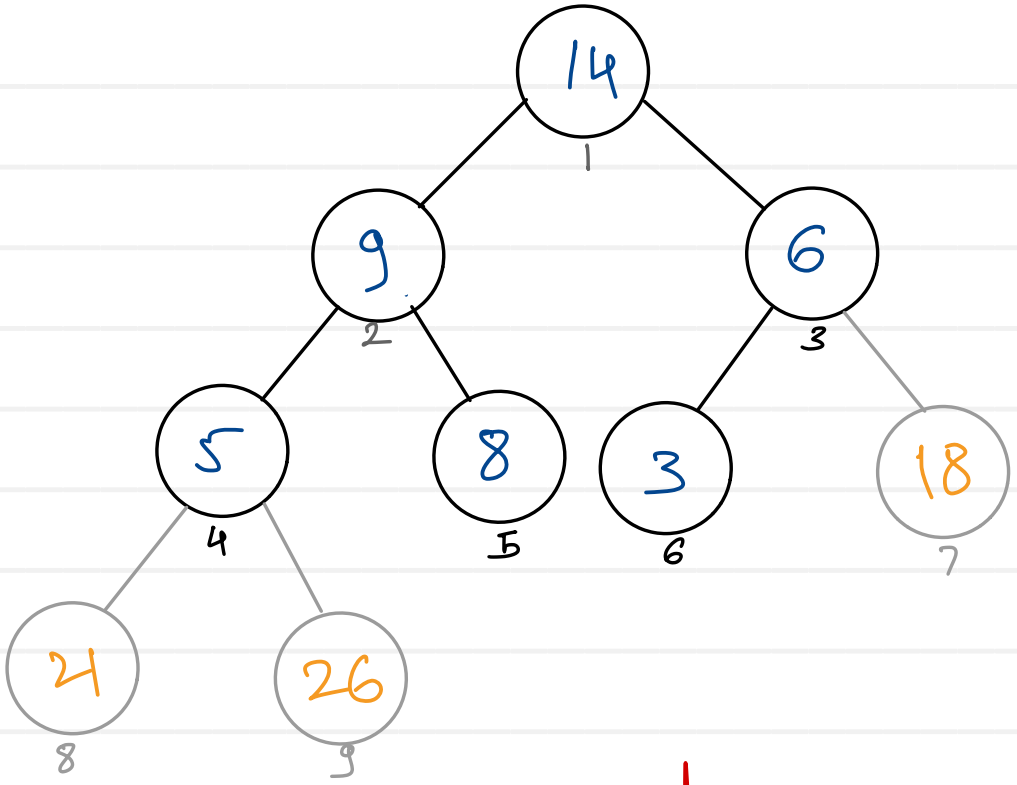
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Heap sort



14	9	6	5	8	3	18	21	26
1	2	3	4	5	6	7	8	9

arr

6	14	3	26	8	18	21	9	5
1	2	3	4	5	6	7	8	9

Algorithm:

