Sorting Algorithms:

There are different types of sorting algorithms because different algorithms have different strength and weakness making them better suited for different situations
We use different algorithms based on

1. Time complexity:Different sorting algorithm have different time complexities which determine how fast they can sort a given set of data.some algorithms have better time complexity for some type of data and while some others are efficient for large data sets
2. Memory usage:some algorithm require more memory space than others for example merge sort is an recursive algorithm which requires additional memory space to create temporary arrays while quick sort is an in-place algorithm that does not require an additional memory
3. Stability: some sorting algorithms are stable which means they preserve the relative order of equal elements in the sorted list for example : in a stable sort if two elements have the same value the one that appears first in the original list will appear first in the sorted list other sorting algorithm such as quick sort are not stable
4. Ease of implementation:Different sorting algorithms have different levels of complexity when it comes to implementation.some algorithms are relatively easy to implement while others require more advanced programming techniques
5. Application-specific requirements:certain applications may require specific sorting algorithm due to their unique requirements ex:some applications may require a stable sort or a sort that is efficient at sorting nearly sorted data

The different sorting algorithms are

1. Insertion sort
2. Bubble sort
3. Selection sort
4. Quick sort
5. Merge sort

Divide and conquer strategy:
A divide and conquer algorithm is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

To use the divide and conquer algorithm, recursion is used.

How Divide and Conquer Algorithms Work?

Here are the steps involved:

1. Divide: Divide the given problem into subproblems using recursion.
2. Conquer: Solve the smaller subproblems recursively. If the subproblem is small enough, then solve it directly.
3. Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Divide and Conquer Applications

- Quick sort
- Merge sort
- Binary Search
- Strassen's Matrix multiplication
- Karatsuba Algorithm

Insertion sort:
Algorithm:

1. Start by defining the insertion sort function which takes an array arr as input.
2. Loop through each element in the array starting from the second element (index 1).

For each element key in the array:
a. Set key as the current element being inserted.
b. Loop backwards through the sorted subarray to its left and compare the key with each element to its left.
c. If an element to the left of the key is greater than key, move that element one position to the right.
d. Repeat step 3.b and 3.c until the correct position for the key is found in the sorted subarray.

3. e. Insert key into its proper position in the sorted subarray.
4. After all elements have been inserted into their proper positions, return the sorted array.

Time complexity:
Worst case:o(n2)
Best case:o(n)
Average Case:o(n2)
Space complexity:o(1)
Insertion sort is suitable for smaller input size

```
function insertionsort(arr){

    for(i=1;i<arr.length;i++)
    {
        let key=arr[i]
        for(let j=i-1;(i>=0&&arr[j]>key);j--)
        {
            arr[j+1]=arr[j]
            arr[j]=key
        }
    }
    return arr
}
```

```
const arr=[9,-3,7,5,-1]
const res=insertionsort(arr)
console.log(res)
```

Bubble sort:
Algorithm
:step 1:create a function called Bubble Sort that takes an array arr as its input
Step 2: create a variable called swapped and initialize it to true
Step 3 create a do while loop that continues to run while swapped is true
Step 4 with in the loop ser swapped to false
Step 5 create a for loop that iterates over arr from the first index to the secondary to the last index
Step 6: within the for loop check if arr[i] is greater than arr[i+1]
Step 7:if arr[i] is greater than arr[i+1] swap the values of arr[i] and arr[i+1]
Step 8: set swapped to true
Step 9: After the for loop completes check if swapped is still true
Step 10:If swapped is still true repeat steps 4-9
Step 11:if swapped is true exit the do while loop
Step 12:return the unsorted array

```
function bubblesort(arr){
let swapped
do{
    for(let i=0;i<arr.length-1;i++){
        swapped=false
        if(arr[i]>arr[i+1])
            {
                temp=arr[i]
                arr[i]=arr[i+1]
                arr[i+1]=temp
                swapped=true
            }
    }

}while(swapped)
}

const arr=[9,-6,-10,4,2,7]
bubblesort(arr)
console.log(arr)
```

