**Insertion sort**

Insertion sort is a simple and intuitive comparison-based sorting algorithm. It works similarly to how you might sort playing cards in your hands. The algorithm builds the final sorted array one item at a time, with the benefit of being easy to implement and efficient for small data sets or nearly sorted data.

Here’s how insertion sort works, step-by-step:

1. \*\*Start with the second element\*\* (the first element is considered sorted).

2. \*\*Compare the current element\*\* (key) with the elements in the sorted part of the array.

3. \*\*Shift elements\*\* in the sorted part to the right until the correct position for the current element is found.

4. \*\*Insert the current element\*\* into the correct position.

5. Repeat the process for all elements until the array is sorted.

Let’s walk through an example with the array `[5, 2, 9, 1, 5, 6]`.

### Step-by-Step Example

\*\*Initial Array:\*\* `[5, 2, 9, 1, 5, 6]`

1. \*\*First iteration\*\* (i = 1):

- Key = 2

- Compare with 5, since 2 < 5, shift 5 to the right.

- Insert 2 at the beginning.

- \*\*Array now:\*\* `[2, 5, 9, 1, 5, 6]`

2. \*\*Second iteration\*\* (i = 2):

- Key = 9

- Compare with 5, 9 is greater, no shifts needed.

- \*\*Array now:\*\* `[2, 5, 9, 1, 5, 6]`

3. \*\*Third iteration\*\* (i = 3):

- Key = 1

- Compare with 9, shift 9 to the right.

- Compare with 5, shift 5 to the right.

- Compare with 2, shift 2 to the right.

- Insert 1 at the beginning.

- \*\*Array now:\*\* `[1, 2, 5, 9, 5, 6]`

4. \*\*Fourth iteration\*\* (i = 4):

- Key = 5

- Compare with 9, shift 9 to the right.

- Compare with 5, no shift needed (equal values can stay in place).

- Insert 5 in the correct position.

- \*\*Array now:\*\* `[1, 2, 5, 5, 9, 6]`

5. \*\*Fifth iteration\*\* (i = 5):

- Key = 6

- Compare with 9, shift 9 to the right.

- Compare with 5, 6 is greater, no more shifts needed.

- Insert 6 in the correct position.

- \*\*Array now:\*\* `[1, 2, 5, 5, 6, 9]`

### Summary

At each step, the insertion sort algorithm takes one element from the unsorted part of the array and inserts it into its correct position within the sorted part of the array, shifting elements as necessary. By the end of the process, the entire array is sorted.

This method is efficient for small datasets and nearly sorted arrays but less so for large, unsorted arrays due to its O(n^2) time complexity in the average and worst cases.

**Quick sort**   
  
Quick sort is a highly efficient sorting algorithm and is based on the divide-and-conquer strategy. Here's a step-by-step explanation of how quick sort works, followed by an example:

### How Quick Sort Works

1. \*\*Choose a Pivot\*\*: Select an element from the array as the pivot. Various strategies can be used to select the pivot, such as picking the first element, the last element, the middle element, or a random element.

2. \*\*Partitioning\*\*: Rearrange the array so that all elements less than the pivot are to the left of the pivot, and all elements greater than the pivot are to the right. The pivot is now in its correct position in the sorted array.

3. \*\*Recursively Apply\*\*: Recursively apply the above steps to the sub-arrays of elements with smaller values and greater values.

### Example

Let's sort the array `[10, 7, 8, 9, 1, 5]` using quick sort.

1. \*\*Initial Array\*\*: `[10, 7, 8, 9, 1, 5]`

2. \*\*Choose Pivot\*\*: Let's choose the last element as the pivot, so pivot = `5`.

3. \*\*Partitioning\*\*:

- Start with two pointers: one at the beginning and one at the end.

- Compare each element with the pivot and rearrange them.

- After partitioning, the array looks like this: `[1, 5, 8, 9, 7, 10]`

- The pivot `5` is now in its correct position.

4. \*\*Recursive Steps\*\*:

- Apply quick sort to the left sub-array `[1]` and the right sub-array `[8, 9, 7, 10]`.

5. \*\*Sorting the Right Sub-array\*\* `[8, 9, 7, 10]`:

- Choose the last element as the pivot, so pivot = `10`.

- After partitioning, the array looks like this: `[8, 9, 7, 10]`

- Pivot `10` is in its correct position.

- Apply quick sort to `[8, 9, 7]`.

6. \*\*Sorting the Sub-array\*\* `[8, 9, 7]`:

- Choose the last element as the pivot, so pivot = `7`.

- After partitioning, the array looks like this: `[7, 8, 9]`

- Pivot `7` is in its correct position.

