Sorting:

What is sorting?

Ascending or descending.

Sorting types: Place (In-place and out of place) and stability (Stable and Un-stable)

In place-> No extra space is required-> Bubble sort.

Out of place-> Extra space-> Merge sort.

Stable sort: After sorting the sequence should not change. Duplicate values should be available-> Insertion sort.

Unstable: Sequence gets changed--> Quick sort (Group by concept)

## Terminology in sorting:

Increasing order (ascending),

Decreasing order (Descending).

Non-Increasing/Decreasing order: Duplicates available in the sequence.

**Types of algorithm to be learned:**

Bubble, Selection, Insertion, Bucket, Merge, Quick, Heap

Based on space, time, and stability.

# Bubble sort:

It is also called sinking sort. It will compare adjacent item and swap if in wrong order.

**3 1 5 2 6 4**

Take first two and compare, swap the places. run n times. Final element will be available in the nth place after first sorting.

proceed the same for n-1 times.

**Bubble sort(array/a[arr])**

int n = arr.length

for (i=0,i<n-1;i++) run from first cell to last cell.

for (j=0,j<n-i-1;j++) skip last iteration.

if(arr[j]>arr[j+1]

swap(arr[j]>arr[j+1])

**Time complexity:** o(n^2) space complexity - O(1)

**When to use:** when input is already sorted, space is a concern and easy to implement.

**Not to use:** Time complexity issues.

# Selection Sort:

Finding min or max element in unsorted array and putting it in its correct position in a sorted array.

**3 1 5 2 6 4**

Put the result of min(find among all the element) in the first cell and make it as sorted. Then work on unsorted array for rest of the n-1 element.

selectionsort(A)

loop:j=0 to n-1

int min =j

loop i= j+1 to n-1

if (a[i]<a[min]

min =i-1

if min!=job

swap(a[j],a[min])

**Time complexity** - o(n^2); Space- O(1)

**Advantages**: Its performance is easily influenced by the initial ordering of the items before the sorting process.

**Disadvantage**: Performance

# Insertion sort:

Divide into sorted and unsorted. Similar to selection, but here we pick the first element of unsorted array and put in the sorted array.

**3 1 5 2 6 4**

insertionsort(A)

loop i=1 to n-1

currentnumber=a[i], j=i-1

while a[j-1]>currentnumber && j>0

a[j]=a[j-1]

j++

a[j]= currentnumber

**Time** - o(n^2); **Space complexity** -O(1)

**Advantage**: No extra space required, simple, When there is continuous inflow then its the best.

**Dis-advantage**: Average case is bad.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Bucket sort:

Sorting algorithm that works by disturbing the elements of array into number of bucket.

Each bucket is sorted individually. Then merge the buckets to get it sorted.

**Creating buckets:**

Create number of buckets: Ceil / floor (Square root of total number of items);

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 3 | 2 | 4 | 9 | 8 | 7 |

5\*3/9

A

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |

B

|  |  |  |
| --- | --- | --- |
| 4 | 5 | 6 |

C

|  |  |  |
| --- | --- | --- |
| 7 | 8 | 9 |

Iterate through each number and place it in app bucket.

**Bucket= Ceil (value\*number of bucket)/max value in array**

**Algorithm:**

Bucketsort(A)

