

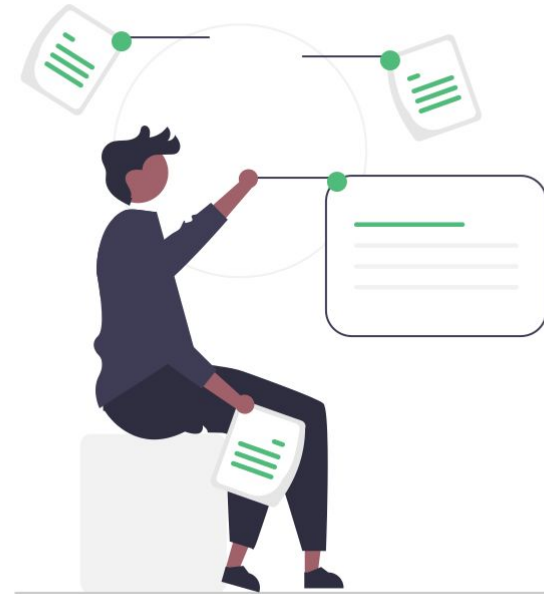
Full Stack Web Development

# Algorithm

- Problem solving
- Pseudocode
- Big O Notation
- Space and time complexity
- Search Algorithm
- Sort Algorithm

# How to Use Algorithms to Solve Problems?

An algorithm is a process or set of rules which must be followed to complete a particular task. This is basically the step-by-step procedure to complete any task. All the tasks are followed a particular algorithm, from making a cup of tea to make high scalable software. This is the way to divide a task into several parts.



# Understand

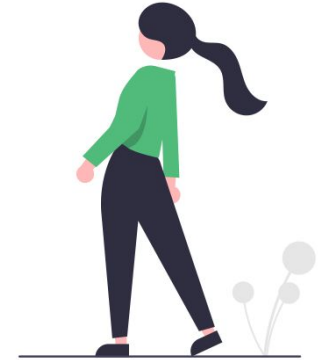
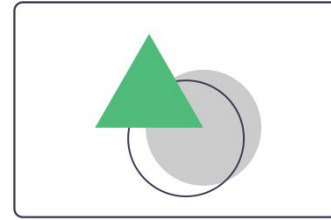
The first step to solve a problem, is to **understand the problem**. We won't be able to formulate a proper solution if we don't even know what the problem asks for.



# Understand

The most essential things you need to understand in a problem, **are its inputs and expected outputs.**

After that, understand what **conditions and/or parameters are given.**



# Understand

For example, a problem asks you to create a program which checks if a word is a palindrome. It asks for you to return a boolean, where *true* means the word is a palindrome, and *false* means the word is not a palindrome.

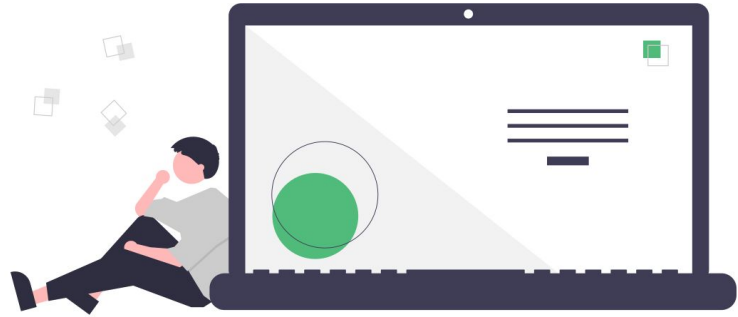
We now know the input should be a word (string), and the output should be a boolean. Now, let's analyze a few more conditions and parameters.



# Understand

We know that the input should be a string datatype because the problem asks us for a word.

But we also need to ask, will there be special characters in the string? If so, how should we handle them? Should we ignore it as if it doesn't exist? Or should we still treat it as a part of the word?



# Strategize

Now that we've gone through the process of understanding the problem, it's time to plan a solution.

This takes us to the next step, strategize.

In this step, we need to try to create a general overview on how to solve the presented problem.

A good way to start is by breaking it into smaller problems.





Based on the palindrome case, let's assume that the input will contain special characters that we need to ignore.