Selection sort

Algorithm:

1. The function selectionsort is defined which takes an array arr as input
2. The outer loops run from i=0 to i<arr.length and selects one element at a time as the minimum value
3. The variable min is initialized to the current index i
4. The inner loop runs from j=i+1 to j<arr.length and compares each element with the current minimum
5. If an element arr[j] is smaller than the current minimum arr[min] each element with the current minimum
6. If an element arr[j] is smaller than the current minimum arr[min] the index of min is updated to j
7. After the inner loop completes if the index of minimum min is different from the current index i then a swap operation is performed between the elements arr[i] and arr[min]
8. The outer loop continues to select the next smallest element until the entire array is sorted
9. Finally the sorted array is returned from the function

```
function selectionsort(arr) {
    // outer loop to select one element at a time
    for(let i = 0; i < arr.length; i++) {
        let min = i; // initialize minimum index

        // inner loop to find the smallest element
        for(let j = i + 1; j < arr.length; j++) {
            if(arr[j] < arr[min]) {
                min = j; // update minimum index
            }
        }

        // swap if minimum is not at its original position
        if(i !== min) {
            let temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }

    // return sorted array
    return arr;
}
```

```
const arr = [0, -1, 5, 8, 22];
const res = selectionsort(arr);
console.log(res); // [-1, 0, 5, 8, 22]
```

Quick sort
Algorithm:

1.Definr the function quicksort which takes an array arr as input

2. Check if the length of the array is less than 2 if yes return the array as it is already sorted

3. Set the pivot as the last element of the array

4,create two empty arrays left and right which will store elements less than or greater than the pivot respectively

5. Loop through array from index 0 to the second last index

6 if the current element is less than the pivot push it into the left array

7 Otherwise push it to the right array

8 Recursively call the quick sort function on the left and right array

9.Concatenate the sorted left array pivot and sorted right array

```
function quicksort(arr)
{
    if (arr.length<2)
    {
        return arr
    }
    let pivot=arr[arr.length-1]
    let left=[]
    let right=[]
    for(let i=0;i<arr.length-1;i++)
    {
        if(arr[i]<pivot)
        {
            left.push(arr[i])
        }
        else{
            right.push(arr[i])
```

```
        }
    }
    return[...quicksort(left),pivot,...quicksort(right)]
}
const arr=[8,20,-2,4,-6]

const res=quicksort(arr)
console.log(res)
```

Merge sort:
Merge Sort is one of the most popular sorting algorithms that is based on the principle of DIvide and conquer strategy.Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Algorithm:

Stack:
Stack is a sequential collection of elements that follows the principle of last in first out
The last element inserted in to the stack is the first element to be removed
Stack is an abstract data type it is defined by its behaviour rather than being a mathematical model
Stack has two operations

1.push
2.pop

Stack usage:

1.Browser History Tracking
2.undo operation when typing
3.Expression conversion
4.call stack in javascript runtime

Stack implementation

1,push(element)

Add an element to the top of the stack

2.pop()

Remove the top most element from the stack

3.Peek()

To get the top value of the element without removing it

4.isEmpty()

Check if the stack is empty

5.size()

Get the number of elements in the stack

6. print()

Visualize the elements in the stack

```
class stack{
    constructor()
    {
        this.item=[]
    }
    push(element)
    {
        this.item.push(element)
    }
    pop()
    {
        return this.item.pop()
    }
    peek()
    {
        return this.item[this.item.length-1]
    }
    isEmpty()
    {
        return this.item.length===0
    }
    size()
    {
        return this.item.length
    }
```

```
    print()
    {
        console.log(this.item.toString())
    }
}

const s=new stack()
console.log(s.isEmpty())
s.push(10)
s.push(20)
s.push(30)
console.log(s.size())
s.print()
console.log(s.pop())
console.log(s.peek())
```

Why we need stack when arrays caan do all these operations

Because Array is an indexed list that allows random read and write operation But stacks works on LIFO principle .so insertion and removal of elements in stack is always constant time complexity where in array it has linear time complexity

Queue:
A Queue is a linear data structure that is open at both ends which follows FIFO approach in its individual operations.Data insertion is done at one end rear end or tail while deletion is done at the other end called or the front end or the head of the queue
Applications of queue:

1.people on a escalator
2.Cashier line in a store
3.Car wash line
4.One way exits

Different types of queue

1.Simple Queue:
2.Circular Queue
3.Priority Queue
` 4. Double ended Queue

Simple Queue:

It is the most basic queue in which the insertion of an item is done at the front of the queue and deletion takes place at the end of the queue.