find no of buckets to be created. -> 1

find divisor rules -> 1

loop i=0 to n-1 -> n

insert A[i] into array B[] using divisor and bucket -> 1

Sort B[] with insertion sort -> n log n

concatenate B1,B2,B3 -> n

**Time complexity- O(n log n)**

**Space – O (n)**

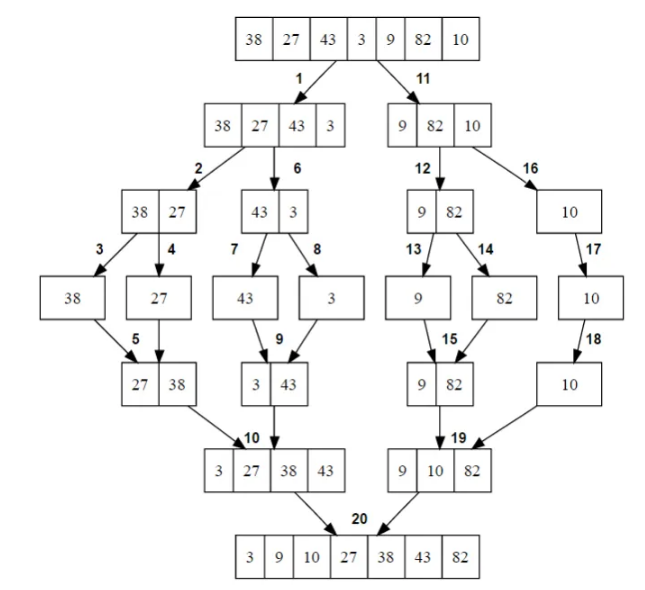
**Advantage**: Better Time complexity, independently sorted.

**Disadvantage**- Space is a concern. Only when the data is uniformly distributed

# Merge sort:

Divide and conquer. Divide array into 2 halves recursively until it become too small to be broken further.

Then each of the broken pieces are merged together to inch toward final array.



mergesort(A,i,r)

if r>1

middle m= i+r/2

mergesort(a,i,m)

mergesort(a,m+1,r)

merge(a,i,m,r)

Merge A,p,m,r)

createtmp arrays L&R and copy A, p,m into L&A, m+1, r into Re-structuring

i+j=0

loop k=p to r/2

if L[i]<R[jj]

a[k]=L[i], i++

else

a[k]=R[j], j++

Time = O(n logn)

Space - O(n)

Adv- stable sort, time ; Dis: Extra space.

# Quicksort:

Divide and conquer.

Each step it finds a pivot and it makes sure that smaller element is towards its left and bigger on the right.

Perform pivot recursively.

**Concept**:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 3 | 2 | 4 | 9 | 8 | 7 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 2 | 1 | 3 | **5** | 6 | 9 | 8 | 7 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | **2** | 4 | 3 | **5** | 6 | 9 | 8 | 7 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | 6 | 9 | 8 | 7 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | 6 | **7** | 8 | 9 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |

**How to find Pivot:**

**3 Parameters** -> I, J, Pivot. End value of I is the Pivot

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 1 | 3 | 2 | 4 | 9 | 8 | 5 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |

# Quicksort Algorithm:

3 variables- i=j-1; j=0, p=last at the beginning; compare j and p - if j is less than p, swap i and j. Increment i. Do it recursively and at the end i value will be the pivot value.

**Quicksort(i, j, p)**

if(j<p)

r= partition (i, j, p)

Quicksort (i, j, r-1)

Quicksort (i, r+1, j)

partition (i, j, p)

pivot = p

i = j-1

for (y=j to p)

if (a[y]<=a[p])

increment i and then swap a[j ,p]

**Time Complexity:** O (n log n)

**Space Complexity:** O (n)

**Advantage**: Best case for time complexity.

**Dis-advantage:** Space concern, when stable sort is needed.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Heap sort:

Inserts the data to be sorted in binary tree called heap.

It then removes the topmost item and inserts it in the current array. It keeps doing until the heap is empty.

Only works in array and not in linked list.

Binary heap will follow the character of Heap (Binary tree). The parent value should be less than its children (Min-heap). Greater than (Max heap)

Every branch should be validated again and again. And tree is build. To make the sorted array, only root node is taken out.

After taking root and placing, the left bottom will replace the root and the binary tree sort should be continued until all the tree becomes null.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 1 | 3 | 2 | 4 | 9 | 8 | 7 |

**Algorithm**:

Heapsort (A)

for i=0 to A.length-1

insertHeap(A{i})

for i=0 to A.length-1

extractfromheap(a[i])

Time complexity: O (n log n); Space - O(1)

**Advantage**: No space and best time.

**Dis-advantage:** Not a stable sort.

Sorting Compared:

