HOW TO EMAIL A RESUME

Check the Templates

1.

Sample email to send resume to recruiter

Dear (Recruiter name)

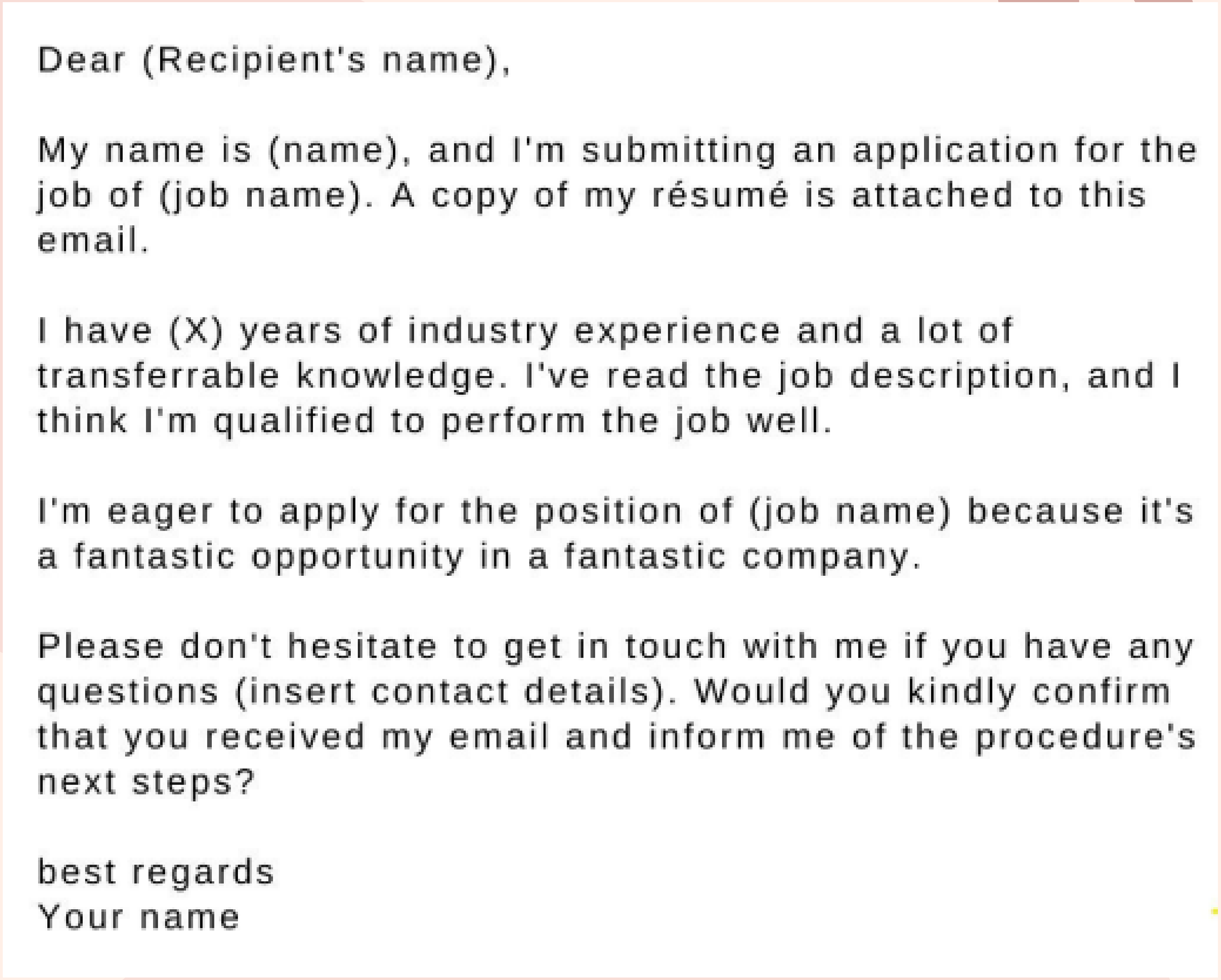
I am (name) and I'm interested in the post of (job name). My résumé is attached to this email.

After reading the job description, I believe I possess all the necessary abilities and credentials. Details about my current employment and previous roles are also available.

Can you describe the steps in the process and how they will be completed?

You can reach me at on (insert details). I'm interested in your response regarding how to proceed with my application.
Your name

2. Sample email to send resume to job



Dear (Recipient's name),

My name is (name), and I'm submitting an application for the job of (job name). A copy of my résumé is attached to this email.

I have (X) years of industry experience and a lot of transferrable knowledge. I've read the job description, and I think I'm qualified to perform the job well.

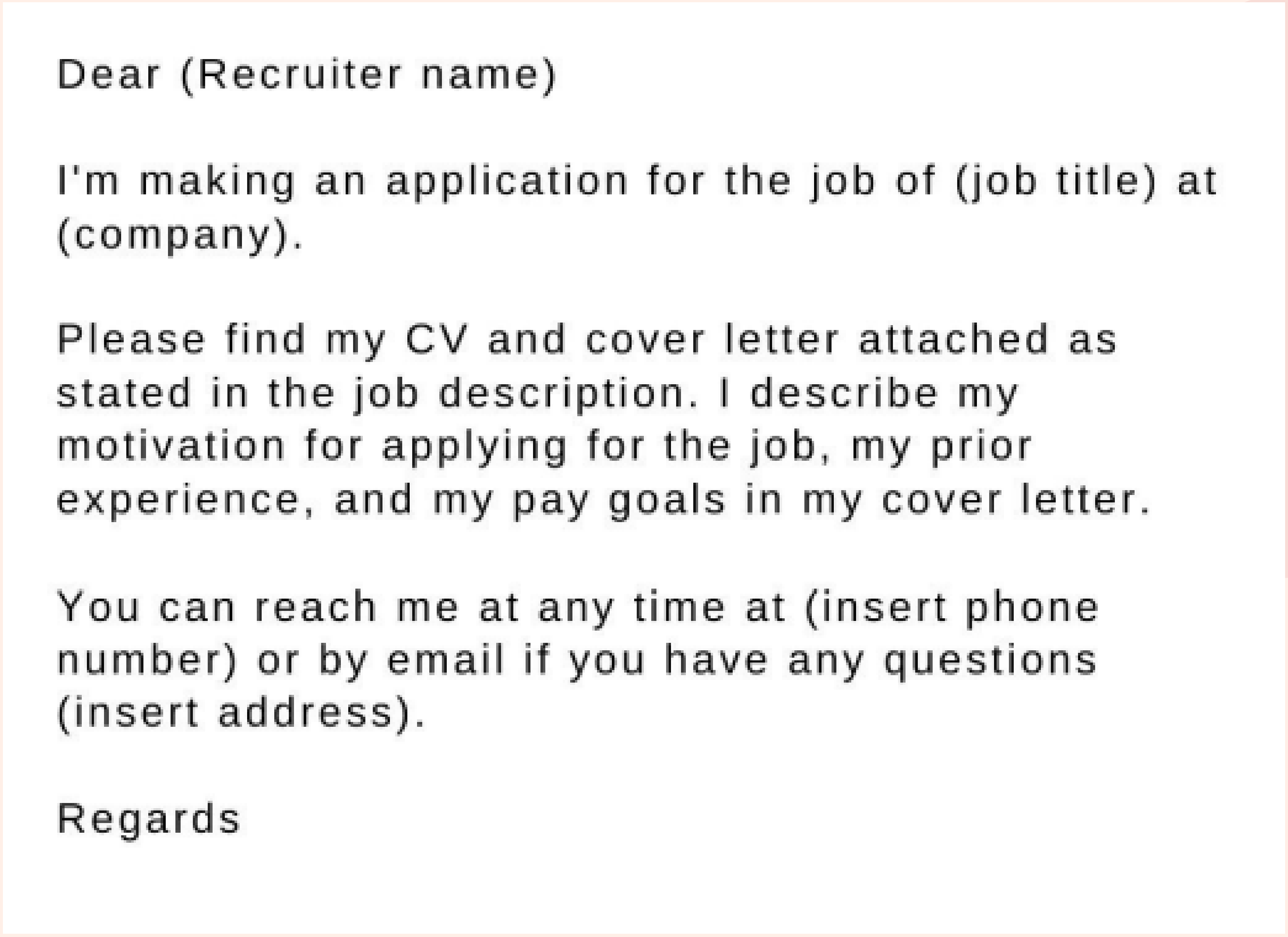
I'm eager to apply for the position of (job name) because it's a fantastic opportunity in a fantastic company.

