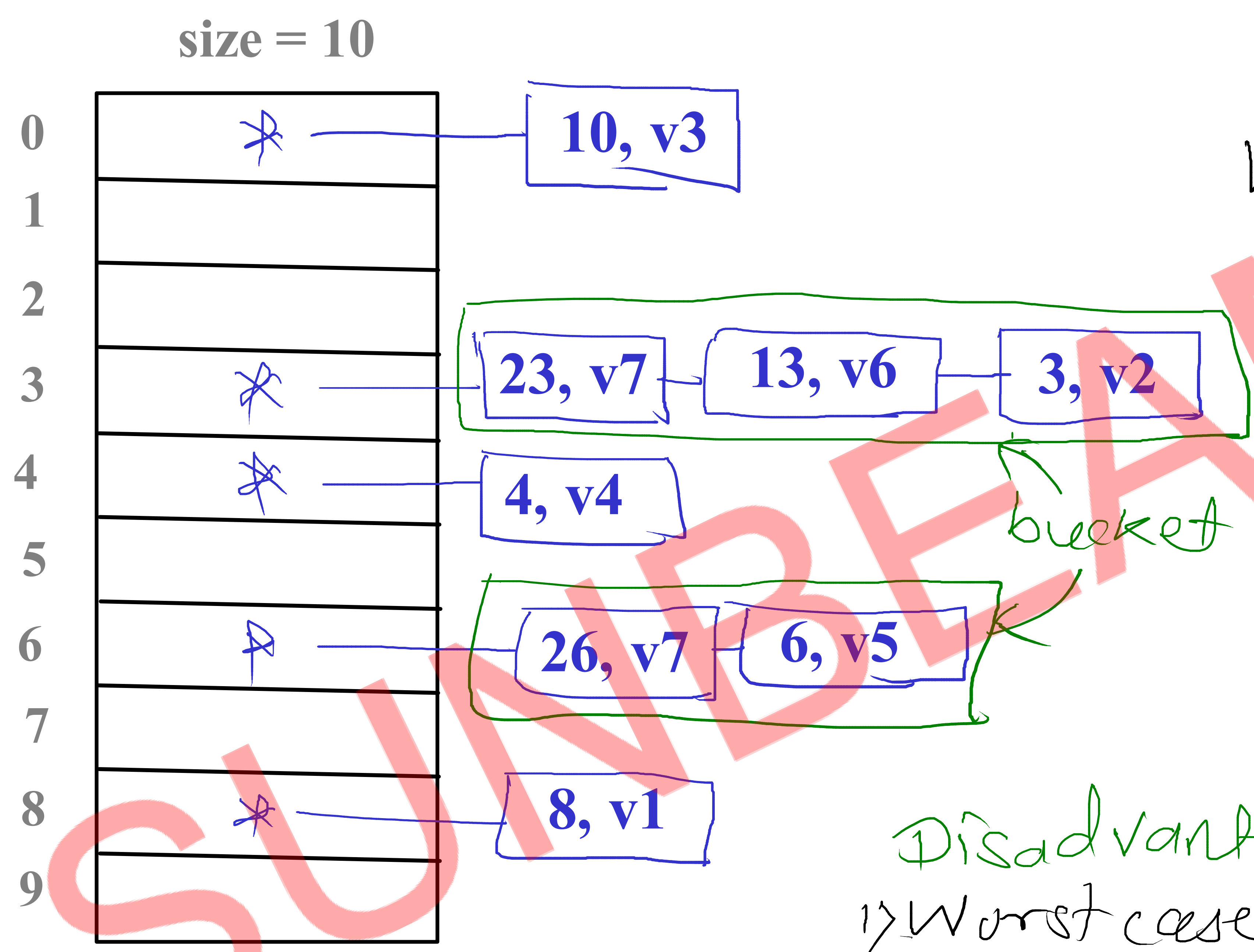
Delete : - O(1)