- Apply quick sort to `[8, 9]`.

7. \*\*Sorting the Sub-array\*\* `[8, 9]`:

- Choose the last element as the pivot, so pivot = `9`.

- After partitioning, the array looks like this: `[8, 9]`

- Pivot `9` is in its correct position.

- Apply quick sort to `[8]`, which is already sorted.

### Summary

By the end of these steps, the entire array is sorted. The final sorted array is `[1, 5, 7, 8, 9, 10]`.

Quick sort efficiently sorts arrays by using a pivot to divide the array into sub-arrays and recursively sorting these sub-arrays. It has an average-case time complexity of O(n log n), making it faster than simpler algorithms like bubble sort and insertion sort for large datasets. However, in the worst case (when the smallest or largest element is always chosen as the pivot), its time complexity can degrade to O(n^2). This worst-case performance can often be mitigated by using more sophisticated pivot selection techniques, such as choosing a random pivot or the median of three elements.

**Merge sort**   
  
Merge sort is a divide-and-conquer algorithm that efficiently sorts an array by dividing it into smaller sub-arrays, sorting those sub-arrays, and then merging them back together. It has a consistent time complexity of O(n log n) in the worst, average, and best cases, making it a reliable choice for sorting large datasets.

### How Merge Sort Works

1. \*\*Divide\*\*: Split the array into two halves.

2. \*\*Conquer\*\*: Recursively sort each half.

3. \*\*Merge\*\*: Merge the two sorted halves back together to form the sorted array.

### Example

Let's sort the array `[38, 27, 43, 3, 9, 82, 10]` using merge sort.

1. \*\*Initial Array\*\*: `[38, 27, 43, 3, 9, 82, 10]`

2. \*\*Divide\*\*: Split the array into two halves:

- Left: `[38, 27, 43]`

- Right: `[3, 9, 82, 10]`

3. \*\*Recursive Sorting\*\*:

- Sort the left half `[38, 27, 43]`:

- Divide: `[38]` and `[27, 43]`

- Sort `[27, 43]`:

- Divide: `[27]` and `[43]`

- Merge: `[27, 43]` (already sorted)

- Merge `[38]` and `[27, 43]`:

- Compare and merge: `[27, 38, 43]`

- Sort the right half `[3, 9, 82, 10]`:

- Divide: `[3, 9]` and `[82, 10]`

- Sort `[3, 9]`:

- Divide: `[3]` and `[9]`

- Merge: `[3, 9]` (already sorted)

- Sort `[82, 10]`:

- Divide: `[82]` and `[10]`

- Merge: `[10, 82]` (sorted)

- Merge `[3, 9]` and `[10, 82]`:

- Compare and merge: `[3, 9, 10, 82]`

4. \*\*Final Merge\*\*:

- Merge the sorted halves `[27, 38, 43]` and `[3, 9, 10, 82]`:

- Compare and merge:

- `[3]` is less than `[27]`, so take `[3]`

- `[9]` is less than `[27]`, so take `[9]`

- `[10]` is less than `[27]`, so take `[10]`

- `[27]` is less than `[82]`, so take `[27]`

- `[38]` is less than `[82]`, so take `[38]`

- `[43]` is less than `[82]`, so take `[43]`

- Finally take `[82]`

- Resulting in the sorted array: `[3, 9, 10, 27, 38, 43, 82]`

### Detailed Steps:

1. \*\*Initial Split\*\*:

```

[38, 27, 43, 3, 9, 82, 10]

/ \

[38, 27, 43] [3, 9, 82, 10]

```

2. \*\*Further Splitting\*\*:

```

[38, 27, 43] [3, 9, 82, 10]

/ \ / \

[38] [27, 43] [3, 9] [82, 10]

/ \ / \ / \

[27] [43] [3] [9] [82] [10]

```

3. \*\*Merging\*\*:

```

[38] [27, 43] -> [27, 43]

[27] [43] -> [27, 43]

[3] [9] -> [3, 9]

[82] [10] -> [10, 82]

[38] [27, 43] -> [27, 38, 43]

[3, 9] [10, 82] -> [3, 9, 10, 82]

[27, 38, 43] [3, 9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82]

```

### Summary

By recursively dividing and merging, merge sort efficiently sorts the array. It is particularly useful for large datasets due to its O(n log n) time complexity, but it requires additional space proportional to the array size for the temporary sub-arrays used during merging.

**Heap sort**   
  
Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure. It sorts an array by first building a max-heap from the input data, then repeatedly extracting the maximum element from the heap and placing it at the end of the array. This process continues until all elements are sorted.