That means the first step to solving our problem is to validate the input, and to eliminate all special characters in the input.



Then, we need to be able to reverse the input.

And last, we compare the original input (already without special characters) against the reversed input.

If it matches, that means the word is a palindrome and we should return *true*. If it doesn't match, then it's not a palindrome and we should return *false*.



# Implement

After gathering all understanding and different strategies to solve the problem, the last step is to, of course, implement them.

However, sometimes it can be difficult to translate your logic into code, especially for beginners.

So here is another way for you to easily solve this problem. Psuedocode.



# What is pseudocode?

Pseudocode is basically an “easier-to-understand” version of programming languages, usually in the form of simple natural language.

Through pseudocode, you can express detailed step-by-step process in a much simpler language.

After that, you can easily implement it in your preferred programming language. Let's take a look at an example



# Pseudocode implementation

Still using the palindrome problem as an example. Let's generate the pseudocode needed to solve the problem.

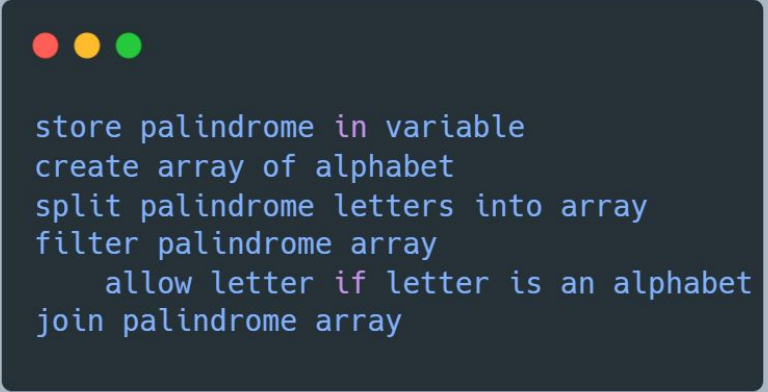
Remember that the first step is to eliminate all special characters in the palindrome. To achieve this, we can probably write this pseudocode.

```
store palindrome in variable  
create array of alphabet  
split palindrome letters into array  
filter palindrome array  
    allow letter if letter is an alphabet  
join palindrome array
```

# Pseudocode implementation

We know that this doesn't seem like it can be executed by a computer.


However, it does give us a pretty detailed picture on what we need to do when we code.

A dark-themed terminal window with three colored window control buttons (red, yellow, green) at the top left. It contains a list of pseudocode steps for palindrome processing.

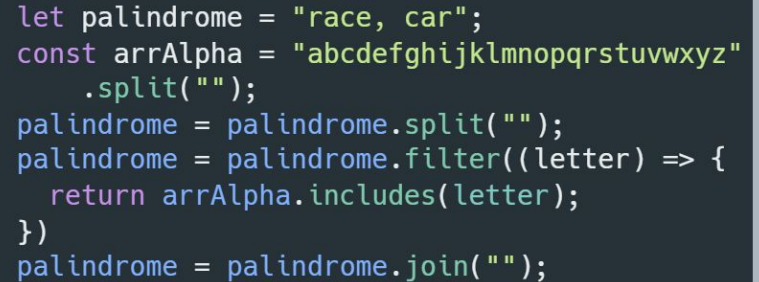
```
store palindrome in variable  
create array of alphabet  
split palindrome letters into array  
filter palindrome array  
    allow letter if letter is an alphabet  
join palindrome array
```

# Pseudocode implementation

Now, using our pseudocode, we can implement our plans and produce actual code using Javascript. Here's a side-by-side comparison of our pseudocode and actual code.



```
store palindrome in variable  
create array of alphabet  
split palindrome letters into array  
filter palindrome array  
    allow letter if letter is an alphabet  
join palindrome array
```



```
let palindrome = "race, car";  
const arrAlpha = "abcdefghijklmnopqrstuvwxyz"  
    .split("");  
palindrome = palindrome.split("");  
palindrome = palindrome.filter((letter) => {  
    return arrAlpha.includes(letter);  
})  
palindrome = palindrome.join("");
```

# Challenge!

Alright! It's time for a bit of practice. So far we've been able to produce the pseudocode and actual code to filter out special characters from the palindrome input.