1. create heap from given array

ascending sort ← max min → descending sort

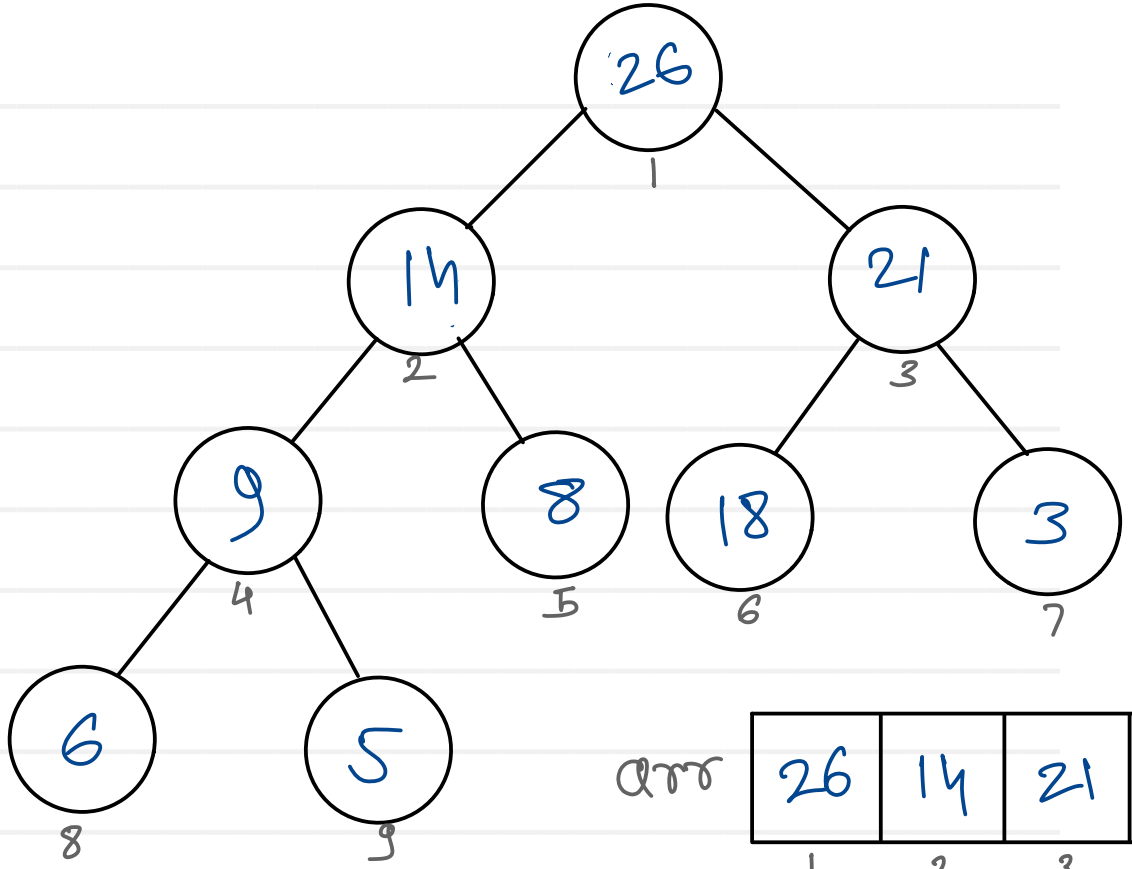
2. delete all elements from heap one by one and place them on last index of array.

create heap = $n * \log n$

delete heap = $n * \log n$

$2n \log n$

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



size = 9

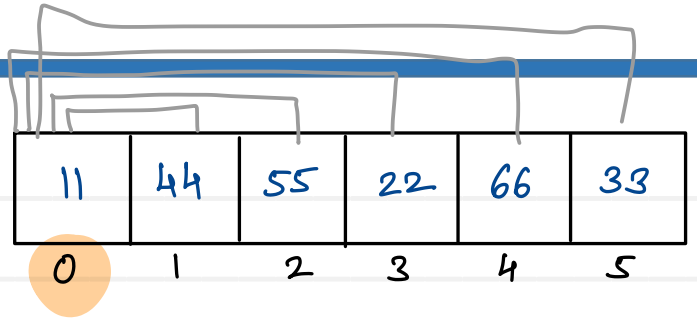
parent = $9/2 = 4$

arr

26	14	21	9	8	18	3	6	5
1	2	3	4	5	6	7	8	9

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)



