Factors:

Factors brute force:

function factorsOfNumber(number) {

let factors = [];

for (let i = 1; i <= number; i++) {

if (number % i === 0) {

factors.push(i);

}

}

return factors;

}

Find upto square root and push with its pair too

function factorsOfNumber(number) {

let factors = [];

for (let i = 1; i \* i <= number; i++) {

if (number % i === 0) {

factors.push(i);

if (i !== number / i) {

factors.push(number / i);

}

}

}

return factors;

}

// Test case 1: Prime number

console.log("Factors of 13 =>", factorsOfNumber(13)); // [1, 13]

// Test case 2: Even number

console.log("Factors of 20 =>", factorsOfNumber(20)); // [1, 20, 2, 10, 4, 5]

// Test case 3: Perfect square

console.log("Factors of 25 =>", factorsOfNumber(25)); // [1, 25, 5]

// Test case 4: Large number

console.log("Factors of 1001 =>", factorsOfNumber(1001)); // [1, 1001, 7, 143, 11, 91, 13, 77, 77, 13, 91, 11, 143, 7, 1001, 1]

// Test case 5: Zero

console.log("Factors of 0 =>", factorsOfNumber(0)); // []

// Test case 6: Negative number

console.log("Factors of -12 =>", factorsOfNumber(-12)); // []

// Test case 7: Large prime number

console.log("Factors of 9973 =>", factorsOfNumber(9973)); // [1, 9973]

// Additional Test case 8: Large non-prime number

console.log("Factors of 12345 =>", factorsOfNumber(12345)); // [1, 3, 5, 15, 823, 2469, 4115, 12345]

// Additional Test case 9: Another prime number

console.log("Factors of 101 =>", factorsOfNumber(101)); // [1, 101]

// Additional Test case 10: Large perfect square

console.log("Factors of 14400 =>", factorsOfNumber(14400)); // [1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 30, 36, 40, 45, 48, 50, 60, 72, 75, 80, 90, 100, 120, 144, 150, 180, 200, 225, 240, 300, 360, 400, 450, 600, 720, 900, 1200, 1800, 3600]

Prime factors:

Trial Division (Basic Approach):

Time Complexity: O(sqrt(n))

Space Complexity: O(1)

function primeFactorsOfNumber(n) {

const factors = [];

for (let i = 2; i <= Math.sqrt(n); i++) {

while (n % i === 0) {

factors.push(i);

n /= i;

}

}

if (n > 1) {

factors.push(n);

}

return factors;

}

Sieve of Eratosthenes + Trial Division:

function primeFactorsOfNumber(n) {

// Sieve of Eratosthenes to generate prime numbers

function sieveOfEratosthenes(limit) {

const sieve = Array(limit + 1).fill(true);

sieve[0] = sieve[1] = false;

for (let p = 2; p \* p <= limit; p++) {

if (sieve[p]) {

for (let i = p \* p; i <= limit; i += p) {

sieve[i] = false;

}

}

}

const primes = [];

for (let i = 2; i <= limit; i++) {

if (sieve[i]) {

primes.push(i);

}

}

return primes;

}

// Generate primes up to the square root of n

const primes = sieveOfEratosthenes(Math.floor(Math.sqrt(n)));

const factors = [];

// Trial division with generated primes

for (const prime of primes) {

while (n % prime === 0) {

factors.push(prime);

n /= prime;

}

}

// If n is still greater than 1, it's a prime factor itself

if (n > 1) {

factors.push(n);

}

return factors;

}

Optimized Trial Division:

Time Complexity: O(sqrt(n) / log(n))

Space Complexity: O(1)

function primeFactorsOfNumber(n) {

const factors = [];

// 2 is only even, so reduce even to odd number

while (n % 2 === 0) {

factors.push(2);

n /= 2;

}

// iterate through all odd numbers

for (let i = 3; i \* i <= n; i += 2) {

while (n % i === 0) {

factors.push(i);

n /= i;

}

}

if (n > 1) {

factors.push(n);

}

return factors;

}

Advanced Algorithms (e.g., Pollard's Rho):

Time Complexity: Can vary but typically better than trial division for large numbers

Space Complexity: O(1)

For advanced algorithms like Pollard's Rho, you would typically use a specialized library or implementation, as they are complex to implement from scratch.