- Ordered collection of comparable data kinds.
- Queue structure is FIFO (First in, First Out).
- When a new element is added, all elements added before the new element must be deleted in order to remove the new element.

Circular Queue

A circular queue is a special case of a simple queue in which the last member is linked to the first. As a result, a circle-like structure is formed.

- The last node is connected to the first node.
- Also known as a Ring Buffer as the nodes are connected end to end.
- Insertion takes place at the front of the queue and deletion at the end of the queue.
- Circular queue application: Insertion of days in a week.

Priority Queue

In a priority queue, the nodes will have some predefined priority in the priority queue. The node with the least priority will be the first to be removed from the queue. Insertion takes place in the order of arrival of the nodes.
Some of the applications of priority queue:

- Dijkstra's shortest path algorithm
- Prim's algorithm
- Data compression techniques like Huffman code

Deque (Double Ended Queue)

In a double-ended queue, insertion and deletion can take place at both the front and rear ends of the queue.

Basic Queue Operations in Queue Data Structure

Operation	Description
enqueue()	Process of adding or storing an element to the end of the queue
dequeue()	Process of removing or accessing an element from the front of the queue

peek()	Used to get the element at the front of the queue without removing it
initialize()	Creates an empty queue
isfull()	Checks if the queue is full
isempty()	Check if the queue is empty

Queue implementation

```
class queue{
    constructor()
    {
        this.items=[]
    }

    enqueue(element)
    {
        this.items.push(element)
    }
    dequeue()
    {
        return this.items.shift()
    }
    isEmpty()
    {
        return this.items.length==0
    }
    peek()
    {   if(this.items){
        return this.items[0]
    }
    return null

    }
    size()
```

```
    {
        return this.items.length
    }
    print()
    {
        console.log(this.items.toString())
    }



}
const q=new queue()
console.log(q.isEmpty())
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
console.log(q.size())
console.log(q.isEmpty())
console.log(q.peek())
console.log(q.dequeue())
```

Queue implementation using object to optimise

```
class queue{
    constructor()
    {
        this.items={}
        this.rear=0
        this.front=0
    }
    enqueue(element)
    {
        this.items[this.rear]=element
        this.rear++
    }
    dequeue()
    {
        const items=this.items[this.front]
        delete this.items[this.front]
        this.front++
        return items
```

```
    }
    isEmpty()
    {
        return this.rear-this.front==0
    }
    peek()
    {
        return this.items[this.front]
    }

    size()
    {
    return this.rear-this.front
    }
    print()
    {
        console.log(this.items)
    }


}
const q=new queue()
console.log(q.isEmpty())
q.enqueue(90)
q.enqueue(80)
q.enqueue(77)
q.print()
console.log(q.size())
console.log(q.peek())
console.log(q.dequeue())
```

Circular queue:

Hash Table:

Hash table also known a s Hashmap is a data structure used to store key value pairs

A hash table is a data structure that allows efficient insertion, deletion, and lookup of key-value pairs. The keys are usually strings, but they can be any other data type that can be hashed. The values can be any data type.

When a key-value pair is inserted into a hash table, the key is hashed using a hash function, which generates an index in the hash table's underlying array. This index is used to store the value. When a lookup is performed on a key, the key is again hashed to generate an index, which is then used to retrieve the corresponding value from the hash table.

Hash functions are designed to generate a unique index for each key-value pair, but in practice, collisions can occur where different keys hash to the same index. To handle collisions, different techniques can be used, such as chaining or open addressing.