javascript函数

函数就是一个功能模块，函数式编程是一种面向过程的编程思想，如果遇到一个大的复杂问题，可以分解成几个部分，每个部分用专门的函数分解实现。

函数语法：

```
function functionName(parameters) {
  执行的代码
}

functionName(parameters)    // 函数调用
```

函数声明后不会立即执行，会在我们需要的时候调用到。

函数提升：

- 提升（Hoisting）是 JavaScript 默认将当前作用域提升到前面去的行为。
- 提升（Hoisting）应用在变量的声明与函数的声明。

因此，函数可以在声明之前调用：

```
console.log(add(3, 4));

function add(a, b){
    return a + b;
}
```

函数表达式：JavaScript 函数可以通过一个表达式定义。

```
const add = function(a, b){
    return a + b;
}

console.log(add(3, 4));
```

箭头函数：表现形式更加简洁。

```
const add = (a, b) => {
    return a + b;
}

console.log(add(3, 4));
```

函数作用域

局部变量：只能在函数内部访问。

变量在函数内声明，变量为局部变量，具有局部作用域。

```
const output = () => {
    let a = 10;
}

console.log(a);
```

变量在函数外定义，即为全局变量。

全局变量有 **全局作用域**： 网页中所有脚本和函数均可使用。

```
let url = "https://noi.hioier.com/";

const output = () => {
    console.log(url);
}

output();
```

哥德巴赫猜想

描述

伟大的哥德巴赫猜想是：任何一大于6的偶数总可以分解为两个素数之和。现在，请你编程验证哥德巴赫猜想，即输入一个大于6的偶数n，将其分解为两个素数之和输出。如果有多种分解答案，请输出字典序最小那一个。

输入

一行一个正整数n，6≤n≤1000。

输出

一行一个表达式，表示字典序最小的一种分解方法，具体格式参见样例。

输入样例 1

```
6
```

输出样例 1

```
6 = 3 + 3
```

输入样例 2

```
14
```

输出样例 2

```
14 = 3 + 11
```

首先，将一个大问题划分成两个子问题：

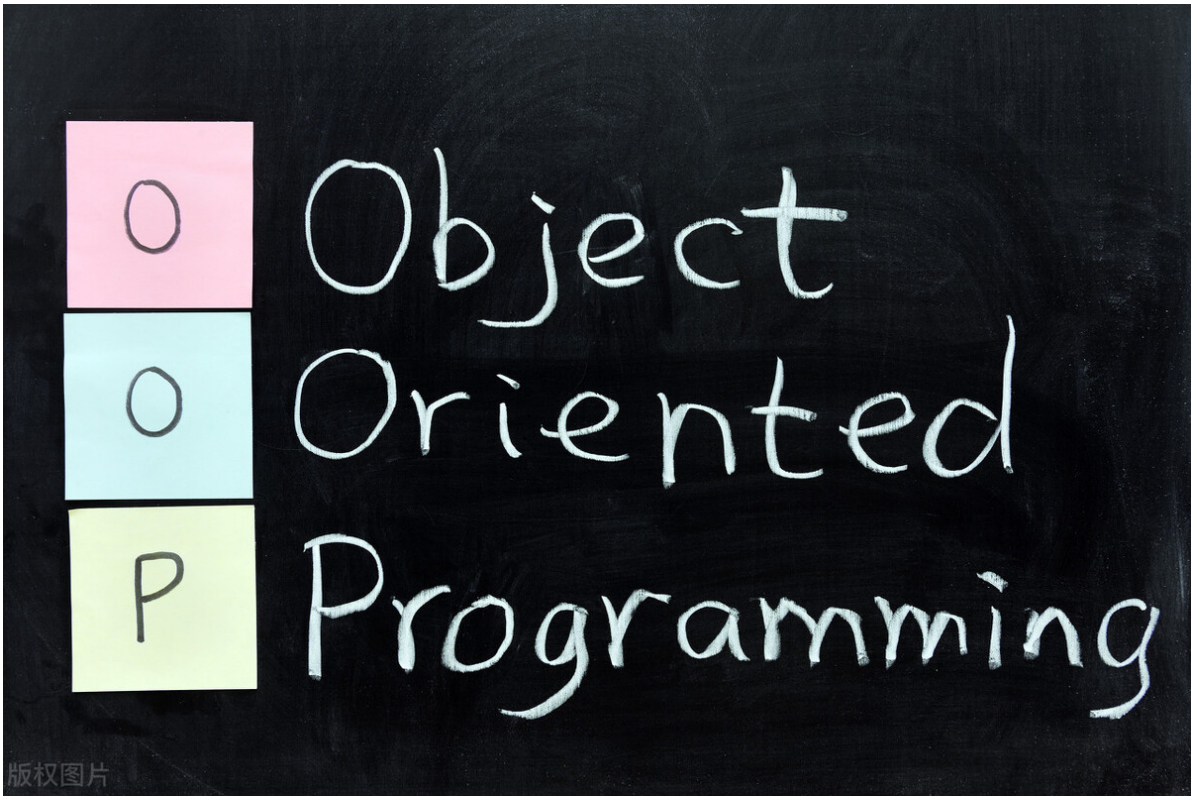
1. 判断一个数是否是质数；
2. 循环遍历2~n，如果i是质数且n-i也是质数，则输出结果，并跳出循环。