Now, give yourself a challenge and try to complete the rest of the solution by yourself! Don't forget to write your pseudocode to help guide you when coding your solution.





# What is Big O Notation?

---

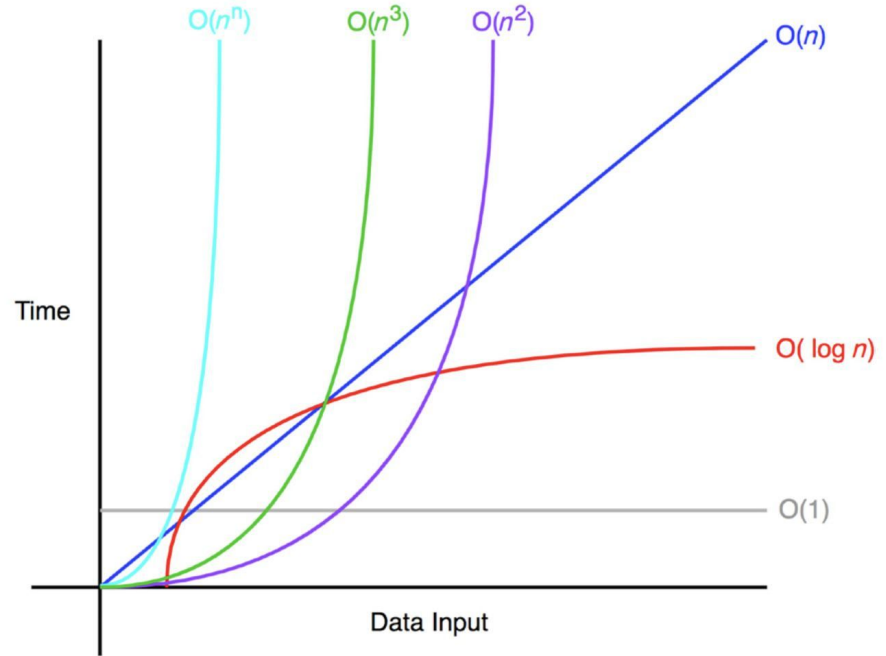
Big O Notation is a way to measure an algorithm's efficiency. It measures the time it takes to run your function as the input grows. Or in other words, how well does the function scale.

There are two parts to measuring efficiency:

1. **Time complexity** is a measure of how long the function takes to run in terms of its computational steps.
2. **Space complexity** has to do with the amount of memory used by the function.

# Measure Big O Notation

- $O(1)$ , constant
- $O(n)$ , linear
- $O(\log n)$ , logarithm  $N$
- $O(n^2)$ , square 2
- $O(n^N)$ ,  $N$  square  $N$



# Time Complexity – Analogy

Let's say we have a box package that need to send from BSD to Jakarta. The mileage between BSD and Jakarta is about one hour. The delivery courier could only bring one box each trip.

What would happen if there are three box of packages?

In this situation, time complexity would be count as  $O(n)$  or called as order linear

## Space Complexity – Analogy

---

Depends on the last situation, let say we use trucks as the transportation which has larger space than motorcycle. The mileage still takes one hour but it could bring three box packages in a single trip.

Have you see the difference?

With trucks it is only need one hour to send 3 box packages, but in return we need more spaces on transportation delivery

## Example Case

- 
- Check if an input (array) contains duplicate
  - Input array are not sorted
  - Return true if there are duplicate data
  - Return false if there are no duplicate data

# Brute Force

- Not efficient but effective
- Easy to implement
- Time complexity  $O(N^2)$



```
function checkDuplicate(arr) {  
  for(let i = 0; i < arr.length; i++) {  
    for(let j = i+1; j < arr.length; j++) {  
      if(arr[i] === arr[j]) return true;  
    }  
  }  
  return false  
}  
  
console.log(checkDuplicate([1,2,3,1]));
```

# Optimize with Extra Memory


- Extra space make runtime faster
- Using hash table
- Time Complexity  $O(N)$
- Space Complexity  $O(N)$



```
function checkDuplicate(arr) {  
  const uniqueData = new Set();  
  for(let i = 0; i < arr.length; i++) {  
    if(uniqueData.has(arr[i])) return true  
    else uniqueData.add(arr[i])  
  }  
  return false  
}  
  
console.log(checkDuplicate([1,2,3,1]));
```