44	11	55	22	66	33
0	1	2	3	4	5

i: 3 — minIndex: 3 — j: 4 5 6

Pass 1

44	11	55	22	66	33
0	1	2	3	4	5

11	44	55	22	66	33
0	1	2	3	4	5

Pass 2

11	44	55	22	66	33
0	1	2	3	4	5

11	22	55	44	66	33
0	1	2	3	4	5

Pass 3

11	22	55	44	66	33
0	1	2	3	4	5

11	22	33	44	66	55
0	1	2	3	4	5

Pass 4

11	22	33	44	66	55
0	1	2	3	4	5

11	22	33	44	55	66
0	1	2	3	4	5

Pass 5

11	22	33	44	66	55
0	1	2	3	4	5

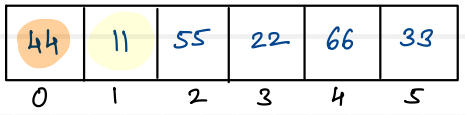
11	22	33	44	55	66
0	1	2	3	4	5

to select position of array : $i = 0 \rightarrow N-2$ $(i < N-1)$

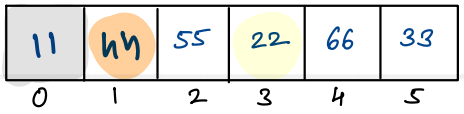
to find smallest element : $j = i+1 \rightarrow N-1$ $(j < N)$

```
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



i	minIndex	j
1	1	2
	3	3
		4
		5
		6

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

Degree of polynomial;
- maximum power in polynomial.
- while writing complexities consider only degree term because it is highest growing term in polynomial

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

N : no. of elements

N-1 : no. of passes

pass	No. of comps
1	N-1
2	N-2
3	N-3
:	:
i	2
N-1	1

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

