Divide & Conquer

▼ **TOC**

▼ Insertion Sort

Insertion Sort is a stable, in-place sorting algorithm that's suitable for small or partially sorted arrays. It belongs to the family of "Insertion Sort" algorithms, which inserts one element at a time into the sorted part of the array. The algorithm maintains a sorted subarray and selects the next element from the unsorted part, inserting it into the correct position in the sorted subarray. However, it can be slow for large input sizes.

Advantages

- Simple to understand and implement.
- Efficient for small data sets or when the data is already partially sorted.
- In-place sorting algorithm with space complexity O(1).

Disadvantages

- Very inefficient for large data sets.
- Time complexity is O(n^2) in the worst-case scenario, which makes it impractical for large data sets.

Algorithm

1. Iterate through each element in the array, starting from the second element.
2. Compare the current element with the one before it.
3. If the current element is smaller, swap it with the previous element.
4. Repeat step 3 until the current element is in its correct position.

5. Continue iterating through the array until all elements are sorted.

Example

Let's say we have an array of numbers: `[4, 2, 7, 1, 3]`. Here is how the algorithm would sort this array:

1. Starting with the second element, compare 2 with 4. Since 2 is smaller, swap them: `[2, 4, 7, 1, 3]`.
2. Compare 7 with 4. Since 7 is larger, leave it in place: `[2, 4, 7, 1, 3]`.
3. Compare 1 with 7, then with 4, and then with 2. Since 1 is smaller than all of these, swap it with 2: `[1, 2, 4, 7, 3]`.
4. Compare 3 with 7, then with 4, and then with 2. Since 3 is smaller than 7 and 4, swap it with them respectively: `[1, 2, 3, 4, 7]`.
5. The array is now sorted.

Complexity

Time Complexity: O(n^2)

Space Complexity: O(1)

▼ Heap Sort

Heap Sort is a sorting algorithm that uses the heap data structure to select the maximum element from the unsorted part of the array in each iteration. It belongs to the "Selection Sort" family of algorithms that moves the selected element to the end of the sorted part of the array. Heap Sort is a comparison-based, in-place, and divide-and-conquer algorithm that efficiently selects the maximum element using the heap data structure.

What is a heap tree?

A heap tree, also known as a binary heap, is a complete binary tree in which each node is greater than or equal to (in a max heap) or less than or equal to (in a min heap) its children nodes.

This ordering property makes it an efficient data structure for implementing priority queues and sorting algorithms.

Types of heap tree:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Advantages

- It is an in-place sorting algorithm, which means it sorts the input in place without requiring any additional memory.
- It has a worst-case time complexity of O(n log n), which makes it efficient for large inputs.

Disadvantages

- It is unstable sorting algorithm, which means that it does not maintain the relative order of equal elements in the input.
- It has a relatively high constant factor compared to other sorting algorithms, which means that it may not be as efficient for small inputs.

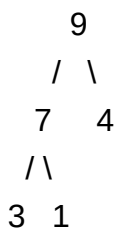
Algorithm

1. Convert string array to a heap tree (ACBT).
2. Apply ascending or descending heapify method on the heap tree to convert it into a min-heap or max-heap tree respectively.
3. Delete the root node of the tree and store it into an array.
4. Repeat step 2 and 3 until there is no element left in the heap tree.

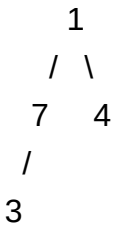
Example

Let's say we have an array of numbers: `[4, 1, 7, 3, 9]`. Here is how the algorithm would sort this array:

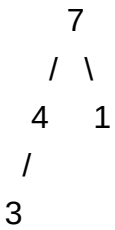
1. Build a heap by using heapify method:



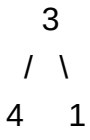
2. Extract the root node and store it in an array: `[9]`



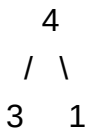
3. Restore the heap property of the tree:



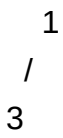
4. Extract the root node and store it in an array: `[9, 7]`



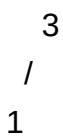
5. Restore the heap property of the tree:



6. Extract the root node and store it in an array: `[9, 7, 4]`



7. Restore the heap property of the tree:



8. Extract the root node and store it in an array: `[9, 7, 4, 3]`

1

9. Put the last element in the array as well: `[9, 7, 4, 3, 1]`
10. The array is now sorted.

Complexity

Time Complexity: O(n log n)

Space Complexity: O(1)

▼ Counting Sort

Counting Sort is a stable and out-of-place sorting algorithm that uses arithmetic operations to sort elements in linear time. It counts the number of occurrences of each distinct element and calculates their position in the sorted array, without relying on element comparisons.

Advantages

- Counting Sort has a time complexity of O(n+k), which is a linear time complexity and is faster than comparison-based sorting algorithms like Merge Sort and Quick Sort in certain situations.
- Counting Sort is a stable sorting algorithm, meaning it maintains the relative order of equal elements in the input array in the output array.
- Counting Sort is easy to implement and can be efficient for small input sizes or when the range of values in the input array is small.

Disadvantages

- Counting Sort is not suitable for large input sizes or when the range of values in the input array is significantly greater than the size of the input array. This is because Counting Sort uses additional memory to store the counts of each distinct element in the input array.
- Counting Sort is not a comparison-based sorting algorithm and can only be used for sorting elements with integer keys or values.
- Counting Sort may be inefficient for sparse input arrays where most of the elements have the same key or value. In such cases, the algorithm may still allocate memory for the full range of values, resulting in unnecessary memory usage.

Algorithm

1. Initialize a count array with zeros of length k+1, where k is the maximum value in the input array.
2. Count the occurrences of each element in the input array by incrementing the count array at the index corresponding to the value of the element.
3. Modify the count array to store the position of each element in the sorted array by adding the current count to the previous count.
4. Initialize a sorted array with zeros of length equal to the length of the input array.
5. Place each element in the sorted array in the correct position by using the count array. For each element in the input array, place it in the sorted array at the index given by the count array for that element value, and decrement the count for that element value.
6. Return the sorted array.

Example

Suppose we have an input array of integers `arr = [4, 1, 3, 4, 3, 1, 2, 4, 2, 2]` that we want to sort using Counting Sort.

1. Find the maximum value in the input array, which is 4. Initialize a count array of length 5 with zeros: `count = [0, 0, 0, 0, 0]`.
2. Count the occurrences of each element in the input array by incrementing the count array at the index corresponding to the value of the element. After counting, the count array becomes: `count = [0, 2, 3, 2, 3]`.
3. Modify the count array to store the position of each element in the sorted array. To do this, add the current count to the previous count, starting from index 1. After modification, the count array becomes: `count = [0, 2, 5, 7, 10]`.
4. Initialize a sorted array of length equal to the length of the input array with zeros: `sorted_arr = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`.
5. Place each element in the sorted array in the correct position by using the count array. For each element in the input array, place it in the sorted array at the index given by the count array for that element value, and decrement the count for that element value. After placing all elements, the sorted array becomes: `sorted_arr = [1, 1, 2, 2, 2, 3, 3, 4, 4, 4]`.
6. The sorted array is now returned: `[1, 1, 2, 2, 2, 3, 3, 4, 4, 4]`.

Complexity

Time Complexity: O(n+k)

Space Complexity: O(k)

▼ Quick Sort

Quick Sort is an in-place and unstable comparison-based sorting algorithm that uses the divide-and-conquer strategy. It selects a pivot element, partitions the array into two sub-arrays, and recursively sorts each sub-array until a sorted array is produced. This algorithm divides the input array into smaller sub-arrays using recursion and combines the results to produce a sorted array.