Please don't hesitate to get in touch with me if you have any questions (insert contact details). Would you kindly confirm that you received my email and inform me of the procedure's next steps?

best regards
Your name

3.

Sample email cover letter with attached resume



Dear (Recruiter name)

I'm making an application for the job of (job title) at (company).

Please find my CV and cover letter attached as stated in the job description. I describe my motivation for applying for the job, my prior experience, and my pay goals in my cover letter.

You can reach me at any time at (insert phone number) or by email if you have any questions (insert address).

Regards

4. Thank you for considering my resume email template

Dear (Recruiter name)

Thank you for taking a look at my application for the post of (position name) at (company name).

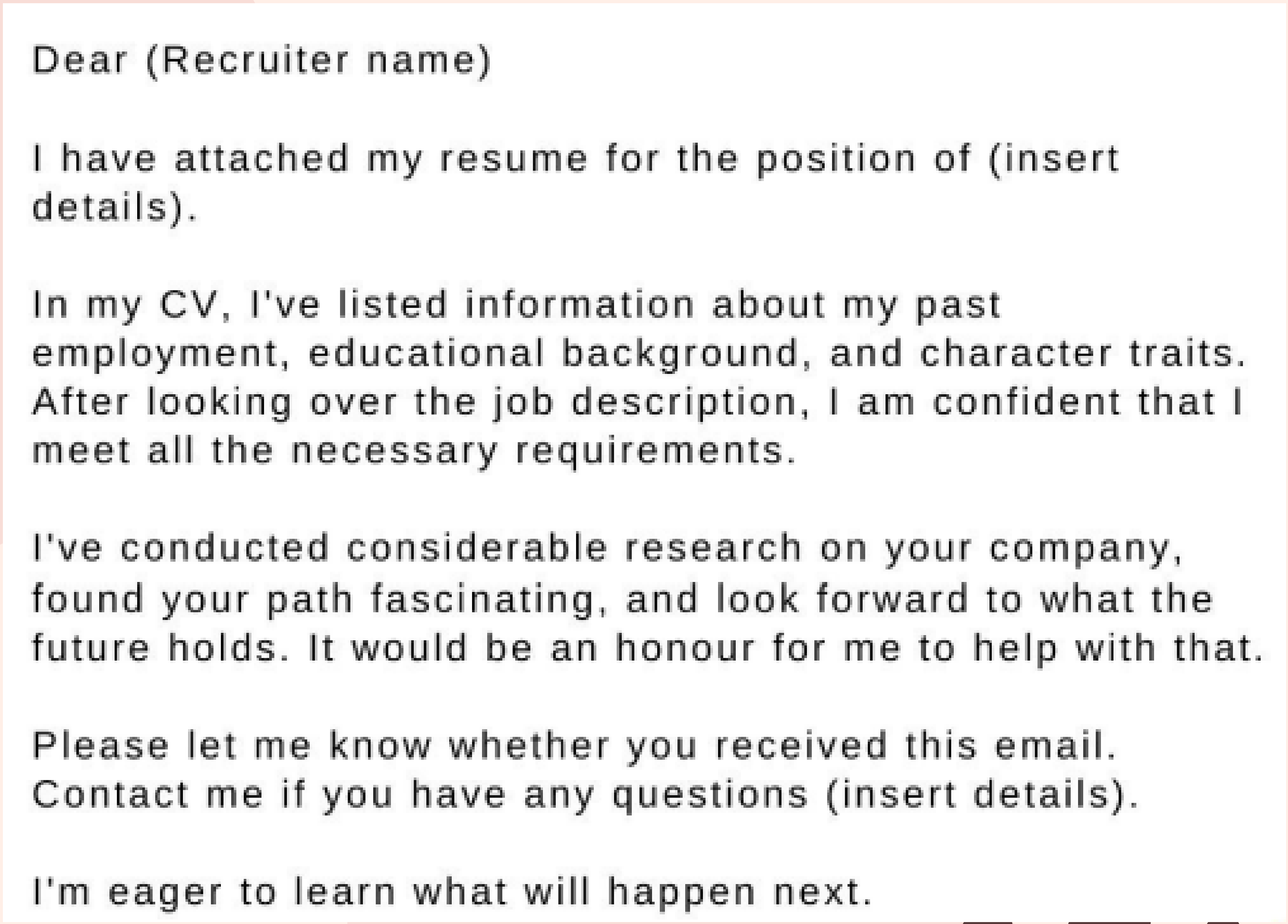
Even though I was unhappy that I wasn't selected for the interview, I can see why. If it's feasible, I'd like to continue in touch with you, so please send me information on any upcoming opportunities you think I could be a good fit for.

I prefer to be reached by email at (insert details)

Please feel free to keep my resume on file and don't be afraid to get in touch with me if you have any inquiries.

5.

Best email template for sending a resume by email



Dear (Recruiter name)

I have attached my resume for the position of (insert details).

In my CV, I've listed information about my past employment, educational background, and character traits. After looking over the job description, I am confident that I meet all the necessary requirements.

I've conducted considerable research on your company, found your path fascinating, and look forward to what the future holds. It would be an honour for me to help with that.

Please let me know whether you received this email. Contact me if you have any questions (insert details).

I'm eager to learn what will happen next.

6.

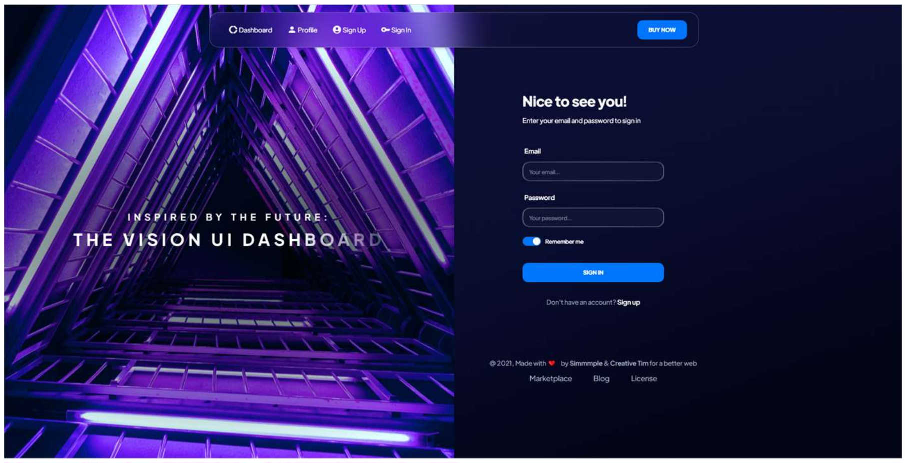
Thank You email post job offer

THANK YOU FOR OFFERING ME THE OPPORTUNITY TO WORK AT [COMPANY NAME] AS [JOB TITLE].
I SINCERELY APPRECIATE YOUR TIME AND CONSIDERATION.
I'M HAPPY TO ACCEPT THE [JOB TITLE] POSITION. AS WE MOVE FORWARD, I'D LIKE TO CLARIFY [YOUR QUESTIONS ABOUT THE POSITION].
PLEASE CONTACT ME AT [YOUR CONTACT INFORMATION] TO ESTABLISH THE NEXT STEPS.
THANK YOU AGAIN FOR THIS OPPORTUNITY.
I'M EXCITED TO WORK WITH THE [COMPANY NAME] TEAM SOON.