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

Time $\propto$ comps

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

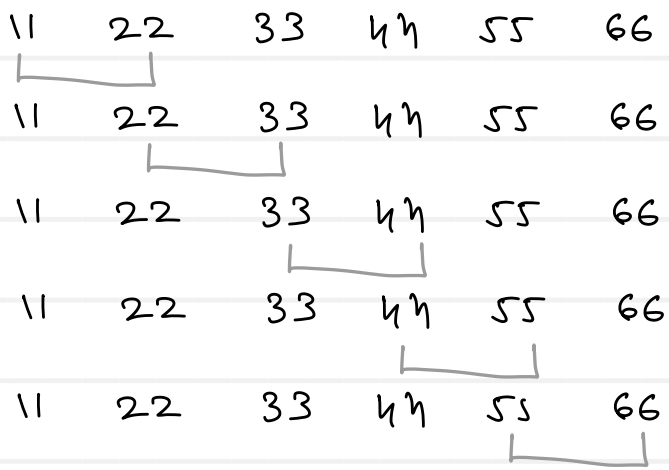
Time $\propto \frac{1}{2}(n^2+n)$

$T(n) = O(n^2)$ — 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)



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

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

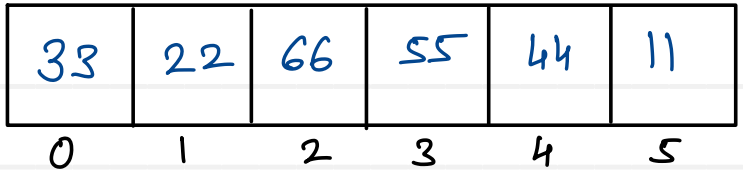
$= \frac{1}{2}(n^2+n)$

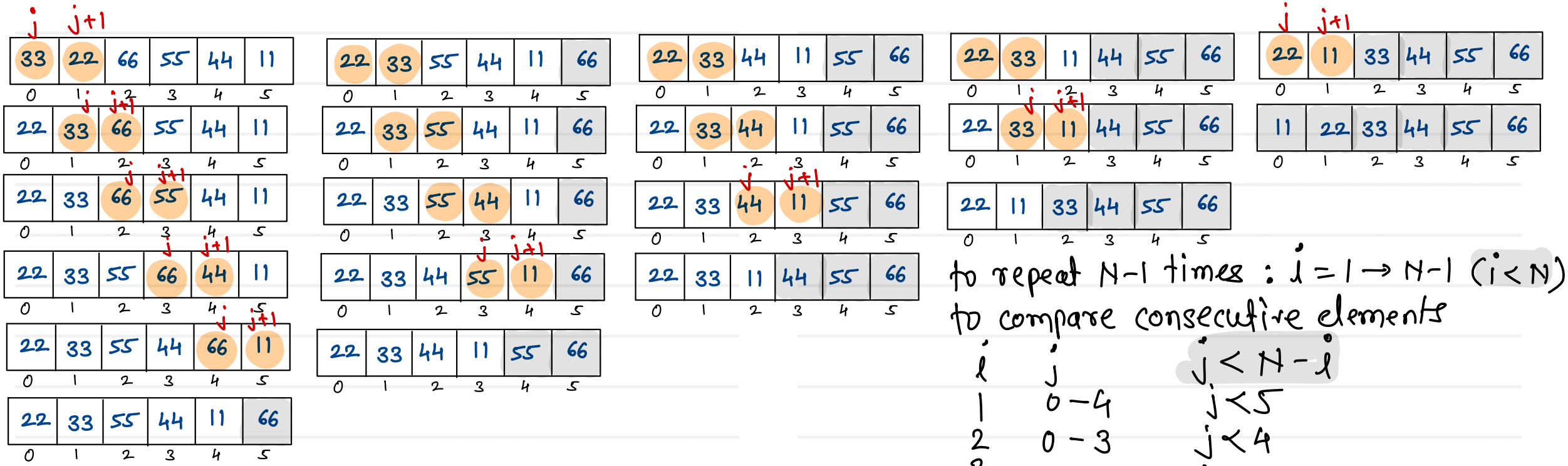
Time ∝ comps

Time ∝ $\frac{1}{2}(n^2+n)$

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

Bubble sort





to repeat N-1 times : $i = 1 \rightarrow N-1$ $(i < N)$

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 index)
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 (N-1)

Total itrs = n-1

Time ∝ 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

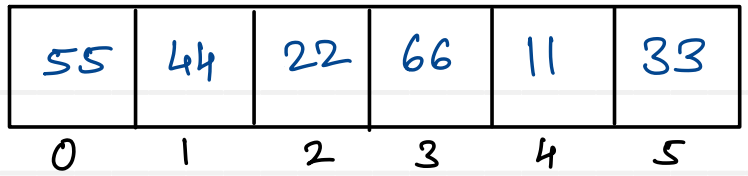
Total comps = $1+2+3+\ldots n$

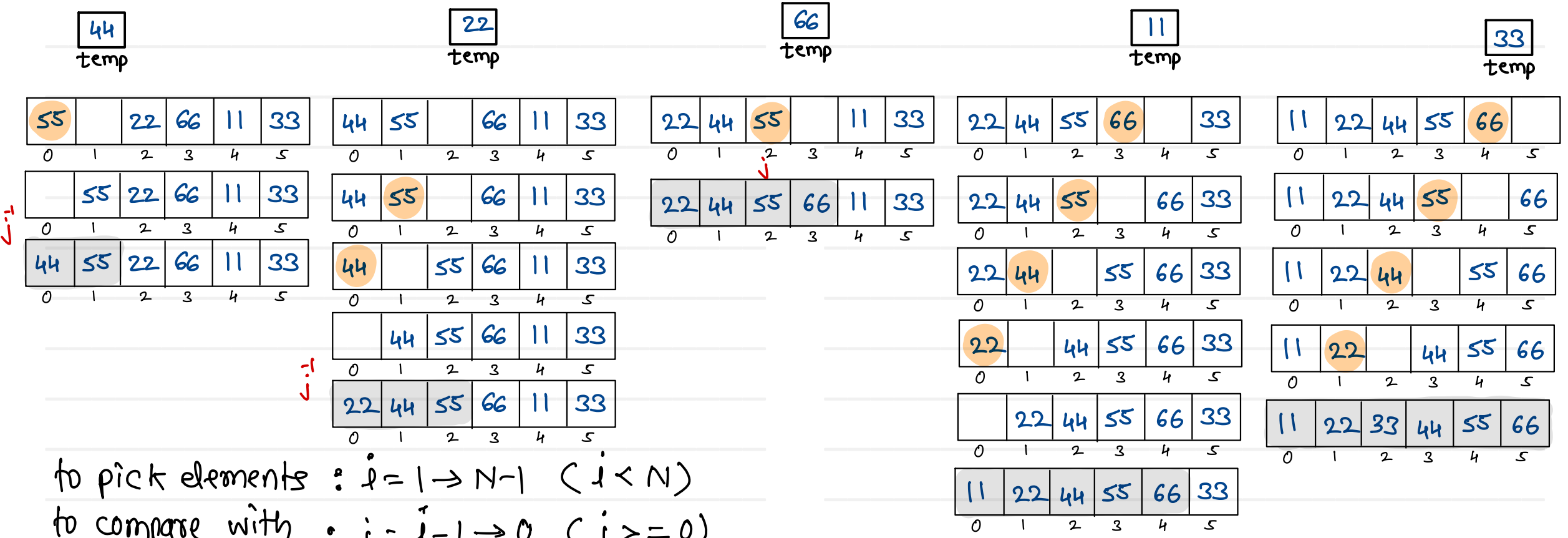
$= \frac{n(n+1)}{2}$

Time ∝ $\frac{1}{2}(n^2+n)$

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

Insertion sort



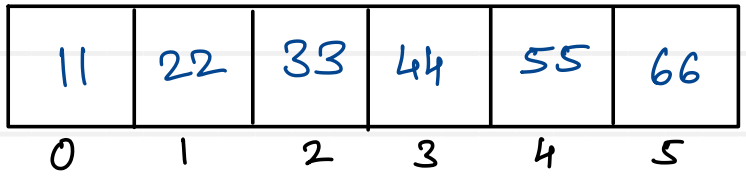


to pick elements : $i = 1 \rightarrow N-1 \quad (i < N)$