Advantages

- It is one of the fastest sorting algorithms with an average case time complexity of O(n log n). This makes it ideal for large datasets.
- It sorts the input array in-place, meaning it does not require any additional space beyond the input array. This makes it memory efficient and practical for sorting large datasets.
- It can be easily modified to handle different data types and sort orders by changing the partitioning process and pivot selection.

Disadvantages

- Worst-case performance: In the worst case, Quick Sort can have a time complexity of O(n^2) if the pivot element is consistently chosen poorly, leading to unbalanced partitions.
- It is an unstable sorting algorithm, meaning it may change the relative order of equal elements in the input array.
- It is a recursive algorithm, which means it can be more difficult to implement and debug than iterative sorting algorithms.

Algorithm

1. Select a pivot element from the array. This can be any element, but a good choice is often the middle element.
2. Partition the array into two sub-arrays, such that all elements less than the pivot are in one sub-array, and all elements greater than or equal to the pivot are in the other sub-array. This can be achieved with the help of these rules:
    a. a[j] > a[v] ⇒ j--

b. a[i] > a[j] ⇒ swap

c. a[v] > a[i] ⇒ i++

3. Recursively sort each sub-array by repeating steps 1 and 2.
4. Combine the sorted sub-arrays to produce the final sorted array.

Example

Suppose the input array is `[5, 1, 9, 3, 7, 4, 8, 6, 2]`

1. Choose a pivot element, let's choose 5. Partition the array into left and right sub-arrays:
   Left sub-array: `[1, 3, 4, 2]`
   Right sub-array: `[9, 7, 8, 6]`

2. Recursively sort the left and right sub-arrays. Sorting left sub-array:
   Choose pivot element 1
   Left sub-array: `[]`
   Right sub-array: `[3, 4, 2]`

   Sorting right sub-array:
   Choose pivot element 3
   Left sub-array: `[2]`
   Right sub-array: `[4]`

   Combine sorted sub-arrays: `[2, 3, 4]`
   Combine sorted sub-arrays with pivot element 1: `[1, 2, 3, 4]`

   Sorting right sub-array:
   Choose pivot element 9
   Left sub-array: `[7, 8, 6]`
   Right sub-array: `[]`

   Sorting left sub-array:
   Choose pivot element 7
   Left sub-array: `[6]`
   Right sub-array: `[8]`

Combine sorted sub-arrays: `[6, 7, 8]`
Combine sorted sub-arrays with pivot element 9: `[6, 7, 8, 9]`

3. Combine the sorted sub-arrays with the pivot element 5. Final sorted array: `[1, 2, 3, 4, 5, 6, 7, 8, 9]`

Complexity

Time Complexity: O(n^2)

Space Complexity: O(log n)

▼ Merge Sort

Merge Sort is an efficient, stable, and out-of-place comparison-based sorting algorithm that uses the divide-and-conquer strategy. It sorts an array by dividing it into two halves, sorting each half recursively, and then merging the sorted halves. This algorithm is a general-purpose sorting algorithm that guarantees a time complexity of O(n log n).

Advantages

- It is a stable sorting algorithm, preserving the order of equal elements in the input array.
- It is an efficient sorting algorithm with a time complexity of O(n log n) in the worst case, making it suitable for large input sizes.
- It is a parallelizable algorithm that can take advantage of multiple processors to speed up the sorting process.

Disadvantages

- It requires additional memory proportional to the size of the input array to perform the merging step, making it less suitable for large arrays that cannot fit into memory.
- It has a higher constant factor than other comparison-based sorting algorithms, due to the overhead of the recursive calls and the merging step.
- It may not be the best choice for small input sizes, where the overhead of the algorithm outweighs its efficiency.

Algorithm

1. If the input array has fewer than two elements, return it as is.
2. Divide the input array into two halves.
3. Recursively sort each half using the Merge Sort algorithm.
4. Merge the two sorted halves back together into a single sorted array:
    - Initialize a new empty array.
    - Compare the first elements of the two sorted halves, and append the smaller one to the new array.
    - Continue comparing the next elements of each half, and appending the smaller one to the new array until one of the halves is empty.
    - Append the remaining elements of the non-empty half to the new array.
5. Return the merged and sorted array.