Kiran P. Bharambe

Save these templates for later!

Design UI screensas per the given screens



Nice to see you!

Enter your email and password to sign in

Email

Your email...

Password

Your password...

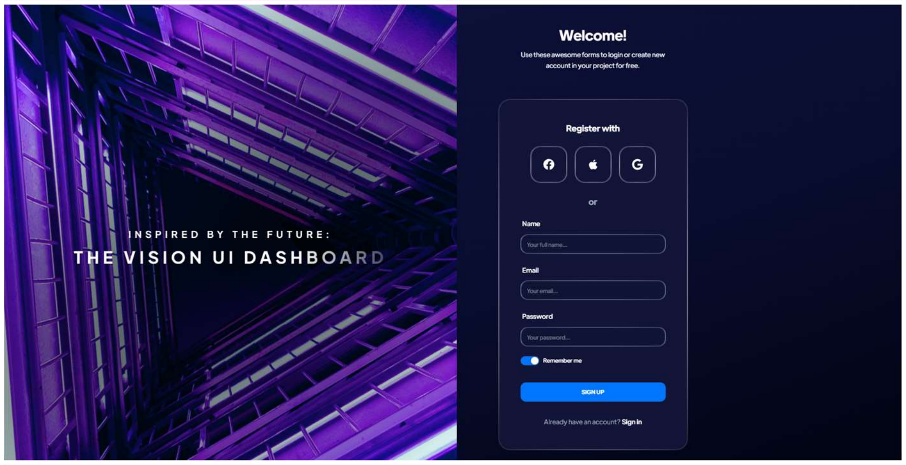
Remember me

SIGN IN

Don't have an account? **Sign up**

@ 2021, Made with ❤ by **Simmmple** & **Creative Tim** for a better web

Marketplace Blog License



Welcome!

Use these awesome forms to login or create new account in your project for free.

Register with

or

Name

Your full name...

Email

Your email...

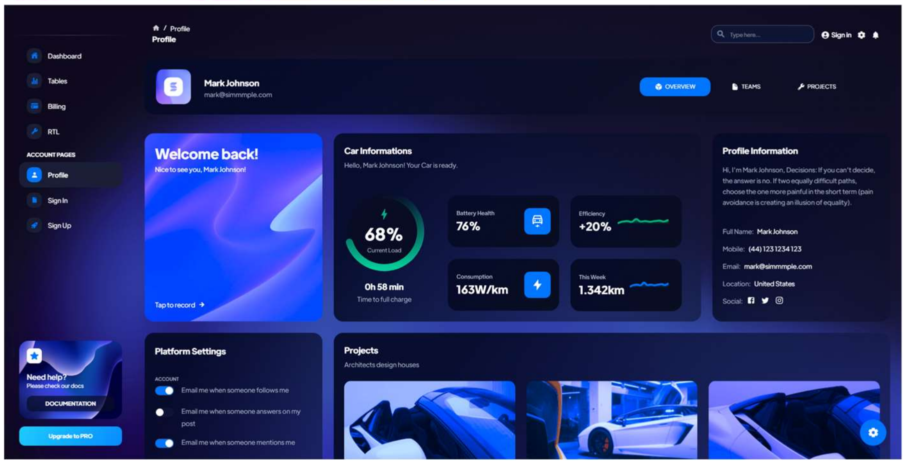
Password

Your password...

Remember me

SIGN UP

Already have an account? **Sign in**



/ Profile

Profile

Mark Johnson
mark@simmmple.com

Welcome back!

Nice to see you, Mark Johnson!

Tap to record →

Car Informations

Hello, Mark Johnson! Your Car is ready.

68%
Current Load

0h 58 min
Time to full charge

Battery Health 76%

Efficiency +20%

Consumption 163W/km

This Week 1.342km

Profile Information

Hi, I'm Mark Johnson, Decisions: If you can't decide, the answer is no. If two equally difficult paths, choose the one more painful in the short term (pain avoidance is creating an illusion of equality).

Full Name: Mark Johnson

Mobile: (44) 123 1234 123

Email: mark@simmmple.com

Location: United States

Social:

Platform Settings

ACCOUNT

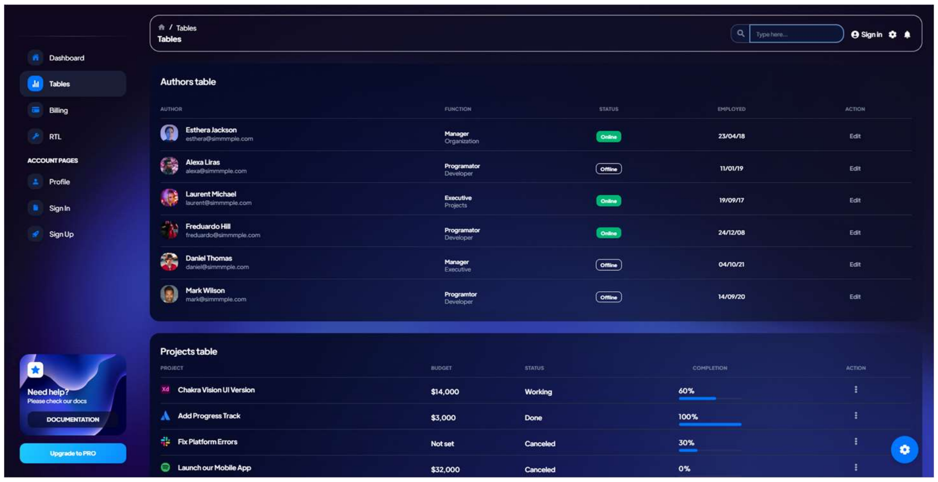
Email me when someone follows me

Email me when someone answers on my post

Email me when someone mentions me

Projects

Architects design houses



Authors table

AUTHOR	FUNCTION	STATUS	EMPLOYED	ACTION
Esthera Jackson esthera@simmmple.com	Manager Organization	Online	23/04/18	Edit
Alexa Liras alexa@simmmple.com	Programator Developer	Offline	11/01/19	Edit
Laurent Michael laurent@simmmple.com	Executive Projects	Online	19/09/17	Edit
Freduardo Hill freduardo@simmmple.com	Programator Developer	Online	24/12/08	Edit
Daniel Thomas daniel@simmmple.com	Manager Executive	Offline	04/10/21	Edit
Mark Wilson mark@simmmple.com	Programtor Developer	Offline	14/09/20	Edit

Projects table

PROJECT	BUDGET	STATUS	COMPLETION	ACTION
Chakra Vision UI Version	$14,000	Working	60%	⋮
Add Progress Track	$3,000	Done	100%	⋮
Fix Platform Errors	Not set	Canceled	30%	⋮
Launch our Mobile App	$32,000	Canceled	0%	⋮

API: need to integrate api of login, signup and list of user data.