### How Heap Sort Works

1. \*\*Build a Max-Heap\*\*: Convert the array into a max-heap. A max-heap is a complete binary tree where the value of each node is greater than or equal to the values of its children.

2. \*\*Extract Elements\*\*: Repeatedly extract the maximum element (the root of the heap) and move it to the end of the array. Then, rebuild the heap with the remaining elements.

3. \*\*Heapify\*\*: Ensure the heap property is maintained after each extraction.

### Example

Let's sort the array `[4, 10, 3, 5, 1]` using heap sort.

\*\*Initial Array\*\*: `[4, 10, 3, 5, 1]`

1. \*\*Build Max-Heap\*\*:

- Start from the first non-leaf node and heapify each node up to the root.

- The first non-leaf node is at index `1` (value `10`).

```

4

/ \

10 3

/ \

5 1

```

- Heapify node at index `1`:

- Left child (index `3`, value `5`), right child (index `4`, value `1`).

- 10 is already greater than both children, so no change needed.

- Move to the root (index `0`):

- Left child (index `1`, value `10`), right child (index `2`, value `3`).

- Swap `4` with `10` since `10` is larger.

```

10

/ \

4 3

/ \

5 1

```

- Heapify node at index `0` again (now value `4`):

- Swap `4` with `5` since `5` is larger.

```

10

/ \

5 3

/ \

4 1

```

- Max-heap is now built: `[10, 5, 3, 4, 1]`

2. \*\*Extract Elements\*\*:

- Swap the root of the heap with the last element of the heap:

```

[1, 5, 3, 4, 10]

```

- Reduce the heap size by one and heapify the root:

- Heapify node at index `0`:

- Left child (index `1`, value `5`), right child (index `2`, value `3`).

- Swap `1` with `5` since `5` is larger.

```

5

/ \

1 3

/ \

4 10

```

- Heapify node at index `1`:

- Swap `1` with `4` since `4` is larger.

```

5

/ \

4 3

/ \

1 10

```

- Swap the root with the second last element of the heap:

```

[1, 4, 3, 5, 10]

```

- Reduce the heap size by one and heapify the root:

- Heapify node at index `0`:

- Swap `1` with `4` since `4` is larger.

```

4

/ \

1 3

/ \

5 10

```

- Swap the root with the third last element of the heap:

```

[3, 1, 4, 5, 10]

```

- Reduce the heap size by one and heapify the root:

- Heapify node at index `0`:

- Swap `3` with `1` since `3` is larger.

```

3

/ \

1 4

/ \

5 10

```

- Finally, swap the root with the fourth last element of the heap:

```

[1, 3, 4, 5, 10]

```

- The heap is now reduced to one element, and the array is sorted.

### Final Sorted Array

The sorted array is `[1, 3, 4, 5, 10]`.

### Summary

Heap sort involves building a max-heap from the array, then repeatedly extracting the maximum element and rebuilding the heap. It has a time complexity of O(n log n) for all cases, making it efficient for large datasets. However, it requires additional space for the heap structure, but this is generally minimal as the heap can be constructed in place within the array.

Certainly! Here's an example of how you can implement heap sort in JavaScript:

```javascript

function heapify(arr, n, i) {

let largest = i; // Initialize largest as root

let left = 2 \* i + 1; // left = 2\*i + 1

let right = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest !== i) {

let swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

function heapSort(arr) {

let n = arr.length;

// Build heap (rearrange array)

for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// One by one extract an element from heap

for (let i = n - 1; i > 0; i--) {

// Move current root to end

let temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Example usage

let arr = [12, 11, 13, 5, 6, 7];

console.log("Original array:", arr);

heapSort(arr);

console.log("Sorted array:", arr);

```

### Explanation:

1. \*\*Heapify Function\*\*:

- The `heapify` function ensures that the subtree rooted at index `i` in an array of size `n` satisfies the heap property.

- It finds the largest element among the root, left child, and right child. If the root is not the largest, it swaps it with the largest and continues heapifying the affected subtree.

2. \*\*Heap Sort Function\*\*:

- The `heapSort` function first builds a max-heap from the input array. It starts from the last non-leaf node and calls `heapify` to ensure the heap property for each node.

- After building the heap, it repeatedly swaps the root of the heap (the largest element) with the last element of the heap, reduces the size of the heap by one, and calls `heapify` on the root to restore the heap property.

3. \*\*Example Usage\*\*:

- The example usage shows an initial array, applies `heapSort` to it, and then prints the sorted array.

You can run this code in any JavaScript environment, such as a browser console or Node.js, to see how heap sort works.