因为枚举过程是从小到大，第一个找到的可行解一定是字典序最小的。

```
let buf = "";

const is_prime = (n) => {
    for(let i = 2; i < n; i++){
        if(n % i == 0)
            return false;
    }

    return true;
}

process.stdin.on("readable", function(){
    let chunk = process.stdin.read();
    if(chunk) buf += chunk.toString();
});

process.stdin.on("end", function(){
   let n = parseInt(buf);

   for(let i = 2; i <= n; i++){
        if(is_prime(i) && is_prime(n - i)){
            console.log(`${n} = ${i} + ${n-i}`);
            break;
        }
   }

   // console.log(is_prime(n))

});
```

面向对象编程

面向对象编程相较于面向过程编程更适合大型程序设计。



类是用于创建对象的模板。我们使用 class 关键字来创建一个类，类体在一对大括号 **{}** 中，我们可以在大括号 {} 中定义类成员的位置，如方法或构造函数。

每个类中包含了一个特殊的方法 constructor()，它是类的构造函数，在创建对象时自动执行。

```
class People{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }

    output(){
        console.log(`My name is ${this.name}, I am ${this.age} years old.`);
    }
}

let xiaoming = new People("小明", 10);
xiaoming.output();
```

继承：

在子类的构造函数中，只有调用super之后，才可以使用this关键字。

成员重名时，子类的成员会覆盖父类的成员。

```
class Student extends People{
    constructor(name, age, score){
        super(name, age);
        this.score = score;
    }

    output(){
        console.log(`My name is ${this.name}, I am ${this.age} years old.My total score is ${this.score}.`);
    }
}

let xiaohong = new Student("小红", 8, 300);
xiaohong.output();
```

静态方法和静态变量

静态方法：在成员函数前添加static关键字即可。静态方法不会被类的实例继承，只能通过类来调用。

```
class People{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }

    output(){
        console.log(`My name is ${this.name}, I am ${this.age} years old.`);
    }

    static current_class_name(){
        console.log("People");
    }
}

let xiaoming = new People("小明", 10);
// xiaoming.output();
// xiaoming.current_class_name();
People.current_class_name();
```

静态变量：只能通过classname.variablename定义和访问。

```
class People{

    constructor(name, age){
        this.name = name;
        this.age = age;
        People.color = 'yellow';
    }

    output(){
        console.log(`My name is ${this.name}, I am ${this.age} years old.`);
    }

    static current_class_name(){
```

```
        console.log("People");
    }
}

console.log(People.color);
```