1. Permutation Question

Input	**Output**
3	1 2 3
[1 2 3]	1 3 2
	2 1 3
	2 3 1
	3 1 2
	3 2 1

```
function permutations(arr, prefix = "") {
  if(arr.length===0){
      console.log(prefix)
      return;
  }
  for(let i=0; i<arr.length; i++){
      let num= arr[i] //1
      let left = arr.slice(0,i);//[]
      let right = arr.slice(i+1) //2,3
      let final= [...left, ...right] //2,3
      permutations(final, prefix + num + " ")
  }
}
function runProgram(read) {
  read= read.split("\n")
  var n= +read[0]
  var arr= read[1].split(" ").map(Number)
  permutations(arr)
}
```

2. Moving zero Push Zero’s of an array the end of the array without changing non zero elements order.

Input	**Output**
1	
5	1 12 3 0 0
0 1 0 12 3	

```
function runProgram(read)
{
    read= read.split("\n");
    let tc= read[0];
    let l=1;
    for(i=0; i<tc; i++){
        let n= read[l++];
        let arr= read[l++].split(" ").map(Number)
        zero(arr)
    }
}
```

```
function zero(arr){
    let bag=[];
    for(let i=0; i<arr.length; i++){
        if(arr[i]!==0){
            bag.push(arr[i])
        }
    }
    for(let j=0; j<arr.length; j++){
        if(arr[j]===0){
            bag.push(arr[j])
        }
    }
    console.log(bag.join(" "))
}
```

3.Happy number

input **output**

```
2           Yes
19          No
2
```

```
function runProgram(read)
{
    read= read.split("\n")
    let tc = +read[0]
    let line =1;
    for(let i=0;i<tc;i++){
      let num = +read[line++]
      myFun(num)
  }
}
function myFun(num){
    if(num<=1){
       console.log("Yes")
       return;
    }
   num = num+""
   let  n = num.split("").map(Number)
    if(n.length<2 && n[0]<10){
        console.log("No")
        return;
    }
    let sum =0;
    for(let i=0;i<n.length;i++){
        sum = sum+n[i]**2
    }
    myFun(sum)
}
```

4. Palindrome List:

```
Input              output
3
1                   true
2
1
```

```
var isPalindrome = function (head) {
    if(head==null)
    return true
    let curr = head;
    let arr = []
    while(curr){
        arr.push(curr.data);
        curr = curr.next;
    }
    let l = 0;
    let r = arr.length-1;
    while(l<r){
        if(arr[l]!==arr[r]){
            return false;
        }
        l++;
        r--;
    }
    return true;
};
```

5: Unique Gifts:

```
Input                 output
2
abadbc                aabbdd
abcabc                aaabc#
```

```
function UniqueGift(str){
    let obj ={}
    for(let i =0;i<str.length;i++){
        let temp = str[i]
        if(obj[temp] == undefined){
            obj[temp]=0
        }
    }
    let que = []
    let ans =[]
    for(let j =0;j<str.length;j++){
        obj[str[j]]++;
        que.push(str[j]);
        while(que.length != 0){
            if(obj[que[0]]==1){
                break;
            }
            que.shift();
        }
        if(que.length ==0){
            ans+="#"
        }else{
            ans+=que[0]
        }
    }
    console.log(ans)
}
```

6: Reverse String

```
Input       output
3
ab          ba
cd          dc
ef          fe
```

Method 1-

```
function reverse(str){
   let arr = [];
   for(let i=0; i<str.length; i++){
      arr.push(str[i]);
   }
   let new_str = ""
   while(arr.length!=0){
      new_str+=arr.pop();
   }
```

```
    console.log(new_str);
}
function runProgram(read){
    let input = read.trim().split("\n");
    let n = +input[0];
    let l = 1;
    for(let i=0; i<n; i++){
        let str = input[l++];
        reverse(str);
    }}
```

Method 2-

```
function reverseString(str) {
  const rev = [];
  for (let i = str.length - 1; i >= 0; i--) {
    rev.push(str[i]);
  }
  return rev.join("");
}

const reversedString = reverseString("hello");
console.log(reversedString);
```

7. Longest consecutive setbits

```
Input               output
4                   1
1                   0
0                   32
4294967295          2
13
```

```
function runProgram(input) {
input=input.split("\n")
let tc=input[0]
let line=1
for(let i=0; i<tc; i++){
let n=+input[line++]
longestConsecutiveSetBit(n)
}  }
function longestConsecutiveSetBit(n) {
  let count = 0;
  let maxCount = 0;
  while (n > 0) {
     if ((n & 1) === 1) {
      count++;
      maxCount = Math.max(maxCount, count);
    } else {
      count = 0;
    }
    n = n >> 1;
  }
  console.log(maxCount);
}
```

8. Spiral traversal matrix.

input **output**

```
4
1 2 3 4          1 2 3 4 8 4 8 7 6 5 1 5 6 7 3 2
5 6 7 8
1 2 3 4
5 6 7 8
```

```
function spirallyTraversingAMatrix(N, matrix){
    let top = 0,
    left=0,
    right=N-1,
    bottom=N-1;
    let moves = N*N;
    let res = "";
    while(moves>0){
        for(let i=left; i<=right; i++){
            res+=matrix[top][i]+" "; //1 2 3 4 8 4 8
            moves--;
        }
        top++;
        for(let i=top; i<=bottom; i++){
            res+= matrix[i][right]+" ";
            moves--;
        }
        right--;
        for(let i=right; i>=left; i--){
            res+= matrix[bottom][i]+" ";
            moves--;
        }
        bottom--;
        for(let i=bottom; i>=top; i--){
            res+= matrix[i][left]+" ";
            moves--;
        }
        left++;
    }
    console.log(res);
}
function runProgram(input)
{
    var input= input.split("\n")
    Var N= +input[0]
    let matrix=[]
    for(i=1;i<=size;i++){
        matrix.push(input[i].trim().split(" ").map(Number))
    }
    spirallyTraversingAMatrix(N,matrix)
}
```

9.Fatorial recursion

```
input                output
5                    120
```

```
function fatorialRecursion(input){
    let res=1;
    if(input>=1){
        for(i=input; i>=1; i--){
            res*=i
        }
    }
    return res
}
function runProgram(input)
{
   var  input= +input
    console.log(fatorialRecursion(input))
}
```

10.Anagram detector

```
input                       oupput
anagram
nag a ram            true
```

```
function runProgram(input) {
    input = input.split("\n");
    var s1 = input[0];
    var s2 = input[1];
    var x = s2.split("").sort().join("").trim();
    var y = s1.split("").sort().join("").trim();

    if (x == y)
        console.log("True");
    else
        console.log("False");
}
```

11. Merge two sort array.

```
input                                   output
let arr1=[2,5,7,8,4,3]                  1,1,1,2,2,2,3,3,4,4,4,5
let arr2=[42,15,16,12,9,44]
```

```
function mergesortarray(n,arr1,arr2){
    const mergedArr = [...arr1, ...arr2];
  for (let i = 0; i < mergedArr.length; i++) {
    for (let j = 0; j < mergedArr.length - i - 1; j++) {
      if (mergedArr[j] > mergedArr[j + 1]) {
           const temp = mergedArr[j];
        mergedArr[j] = mergedArr[j + 1];
        mergedArr[j + 1] = temp;
      }
    }
  }
  console.log(mergedArr.join(“ ”);
}
function runProgram(input)
{
    input= input.split("\n");
    let n= input[0];
    arr1=input[1].split(" ").map(Number)
    arr2=input[2].split(" ").map(Number)
    mergesortarray(n,arr1,arr2)

}
```