# Optimize without Extra Memory

- Uses sorting
- Space Complexity  $O(N \log N)$



```
function checkDuplicate(arr) {  
  arr.sort();  
  for(let i = 0; i < arr.length - 1; i++) {  
    if(arr[i] === arr[i+1]) return true  
  }  
  return false  
}  
  
console.log(checkDuplicate([5,1,3,1]));
```



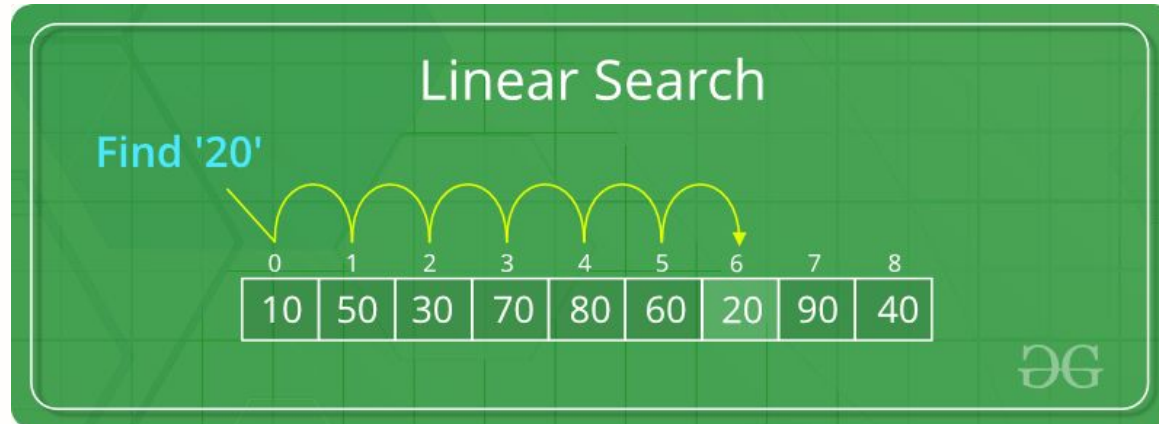
- Linear Search
- Binary Search
- Jump Search
- Interpolation Search
- Exponential Search
- Sublist Search (Search a linked list in another list)
- Fibonacci Search
- Etc

Find out more on: <https://www.geeksforgeeks.org/searching-algorithms/>

# Linear Search vs Binary Search

**Linear Search** – A simple approach is to do a linear search:

- Start from the leftmost element of `arr[]` and one by one compare `x` with each element of `arr[]`
- If `x` matches with an element, return the index.
- If `x` doesn't match with any of the elements, return -1.




# Linear Search vs Binary Search

## Linear Search

Time complexity:  $O(n)$

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching compared to Linear search.



```
function search(arr, x)
{
  let i;
  for (i = 0; i < arr.length; i++) {
    if (arr[i] == x) {
      return i;
    }
  }
  return -1;
}

search([2,20,10,3], 20);
//return 20
```

# Linear Search vs Binary Search

**Binary Search** – is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to  $O(\log n)$ .

The basic steps to perform Binary Search are:

- Begin with the mid element of the whole array as a search key.
- If the value of the search key is equal to the item then return an index of the search key.
- Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise, narrow it to the upper half.
- Repeatedly check from the second point until the value is found or the interval is empty.

# Linear Search vs Binary Search

## Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 <sup>nd</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 < 56 take 1 <sup>st</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91



# Linear Search vs Binary Search



```
function binarySearch(arr, l, r, x){  
  if (r >= l) {  
    let mid = l + Math.floor((r - l) / 2);  
    // If the element is present at the middle itself  
    if (arr[mid] == x) return mid;  
    // If element is smaller than mid, then  
    // it can only be present in left subarray  
    if (arr[mid] > x)  
      return binarySearch(arr, l, mid - 1, x);  
    // Else the element can only be present in right subarray  
    return binarySearch(arr, mid + 1, r, x);  
  }  
  return -1;  
}  
let arr = [ 2, 3, 4, 10, 40 ];  
let x = 10;  
console.log(binarySearch(arr, 0, arr.length-1, x))
```

---

Sorting algorithms are categorized into:

- **Internal sorting algorithms** – These are sorting algorithms applied to a small amount of data. Only the main memory is used. Examples would be bubble sort, insertion sort, and quicksort.
- **External sorting algorithms** – These are sorting algorithms that can be applied to massive amounts of data. As a result, external storage devices such as hard drives, and flash disks are used. An example would be merge sort.