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

Insertion sort

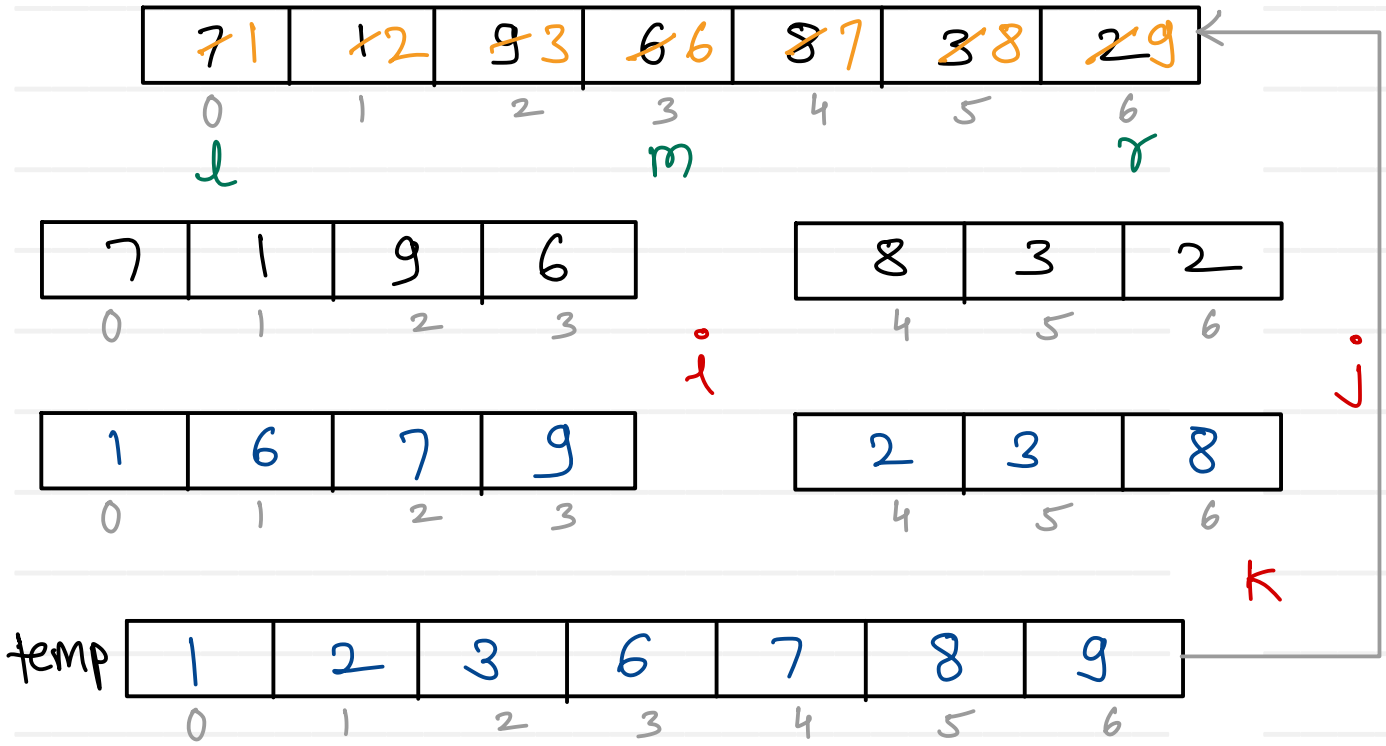
```
for( i=1 ; i<N ; i++) {
    temp = arr[i]
    j;
    for(j=i-1 ; j>=0; j--){
        if(arr[j] > temp)
            arr[j+1] = arr[j];
        else
            break;
    }
    arr[j+1] = temp;
}
```



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



$$mid = \frac{left + right}{2}$$

left partition : left → mid

right partition : mid+1 → right

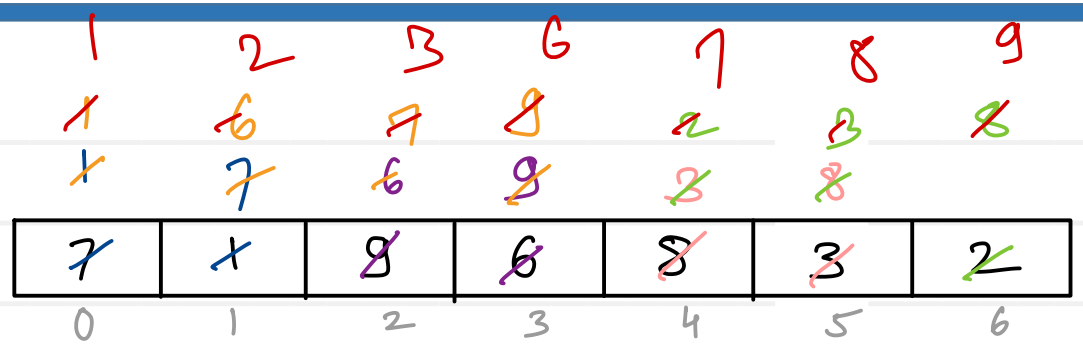
size of temp array (size) = right − left + 1

i = left → mid

j = mid+1 → right

k = 0 → size

Merge sort

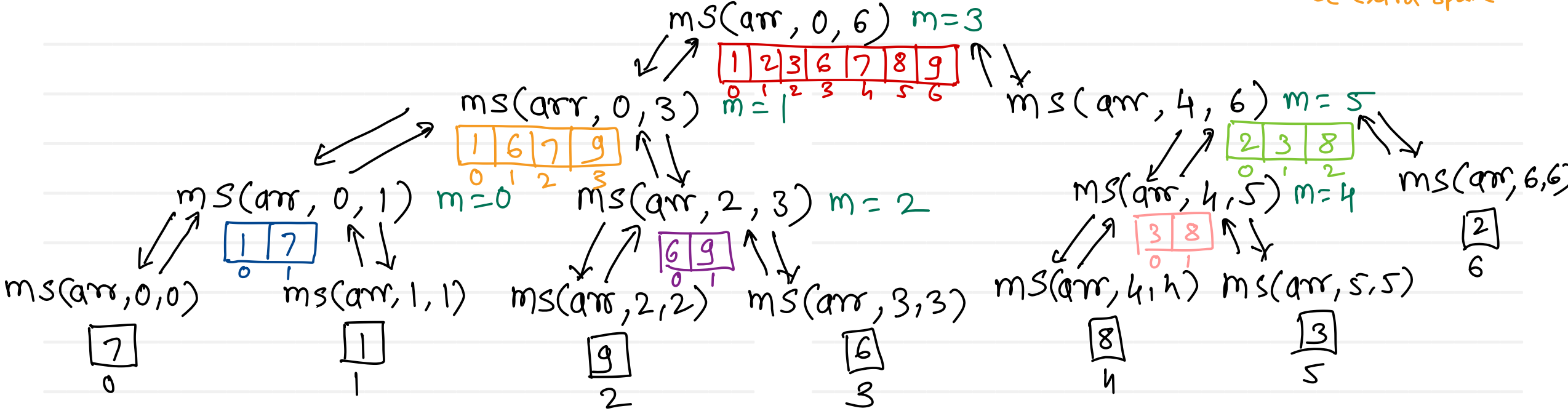


No. of levels = $\log n$
comps per level = n
total comps = $n \log n$

Best Avg Worst $T(n) = O(n \log n)$

$S(n) = O(n)$

↳ space of temp array will be extra space



	space	Time Best	Time Avg	Time Worst
selection sort	O(1)	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
insertion sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	O(n)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

in place sorting algorithm

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]