12. Number of ways

```
input                    ouput
4                        7
```

```
function numberofWays(input){
   if(input === 0){
      return 1
   }
   else if(input<0){
      return 0
   }
   else{
      return numberofWays(input-1)+numberofWays(input-
2)+numberofWays(input-3);
   }
}
function runProgram(input)
{
  var  input= +input
   console.log(numberofWays(input))
}
```

13.Implement queue using stack.

```
input                    ouput
6
0  1                     1
0  2                     2
0  3
1
2
1
```

```
class Queue {
   constructor()
   {
      this.S1 = new Stack()
      this.S2 = new Stack()
   }
```

```
    push(value) {
                while(!this.S1.isEmpty()){
            this.S2.push(this.S1.top());
            this.S1.pop();
        }
        this.S1.push(value);
        while(!this.S2.isEmpty()){
            this.S1.push(this.S2.top());
            this.S2.pop();
        }
    }
    pop() {
        this.S1.pop();
    }
    front() {
        return this.S1.top()
    }
    isEmpty() {
        return this.S1.isEmpty();
    }}
```

14. Balence parentheses

```
input                          ouput
((((((((()))))))))             balanced
```

```
function myFun(str){
    const stack = [];
  for (let i of str) {
    if (i === "(" || i === "[" || i === "{") {
      stack.push(i);
    } else if (i === ")" || i === "]" || i === "}") {
      if (stack.length === 0) {
        return "unbalanced";
      }
      let top = stack[stack.length - 1];
      if ((i === ")" && top !== "(") || (i === "]" && top !== "[") || (i === "}" && top !== "{")) {
        return "unbalanced";
    }
    stack.pop();
   }
  }
 return stack.length === 0 ? "balanced" : "unbalanced";
}
function runProgram(read){
        Let str= read
    myFun(str);
}
```

15. Next greater node linklist

```
input               output
2 1 5               5 5 0
```

```
const nextLargerNodes = function(head) {
  const A = []
  while (head != null) A.push(head.val), (head = head.next)
  const res = new Array(A.length).fill(0)
  const stack = []
  for (let i = 0; i < A.length; i++) {
    while (stack.length && A[stack[stack.length - 1]] < A[i])
      res[stack.pop()] = A[i]
    stack.push(i)
  }
  return res
}
```

16.Transformation in education.

```
input                            output
A Transformation in education    education in Transformation A
```

```
function reverseString(str) {
    let arr =str.split(" ")
    let bag= []
    for(i=arr.length-1; i>=0; i--){
       bag.push(arr[i])
    }
    console.log(bag.join(" "))
}
function runProgram(read)
{
    let str=read
    reverseString(str)}
```

17. Roman Numbers.

```
function toRomanNumeral(str) {
  const roman = {
    'I': 1,
    'V': 5,
    'X': 10,
    'L': 50,
    'C': 100,
    'D': 500,
    'M': 1000
  };
  let dec = 0;
  for (let i = 0; i < str.length; i++) {
    let cur = roman[str[i]];
    let next = roman[str[i + 1]];
    if (next && cur < next) {
      dec += next - cur;
      i++;
    } else {
      dec += cur;
    }
  }
  console.log(dec) ;
}
function runProgram(read)
{
    let str=read
    toRomanNumeral(str)
}
```

18. Binary matrix

```
input                 output
3 2
1 0          0 1
0 1          1 0
```

```
1 1          0 0
function binaryMatrix(N, M, matrix) {
    for(let i=0; i<N; i++){
        let row=""
        for(let j=0; j<M; j++){
            if(matrix[i][j]===0){
                row+= '1 '
            }else{
                row+= '0 '
            }
        }
        console.log(row)
    }
}
```

19.Remove upper case.

```
input                output
MasAi        asi
function runProgram(input) {
input=input.split("\n")
let N=+input[0];
let str=input[1];
removeUppercase(N, str)
}
function removeUppercase(N, str) {
  var lower ="abcdefghijklmnopqrstuvwxyz";
  var count= "";
  for(var i=0; i<N; i++){
      var x = str[i];
      for(var j=0; j<lower.length; j++){
          if(x===lower[j]){
              count = count + x;
          }
      }
  }
   console.log(count);
}
```

20. Transpose matrix.

```
function transpose(N, M, matrix){
    for(j=0; j<M; j++){
        let bag="";
        for(i=0; i<N; i++){
            bag+=matrix[i][j]+""
    }
    console.log(bag)
    }
}
function runProgram(input)
{
    var input= input.trim().split("\n")
    var x= input[0].split(" ").map(Number)
    let N= x[0]
    let M= x[1]
    let matrix=[]
    let line=1;
    for(i=1;i<=n;i++){
        matrix.push(input[line].trim().split(" ").map(Number))
        line++;
    };
    transpose(N,M,matrix)
}
```

DSA LONG QUESTIONS

1) Happy Numbers

2) Palindrome List (Check Palindrome Using Linked List)

3) Balanced Parenthesis(Again a classical Problem)

4) Unique Gifts

5) Binary Bits (Convert to Binary and count the continuous number of 1 in the matrix)

6)Transpose the matrix

7)Flip the Binary Matrix

8)Reverse String

9)Spiral Traversal (clockwise and anti-clockwise)

10) Factorial N

11) N-Queens

12) Employees

13)Merge two arrays in sorted form

14)Number of ways

15)Betty buys a present

16)Make Leaderboard-imp

----

1) The time complexity of push, pop, and peek?

Ans- The time complexities of basic stack operations are as follows:

1. Push: The time complexity of the push operation in a stack is O (1), which means it takes constant time. This is true for most stack implementations, including arrays and linked lists. When pushing an element onto the stack, it involves

adding the element to the top of the stack, which can be done in constant time regardless of the stack size.

2. Pop: The time complexity of the pop operation in a stack is also O (1), indicating constant time. When popping an element from the stack, it involves removing the top element and updating the stack's structure accordingly. This operation can be done in constant time, regardless of the number of elements in the stack.
3. Peek: The time complexity of the peek operation in a stack is O (1), which means it also takes constant time. The peek operation allows you to view the element at the top of the stack without removing it. Since the top element is always readily available, accessing it takes constant time.

In summary, the push, pop, and peek operations in a stack have a time complexity of O (1), meaning they can be performed in constant time.

2)What is the time complexity of enqueue, dequeue, and peak?

The time complexity of enqueue, dequeue, and peek operations on a queue are all O(1). This means that no matter how many elements are in the queue, these operations will always take the same amount of time to complete.

- Enqueue is the operation of adding an element to the end of the queue. This can be done by simply adding the element to the end of the linked list or array that represents the queue.
- Dequeue is the operation of removing an element from the beginning of the queue. This can be done by simply removing the element at the beginning of the linked list or array that represents the queue.
- Peek is the operation of looking at the element at the beginning of the queue without removing it. This can be done by simply accessing the element at the beginning of the linked list or array that represents the queue.

It is important to note that the time complexity of enqueue, dequeue, and peek operations on a queue are only O(1) if the queue is implemented correctly. If the queue is implemented incorrectly, these operations may take longer to complete. For example, if a queue is implemented as a linked list and the linked list is not properly maintained, then enqueue and dequeue operations may take O(n) time to complete.

Here are some additional details about the time complexity of enqueue, dequeue, and peek operations on a queue:

- Enqueue: The time complexity of enqueue is O(1) because it only requires a single pointer manipulation.
- Dequeue: The time complexity of dequeue is O(1) because it only requires a single pointer manipulation.
- Peek: The time complexity of peek is O(1) because it only requires a single pointer manipulation.

3)What is the meaning of time complexity and Space Complexity and how to calculate time and space complexity?

Time complexity and space complexity are two important measures of the efficiency of an algorithm.

- Time complexity measures how long an algorithm takes to run, in terms of the size of the input.
- Space complexity measures how much memory an algorithm uses, in terms of the size of the input.

Time complexity is typically measured using big O notation. Big O notation is a way of describing the asymptotic behavior of a function. In other words, it describes how the function behaves as the input size gets larger and larger.

There are three main types of time complexity:

- O(1) time complexity means that the algorithm takes the same amount of time to run, regardless of the size of the input.
- O(n) time complexity means that the algorithm takes time proportional to the size of the input.
- O(n^2) time complexity means that the algorithm takes time proportional to the square of the size of the input.

Space complexity is also typically measured using big O notation. The three main types of space complexity are:

- O(1) space complexity means that the algorithm uses a constant amount of space, regardless of the size of the input.

- O(n) space complexity means that the algorithm uses space proportional to the size of the input.
- O(n^2) space complexity means that the algorithm uses space proportional to the square of the size of the input.

Calculating time and space complexity can be a complex task, but there are a few general principles that can be used.

- For time complexity, start by identifying the most expensive operation in the algorithm. Then, estimate how many times that operation will be executed, in terms of the size of the input. The time complexity of the algorithm will be the same as the time complexity of the most expensive operation.
- For space complexity, start by identifying the amount of space that each variable in the algorithm uses. Then, add up the amount of space used by all of the variables. The space complexity of the algorithm will be the same as the total amount of space used by all of the variables.

It is important to note that time and space complexity are only two measures of the efficiency of an algorithm. There are other factors that can also affect the efficiency of an algorithm, such as the specific implementation of the algorithm and the hardware that the algorithm is running on.

4)What is the common uses of 2D array in Programing?

There are many common uses of 2D arrays in programming. Some of the most common uses include:

- Storing data in a tabular format. For example, a 2D array can be used to store a table of student grades, a table of inventory items, or a table of weather data.
- Representing a grid. For example, a 2D array can be used to represent a game board, a city map, or a maze.
- Solving linear equations. A 2D array can be used to represent the coefficients of a system of linear equations.
- Performing matrix operations. A 2D array can be used to represent a matrix. Matrix operations are used in many different areas of mathematics and science, such as linear algebra, statistics, and physics.
- Image processing. 2D arrays are used to represent images. Image processing is a field of computer science that deals with the manipulation of images.

- Natural language processing. 2D arrays are used to represent text. Natural language processing is a field of computer science that deals with the interaction between computers and human (natural) languages.

These are just a few of the many common uses of 2D arrays in programming. 2D arrays are a versatile data structure that can be used to solve a wide variety of problems.

5)What are the benefits and Limitations of Two Pointer technique?

The two pointer technique is a simple and efficient algorithm that can be used to solve a variety of problems. It is particularly useful for problems that involve finding pairs or sequences of elements in a sorted array.

The two pointer technique works by using two pointers, one that starts at the beginning of the array and one that starts at the end of the array. The pointers are then moved in opposite directions until they meet or until a match is found.

The two pointer technique has several benefits, including:

- It is a simple and easy-to-understand algorithm.
- It is efficient and can be used to solve problems quickly.
- It can be used to solve a variety of problems.

However, the two pointer technique also has some limitations, including:

- It can only be used to solve problems on sorted arrays.
- It may not be the most efficient algorithm for all problems.
- It may not be able to solve all problems.

Overall, the two pointer technique is a powerful and versatile algorithm that can be used to solve a variety of problems. It is simple to understand and efficient, but it may not be the most efficient algorithm for all problems.

6) What are the Benefits of Linked List over Array List?

There are several benefits of using a linked list over an array list.

- Dynamic size: A linked list can grow or shrink dynamically, as needed. This is in contrast to an array list, which has a fixed size.
- Efficient insertion and deletion: Insertion and deletion of elements in a linked list is very efficient. This is because elements can be added or removed from the beginning or end of the list in constant time. In an array list, insertion and deletion of elements can be expensive, especially if the element is not at the beginning or end of the list.
- Memory efficiency: Linked lists are more memory efficient than array lists. This is because linked lists do not need to store the size of the list. In an array list, the size of the list must be stored, which takes up additional memory.
- Ease of implementation: Linked lists are easier to implement than array lists. This is because linked lists are simpler data structures.

However, there are also some limitations to using a linked list.

- Slow random access: Random access to elements in a linked list is slow. This is because each element in a linked list is stored as a separate node. To access an element in the middle of a linked list, all of the elements before it must be traversed. In an array list, random access to elements is fast.
- Not contiguous in memory: Linked lists are not contiguous in memory. This means that the elements of a linked list are not stored next to each other in memory. In an array list, the elements of the list are stored next to each other in memory.

Overall, linked lists are a good choice for applications where dynamic size, efficient insertion and deletion, and memory efficiency are important. Array lists are a good choice for applications where random access is important.

7)Find the sum of digits of a number in its decimal representation

```
Def(n);

Ans=0;

While(n>0)

Ans +=n%10

N /=10;

Print(ans)
```

7)Can You provide an example how to implement a variable length sliding window algorithm?

```
function variableLengthSlidingWindow(array, windowSize) {
  // Initialize the sliding window.
  let window = [];

  // Iterate over the array.
  for (let i = 0; i < array.length; i++) {

    // Add the current element to the sliding window.
    window.push(array[i]);

    // If the sliding window is too large, remove the first element.
    if (window.length > windowSize) {
      window.shift();
    }

    // Keep track of the maximum value in the sliding window.
    let maxValue = Math.max(...window);

    // Yield the maximum value.
    yield maxValue;
  }
}
```

8) Practice custom sort i.w Bubble sort and Selection sort.

```
function bubbleSort(array) {
  // Initialize the swapped flag.
  let swapped = true;

  // Iterate over the array until no more swaps are needed.
  while (swapped) {

    // Set the swapped flag to False.
    swapped = false;

    // Iterate over the array from the beginning.
    for (let i = 0; i < array.length - 1; i++) {

      // If the current element is greater than the next element, swap them.
      if (array[i] > array[i + 1]) {
        [array[i], array[i + 1]] = [array[i + 1], array[i]];
        swapped = true;
      }
    }
  }

  return array;
}


function selectionSort(array) {
  // Iterate over the array from the beginning.
  for (let i = 0; i < array.length - 1; i++) {

    // Initialize the min_index to the current index.
    let min_index = i;

    // Iterate over the array from the current index to the end.
    for (let j = i + 1; j < array.length; j++) {

      // If the current element is less than the min_index element, update the min_index.
      if (array[j] < array[min_index]) {
        min_index = j;
      }
    }

    // Swap the current element and the min_index element.
    [array[i], array[min_index]] = [array[min_index], array[i]];
  }

  return array;
}
```

9)What are Stable sort and unstable sort? Explain using an Example.

Sure. A stable sort is a sorting algorithm that preserves the original order of equal elements. An unstable sort is a sorting algorithm that does not preserve the original order of equal elements.

For example, consider the following array:

JavaScript
```
const array = [1, 1, 2, 3, 4];
```
Use code with caution. Learn more
content_copy

If we sort this array using a stable sort, the result will be:

JavaScript
```
const sortedArray = [1, 1, 2, 3, 4];
```
Use code with caution. Learn more
content_copy

Notice that the two 1s in the original array are still next to each other in the sorted array.

If we sort this array using an unstable sort, the result could be:

JavaScript
```
const sortedArray = [1, 2, 1, 3, 4];
```
Use code with caution. Learn more
content_copy

Notice that the two 1s in the original array are not next to each other in the sorted array.