Other approaches:

Iterate through only prime values…

const checkPrime = (n) => {

if (n <= 1) return false;

else if (n <= 3) return true;

else if (n % 2 === 0 || n % 3 === 0) return false;

let i = 5;

while (i \* i <= n) {

if (n % i === 0 || n % (i + 2) === 0) return false;

i += 6;

}

return true;

};

function primeFactorsOfNumber(n) {

const factors = [];

while (n % 2 === 0) {

factors.push(2);

n /= 2;

}

for (let i = 3; i <= Math.sqrt(n); i += 2) {

while (n % i === 0) {

if (checkPrime(i)) {

factors.push(i);

}

n /= i;

}

}

if (n > 1 && checkPrime(n)) {

factors.push(n);

}

return factors;

}

Divisors finding and do things ( not performant )

const checkAnyDivisors = (number, divs) => {

for (let i = 0, len = divs.length; i <= len; i++) {

if (divs[i] \* divs[i] <= number && number % divs[i] === 0) return true;

}

return false;

};

const getPrimeNumbersUpToLimit = (n) => {

let primes = [];

for (let i = 2; i <= n; i++) {

if (!checkAnyDivisors(i, primes)) {

primes.push(i);

}

}

return primes;

};

function primeFactorsUsingCustomPrimes(n) {

const primes = getPrimeNumbersUpToLimit(Math.floor(Math.sqrt(n)));

const factors = [];

for (const prime of primes) {

while (n % prime === 0) {

factors.push(prime);

n /= prime;

}

}

if (n > 1) {

factors.push(n);

}

return factors;

}

Test cases:

// Prime number

console.log("Factors of 9973 =>", primeFactorsOfNumber(9973)); // [9973]

// Composite numbers

console.log("Factors of 24 =>", primeFactorsOfNumber(24)); // [2, 2, 2, 3]

console.log("Factors of 100 =>", primeFactorsOfNumber(100)); // [2, 2, 5, 5]

// Edge cases

console.log("Factors of 1 =>", primeFactorsOfNumber(1)); // []

console.log("Factors of 2 =>", primeFactorsOfNumber(2)); // [2] (Smallest prime)

console.log("Factors of 2 =>", primeFactorsOfNumber(17)); // [17] ( prime)

console.log("Factors of 0 =>", primeFactorsOfNumber(0)); // [] (Invalid input)

Fibonacci:

/\*\*

Fibonacci:

1,1,2,3,5,8

\*/

// basic recursive solution (t => O(2\*\*n), s => O(1))

const fib = (n) => {

if(n<3) return 1;

else return fib(n-1)+fib(n-2)

}

// dynamic programming memoization solution (t => O(n), s => O(n))

const fib1 = (n, cache = {1:1, 2:1}) => {

if(cache[n]) return cache[n];

else {

cache[n] = fib1(n-1, cache)+fib1(n-2, cache)

return cache[n]

}

}

// dynamic programming tablulation solution (t => O(n), s => O(n))

function fib2(n){

var fibo = [0, 1];

if (n <= 2) return 1;

for (var i = 2; i <=n; i++ ){

fibo[i] = fibo[i-1]+fibo[i-2];

console.log(i, fibo)

}

return fibo[n];

}

// Efficient solution since we need only last 2 fibs... (t => O(n), s => O(1))

function fib3(n){

if (n <= 2) return 1;

let [num1, num2] = [1,1];

for (let i = 3; i <=n; i++ ){

const newVal = num2+num1;

num1 = num2 ;

num2 = newVal;

}

return num2;

}

// console.log(fib(40))//=> 102334155

// console.log(fib1(40))//=> 102334155

// console.log(fib2(40))

// console.log(fib3(40))