---

Some sorting algorithms are more efficient than others. The effectiveness of a sorting algorithm is usually defined by the following performance measures:

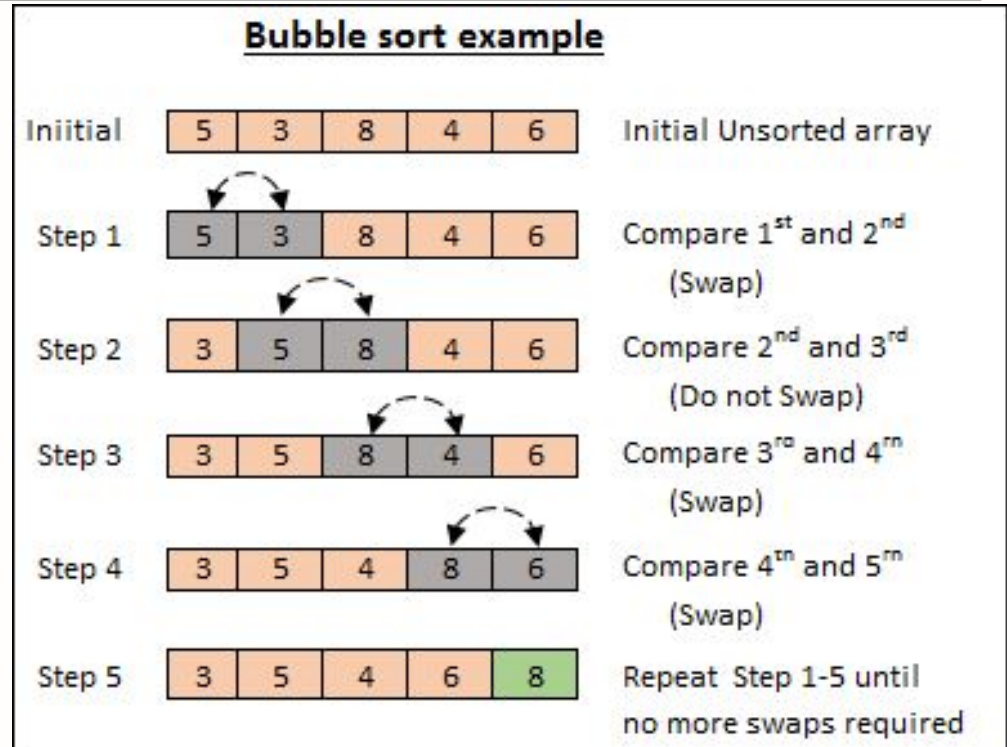
- **Time complexity:** This is the amount of time required by the computer to perform the sorting based on an algorithm.
- **Space complexity:** It is the amount of computer memory required by the computer to perform the sorting based on an algorithm.



# Bubble Sort

Step-by-step guide:

- Start by comparing the first two elements in an array.
- Swap them if required.
- Continue till the end of the array. At this point, you have made a series of inner passes and completed an outer pass.
- Repeat the process until the entire array is sorted.



# Bubble Sort

Bubble sort has the following performance cases:

- Worst-case time complexity: Big  $O(n^2)$ .
- Space complexity: Big  $O(1)$ .



```
function bubbleSort(arr){  
  for(let i = 0; i < arr.length; i++){  
    for(let j = 0; j < arr.length - i - 1; j++){  
      if(arr[j + 1] < arr[j]){  
        [arr[j + 1],arr[j]] = [arr[j],arr[j + 1]]  
      }  
    }  
  }  
};  
return arr;  
};  
console.log(bubbleSort([5,3,8,4,6]));
```