Here are some examples of stable sorting algorithms:

- Bubble sort
- Insertion sort
- Merge sort
- Timsort

Here are some examples of unstable sorting algorithms:

- Quick sort
- Selection sort
- Heap sort

In general, stable sorting algorithms are more complex than unstable sorting algorithms. However, stable sorting algorithms are often preferred when it is important to preserve the original order of equal elements.

10) Difference between the algorithm and Data Structure?

The main difference between an algorithm and a data structure is that an algorithm is a set of instructions that tells a computer how to solve a problem, while a data structure is a way of organizing data so that it can be easily accessed and manipulated.

An algorithm is a step-by-step procedure that solves a problem. It is a sequence of instructions that tell a computer what to do. Algorithms are used in many different areas of computer science, including artificial intelligence, machine learning, and data science.

A data structure is a way of organizing data so that it can be easily accessed and manipulated. Data structures are used to store and manage data in a computer program. There are many different types of data structures, each with its own advantages and disadvantages. Some common data structures include arrays, linked lists, trees, and graphs.

Algorithms and data structures are both important concepts in computer science. They are used together to solve problems and store data in a computer program.

11)Is recursion an algorithm?

Yes, recursion is an algorithm. It is a technique that allows a function to call itself. Recursion can be used to solve problems that are naturally recursive, such as finding the factorial of a number or the Fibonacci sequence.

Recursion is a powerful tool, but it can also be difficult to understand and implement. It is important to carefully design recursive algorithms to avoid infinite loops and stack overflows.

12)Difference Between Singly Linked List and Doubly Linked List?

A singly linked list is a data structure that consists of a sequence of nodes, where each node contains data and a pointer to the next node in the sequence. A doubly linked list is a data structure that consists of a sequence of nodes, where each node contains data, a pointer to the next node in the sequence, and a pointer to the previous node in the sequence.

The main difference between singly linked lists and doubly linked lists is that singly linked lists can only be traversed in one direction, while doubly linked lists can be traversed in both directions. This means that singly linked lists can only be accessed from the beginning, while doubly linked lists can be accessed from the beginning or the end.

Singly linked lists are simpler to implement than doubly linked lists, but they are less flexible. Doubly linked lists are more complex to implement, but they are more flexible.

Here is a table that summarizes the differences between singly linked lists and doubly linked lists:

Feature	Singly linked list
Direction of traversal	Forward
Implementation complexity	Simple
Flexibility	Less flexible

Singly linked lists are often used in situations where only forward traversal is needed, such as in linked lists and queues. Doubly linked lists are often used in situations where forward and backward traversal is needed, such as in trees and graphs.

13) What is Binary Search? Explain it using an Example.

Binary search is a search algorithm that finds the position of a target value within a sorted array. Binary search works by repeatedly dividing the search interval in half until the target value is found.

For example, consider the following sorted array:

Code snippet
```
[1, 3, 5, 7, 9]
```
Use code with caution. Learn more
content_copy

If we want to find the position of the value 5, we can use binary search as follows:

1. Start with the middle element of the array, which is 5.

2. If the target value is equal to the middle element, we have found it.
3. If the target value is less than the middle element, we know that it must be in the left half of the array.
4. If the target value is greater than the middle element, we know that it must be in the right half of the array.
5. Repeat steps 2-4 until the target value is found.

In this case, we would find the target value 5 in the first step.

Binary search is a very efficient algorithm, and it can find the position of a target value in a sorted array in logarithmic time. This means that the number of steps required to find the target value grows logarithmically with the size of the array.

Here is an example of binary search in JavaScript:

Code snippet
```
function binarySearch(array, target) {
  let low = 0;
  let high = array.length - 1;

  while (low <= high) {
    let mid = Math.floor((low + high) / 2);

    if (array[mid] === target) {
      return mid;
    } else if (array[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return -1;
}
```

14)Types of Sorting Algorithm.

There are many different types of sorting algorithms, each with its own advantages and disadvantages. Some of the most common sorting algorithms include:

- Bubble sort is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. Bubble sort is very inefficient, but it is easy to understand and implement.
- Selection sort is a sorting algorithm that works by repeatedly finding the smallest element in an array and swapping it with the first element. Selection sort is also inefficient, but it is slightly faster than bubble sort.
- Insertion sort is a sorting algorithm that works by repeatedly inserting elements into a sorted subarray. Insertion sort is more efficient than bubble sort and selection sort, but it is more complex to implement.
- Merge sort is a sorting algorithm that works by dividing an array in half and recursively sorting the two halves. Merge sort is very efficient, but it is more complex to implement than bubble sort, selection sort, and insertion sort.
- Quick sort is a sorting algorithm that works by selecting a pivot element and partitioning the array around the pivot element. Quick sort is very efficient, but it is not stable, meaning that it may change the relative order of equal elements.

There are many other sorting algorithms, each with its own advantages and disadvantages. The best sorting algorithm to use depends on the specific needs of the application.

15) Difference between Stack and Queue.

A stack is a linear data structure in which elements are added and removed from one end, called the top. A queue is a linear data structure in which elements are added and removed from opposite ends, called the front and rear.

In a stack, the last element added is the first element removed (LIFO), whereas in a queue, the first element added is the first element removed (FIFO).

Stacks are often used to implement backtracking, recursion, and undo/redo functionality. Queues are often used to implement scheduling, event handling, and message passing.

Here is a table that summarizes the differences between stacks and queues:

Feature	Stack	Queue
Data structure	Linear	Linear

Access	Last in, first out (LIFO)	First in, first out (
Implementation	Linked list or array	Linked list or arra
Applications	Backtracking, recursion, undo/redo	Scheduling, ever

16)What is recursion?

In computer science, recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Recursion solves such recursive problems by using functions that call themselves from within their own code.

Recursion is a powerful tool for solving problems that are naturally recursive, such as finding the factorial of a number or the Fibonacci sequence.

Recursion is a powerful tool, but it can also be difficult to understand and implement. It is important to carefully design recursive algorithms to avoid infinite loops and stack overflows.

Here is an example of a recursive function that finds the factorial of a number:

Code snippet
```
function factorial(n) {
  if (n === 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```
Use code with caution. Learn more
content_copy

This function works by recursively calling itself to find the factorial of n - 1. The base case is when n is 0, in which case the function returns 1.

Here is an example of a recursive function that finds the Fibonacci sequence:

Code snippet
```
function fibonacci(n) {
  if (n === 0) {
    return 0;
  } else if (n === 1) {
    return 1;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```
Use code with caution. Learn more
content_copy

This function works by recursively calling itself to find the Fibonacci numbers for n - 1 and n - 2. The base cases are when n is 0 or 1, in which case the function returns 0 or 1 respectively.

Recursion is a powerful tool that can be used to solve many different kinds of problems. It is important to understand the basics of recursion in order to be able to use it effectively.

17) What is the Bubble Sort algorithm?

Bubble sort is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. Bubble sort is very inefficient, but it is easy to understand and implement.

Here is an example of how bubble sort works:

Code snippet
```
// Start with an unsorted array.
let array = [5, 3, 1, 2, 4];

// Iterate through the array, comparing adjacent elements.
for (let i = 0; i < array.length - 1; i++) {
  // If the current element is greater than the next element, swap them.
  if (array[i] > array[i + 1]) {
    let temp = array[i];
    array[i] = array[i + 1];
    array[i + 1] = temp;
  }
}

// The array is now sorted.
console.log(array); // [1, 2, 3, 4, 5]
```

content_copy

Bubble sort is a very inefficient algorithm, and it is not recommended for use in applications where performance is critical. However, it is a good algorithm to understand because it is simple and easy to implement.