1) find slot

2) arr[slot] = null

Closed Addressing/ Seperate Chaining / Chaining

8, v1
3, v2
10, v3
4, v4
6, v5
13, v6
23, v7
26, v7



Hash Table

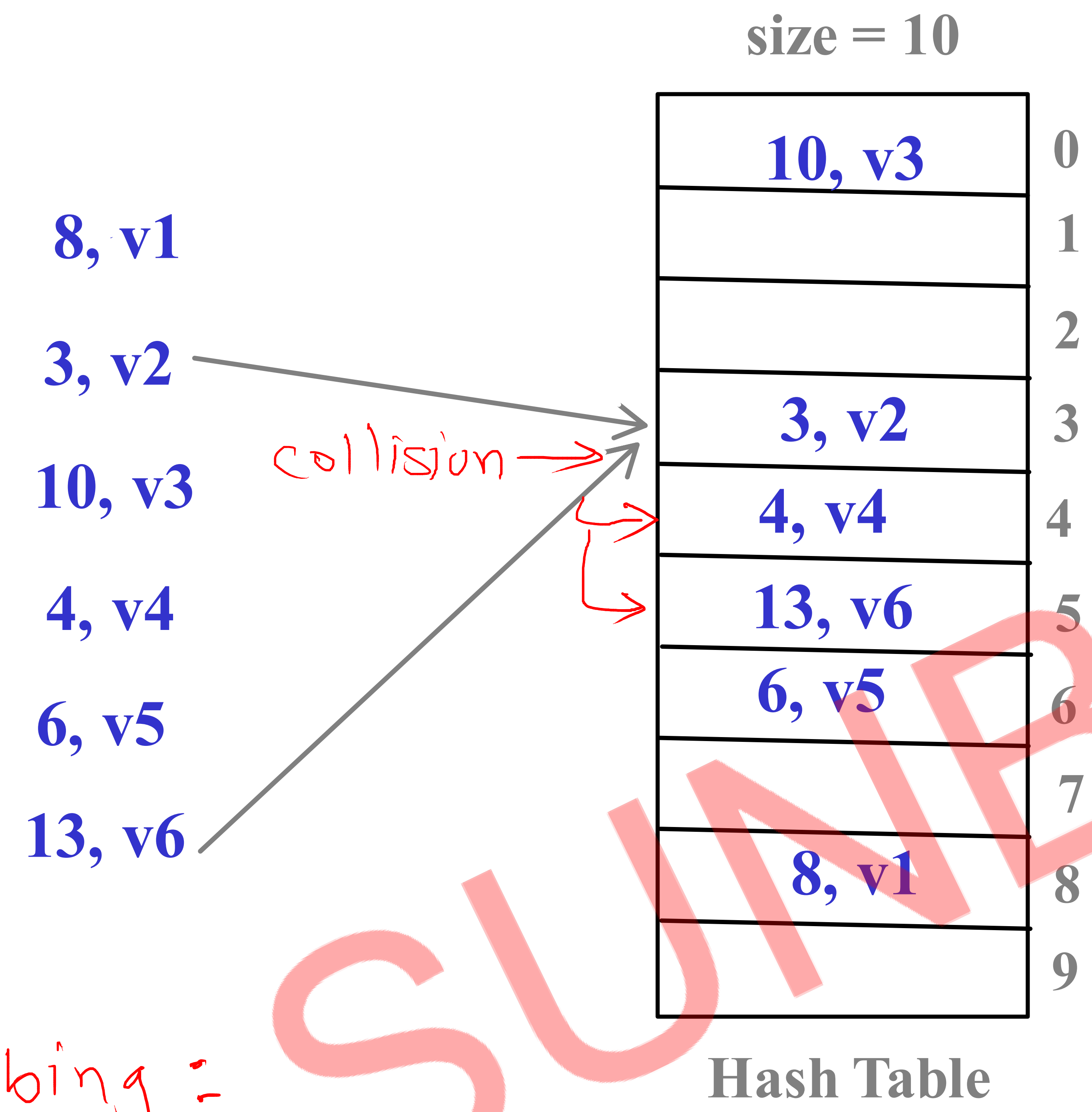
h(k) = k % size

h(8) = 8 % 10 = 8
h(3) = 3 % 10 = 3
h(10) = 10 % 10 = 0
h(4) = 4 % 10 = 4
h(6) = 6 % 10 = 6
h(13) = 13 % 10 = 3 c
h(23) = 23 % 10 = 3 c
h(26) = 26 % 10 = 6 c

Disadvantages:

1) Worst case time complexity is O(n), if multiple keys yield same slot.
2) need more space is required.
3) data (key/value) is stored outside table

Open Addressing - Linear Probing



$h(k) = k \% size$

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

$f(i) = i$

where $i = 1, 2, 3, \ldots$

probe numbers

$h(8) = 8 \% 10 = 8$

$h(3) = 3 \% 10 = 3$

$h(10) = 10 \% 10 = 0$

$h(4) = 4 \% 10 = 4$

$h(6) = 6 \% 10 = 6$

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

$h(13,1) = [3 + 1] \% 10$

$= 4$ (1^{st} probe) c

$h(13,2) = [3+2] \% 10$

$= 5$ (2^{nd} probe)

Probing :

- process of finding empty slot whenever collision will occur