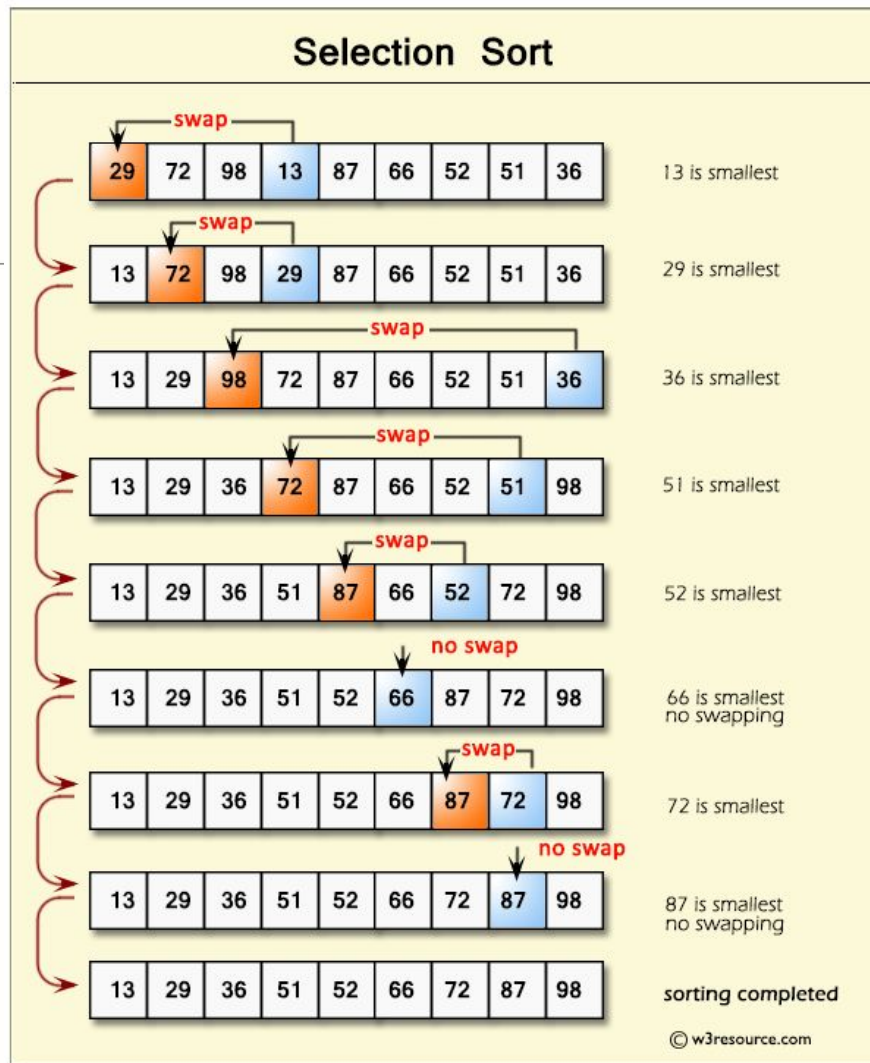
---

Selection sort uses the recursion technique. In the guide below, we are using ascending order. For descending order, you do the reverse.

Step-by-step guide:

- Given an array, assume that the first element in the array is the smallest.
- From the other portion of the array, find the minimum value, and swap it with the first element. At this point, you have completed the first pass.
- Repeat the same procedure with the rest of the array comparing the elements to the right, not the left.

# Selection Sort



# Selection Sort

Selection sort has the following performance cases:

- Worst-case time complexity: Big O ( $n^2$ ).
- Space complexity: Big O(1).

```
function selectionSort(arr) {  
  let min;  
  for (let i = 0; i < arr.length; i++) {  
    //index of the smallest element to be the ith element.  
    min = i;  
    //Check through the rest of the array for a lesser element  
    for (let j = i + 1; j < arr.length; j++) {  
      if (arr[j] < arr[min]) min = j;  
    }  
    //compare the indexes and swap  
    if (min !== i) [arr[i], arr[min]] = [arr[min], arr[i]];  
  }  
  return arr;  
}  
console.log(selectionSort([29, 72, 98, 13, 87, 66, 52, 51, 36]));
```

# Thank You!