18)What is a Linked List?

In computer science, a linked list is a linear data structure in which the elements are not stored contiguously in memory. Instead, each element is a separate object, called a node, which contains two parts: data, and a pointer to the next node in the list. The first node in the list is called the head, and the last node is called the tail.

Linked lists are a dynamic data structure, which means that they can grow and shrink as needed. This makes them well-suited for applications where the number of elements is not known in advance, such as a linked list of files in a directory.

Linked lists are also a versatile data structure, and can be used to implement many different data structures, such as stacks, queues, and trees.

Here are some of the advantages of using linked lists:

- They are dynamic, meaning that they can grow and shrink as needed.
- They are easy to insert and delete elements from.
- They are easy to traverse, meaning that you can easily iterate through the elements in the list.

Here are some of the disadvantages of using linked lists:

- They can be slower than arrays for accessing elements at specific indexes.
- They can be more memory-intensive than arrays.
- They can be more difficult to implement than arrays.

19)Why do we use Queue Data Structure? Give an Example of Queue Data structure.

We use queue data structure because it follows the First In First Out (FIFO) principle.

This means that the first element that is added to the queue is the first element that is removed from the queue. This makes queue data structure a good choice for applications where you need to process elements in the order they were received, such as printing jobs, network requests, and message queues.

Here are some examples of queue data structures:

- Printing jobs: A printer can be thought of as a queue. The first job that is sent to the printer is the first job that is printed.
- Network requests: A web server can be thought of as a queue. The first request that is received by the web server is the first request that is processed.
- Message queues: A message broker can be thought of as a queue. The first message that is sent to the message broker is the first message that is delivered to the recipient.

Queue data structures are a versatile tool that can be used in a variety of applications. They are easy to implement and use, and they provide a reliable way to process elements in the order they were received.

20)def f();

Int a[n][n]

//find the sum of elements of amatrix that are above or on the diagonal.

```
Sum=0;
For i=1 to n;
For j=I to n;
Sum +=a[i][j]
Print (sum)
```

21)How did Dequeue differ from the queue?

22)def f()

```
Ans=0;

For i=1 to n;

For j=1 to log(i);

Ans +=1;

Print(ans)
```

23)What are the time Complexity of basic queue operations (enqueue,Dequeue,peek)?

The time complexity of basic queue operations depends on the implementation of the queue. In general, enqueue and dequeue operations have O(1) time complexity, while peek operation has O(1) time complexity in most implementations.

- Enqueue: The enqueue operation adds an element to the end of the queue. In most implementations, this can be done in constant time by simply incrementing the rear pointer and assigning the new element to the location pointed to by the rear pointer.
- Dequeue: The dequeue operation removes an element from the front of the queue. In most implementations, this can be done in constant time by simply decrementing the front pointer and returning the element pointed to by the front pointer.
- Peek: The peek operation returns the element at the front of the queue without removing it. In most implementations, this can be done in constant time by simply returning the element pointed to by the front pointer.

However, there are some implementations of queues where the time complexity of these operations may be different. For example, a queue implemented as a linked list may have O(n) time complexity for enqueue and dequeue operations, where n is the number of elements in the queue. This is because these operations may require traversing the entire linked list to find the appropriate location to add or remove an element.

It is important to note that the time complexity of these operations is only an average case. In the worst case, the time complexity may be higher. For example, if the queue is full, the enqueue operation may have O(n) time complexity, where n is the capacity of the queue. This is because the queue may need to resize itself to accommodate the new element.

Overall, the time complexity of basic queue operations is typically O(1). However, there are some implementations where the time complexity may be different. It is important to consider the specific implementation of the queue when choosing the appropriate operation.

24)What is merge sort? What are its advantages?

Merge sort is a divide-and-conquer algorithm for sorting arrays. It works by recursively dividing the array in half, sorting each half, and then merging the two sorted halves together.

Merge sort is a stable algorithm, which means that it preserves the original order of equal elements. It is also an in-place algorithm, which means that it does not require any additional space to sort the array.

Merge sort is a very efficient algorithm, with a worst-case time complexity of O(n log n). It is also a very simple algorithm to understand and implement.

Here are some of the advantages of using merge sort:

- It is a very efficient algorithm, with a worst-case time complexity of O(n log n).
- It is a stable algorithm, which means that it preserves the original order of equal elements.
- It is an in-place algorithm, which means that it does not require any additional space to sort the array.
- It is a very simple algorithm to understand and implement.

Here are some of the disadvantages of using merge sort:

- It can be slower than other sorting algorithms, such as quick sort, for small arrays.
- It can be more difficult to parallelize than other sorting algorithms.

Overall, merge sort is a very efficient and versatile sorting algorithm. It is a good choice for sorting large arrays, and it is also a good choice for sorting arrays that need to be sorted in a stable manner.

25)How does the Sliding Window algorithm work, and what are its key characteristics?

The sliding window algorithm is a technique for searching for a pattern in a stream of data. It works by maintaining a window of fixed size, and sliding the window over the data stream one element at a time. The algorithm checks each element in the window to see if it matches the pattern. If it does, the algorithm returns the index of the element in the stream. If it doesn't, the algorithm moves the window forward one element and repeats the process.

The sliding window algorithm is a very efficient algorithm for searching for patterns in a stream of data. It has a time complexity of O(n), where n is the length of the data stream. It is also a very simple algorithm to understand and implement.

Here are some of the key characteristics of the sliding window algorithm:

- It is a very efficient algorithm for searching for patterns in a stream of data.
- It is a very simple algorithm to understand and implement.
- It can be used to search for patterns of any length.
- It can be used to search for patterns in any type of data.

The sliding window algorithm is a very versatile algorithm that can be used to solve a variety of problems. It is a good choice for problems where you need to search for a pattern in a stream of data, such as real-time data analysis and network packet filtering.

26)What is Bubble sort what are the advantages over other sorting Algorithms?

Bubble sort is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. Bubble sort is very inefficient, but it is easy to understand and implement.

Here is an example of how bubble sort works:

Code snippet
```
// Start with an unsorted array.
let array = [5, 3, 1, 2, 4];

// Iterate through the array, comparing adjacent elements.
```

```
for (let i = 0; i < array.length - 1; i++) {
  // If the current element is greater than the next element, swap them.
  if (array[i] > array[i + 1]) {
    let temp = array[i];
    array[i] = array[i + 1];
    array[i + 1] = temp;
  }
}

// The array is now sorted.
console.log(array); // [1, 2, 3, 4, 5]
content_copy
```

Bubble sort is a very inefficient algorithm, and it is not recommended for use in applications where performance is critical. However, it is a good algorithm to understand because it is simple and easy to implement.

Here are some of the advantages of using bubble sort:

- It is a very simple algorithm to understand and implement.
- It is a stable algorithm, which means that it preserves the original order of equal elements.
- It is an in-place algorithm, which means that it does not require any additional space to sort the array.

Here are some of the disadvantages of using bubble sort:

- It is a very inefficient algorithm, with a worst-case time complexity of O(n^2).
- It can be slow for large arrays.
- It is not a good choice for real-time applications.

Overall, bubble sort is a simple and easy-to-understand sorting algorithm. It is not a good choice for performance-critical applications, but it is a good choice for educational purposes.

27)If n is the size of i/p ,Which function is the most efficient? In other words which loop complete first.

a) for(let i=0;i<n;i++)

b) for(let i=0;i<n;i++)

c) for(let i=0;i<n;i*2)

d) for(let i=n;i>-1;i/=2)