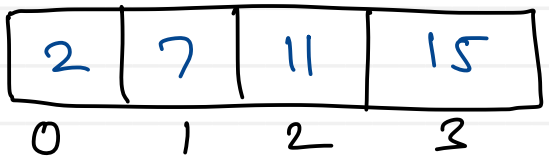
$T(n) = O(n^2)$

$S(n) = O(1)$

```
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 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.



- 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:

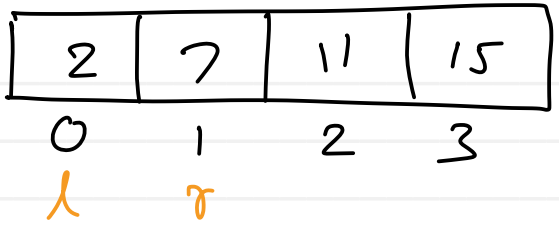
Input: nums = [3,2,4], target = 6

Output: [1,2]

Example 3:

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

Output: [0,1]



target 9

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

```
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[] {};
}
```

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.

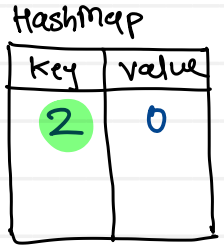
You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

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

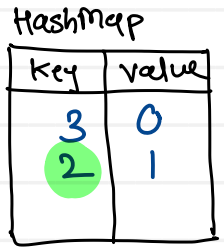
Output: [0,1]



Example 2:

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

Output: [1,2]



Example 3:

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

Output: [0,1]

$T(n) = O(n)$

$S(n) = O(n)$

```
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[]{};
}
```

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com