Example

Suppose the array is `[38, 27, 43, 3, 9, 82, 10]`

1. Divide the array into two halves: `[38, 27, 43, 3]` and `[9, 82, 10]`.
2. Recursively sort each half:
    - Sort `[38, 27, 43, 3]` by dividing it into `[38, 27]` and `[43, 3]`, then sorting each half:
        - Sort `[38, 27]` by dividing it into `[38]` and `[27]`, then merging them back into `[27, 38]`.
        - Sort `[43, 3]` by dividing it into `[43]` and `[3]`, then merging them back into `[3, 43]`.
        - Merge the two sorted halves `[27, 38]` and `[3, 43]` back into `[3, 27, 38, 43]`.
    - Sort `[9, 82, 10]` by dividing it into `[9]` and `[82, 10]`, then sorting each half:
        - Sort `[82, 10]` by dividing it into `[82]` and `[10]`, then merging them back into `[10, 82]`.
        - Merge the sorted `[9]` and `[10, 82]` back into `[9, 10, 82]`.

3. Merge the two sorted halves `[3, 27, 38, 43]` and `[9, 10, 82]` back into a single sorted array:
    - Initialize an empty array `merged`.
    - Compare the first elements of each sorted half (`3` and `9`), and append the smaller one (`3`) to `merged`.
    - Compare the next elements (`27` and `9`), and append the smaller one (`9`) to `merged`.
    - Compare the next elements (`27` and `10`), and append the smaller one (`10`) to `merged`.
    - Compare the next elements (`27` and `82`), and append the smaller one (`27`) to `merged`.
    - Compare the next elements (`38` and `82`), and append the smaller one (`38`) to `merged`.
    - Compare the remaining elements (`43` and `82`), and append them in order to `merged`.
    - The merged and sorted array is `[3, 9, 10, 27, 38, 43, 82]`.

Therefore, the Merge Sort algorithm correctly sorts the input array `[38, 27, 43, 3, 9, 82, 10]` into the output array `[3, 9, 10, 27, 38, 43, 82]`.

Complexity

Time Complexity: O(n log n)

Space Complexity: O(n)

▼ Selection Sort

Selection Sort is a simple, in-place, and unstable sorting algorithm that's useful in situations where memory is limited or for small data sets. It repeatedly selects the smallest element from the unsorted part of the list and places it at the beginning of the list. This algorithm is based on comparisons and has a straightforward implementation, requiring little additional memory compared to other sorting algorithms.

Advantages

- Simple to implement and understand.

Disadvantages

- Has a time complexity of O(n^2), which makes it inefficient for large

- It is an in-place sorting algorithm, meaning that it does not require any additional memory.
- Performs well on small datasets.

datasets.

- It is an unstable sorting algorithm, meaning that it does not maintain the relative order of equal elements.
- It always performs the same number of comparisons for a given dataset, regardless of whether the data is already sorted or not, resulting in poor performance for partially sorted datasets.

Algorithm

1. Set the minimum index to 0
2. Iterate through the list of elements from 0 to n-1
3. Set the minimum index to the current index of the outer loop
4. Iterate through the unsorted part of the list from the next index to the end
5. If the element at the current index is less than the element at the minimum index, update the minimum index
6. Swap the element at the minimum index with the first element of the unsorted part of the list
7. Repeat steps 2-6 until the entire list is sorted

Example

Suppose we have an array A = [8, 4, 1, 6, 9, 2, 7]

1. The first element 8 is assigned as the minimum.
2. Compare 8 with the second element 4. Since 4 is smaller than 8, 4 is assigned as the new minimum.
3. Compare the current minimum 4 with the third element 1. Since 1 is smaller than 4, 1 is assigned as the new minimum.
4. Continue comparing the minimum with the remaining elements until the last element.

5. Swap the second smallest element 2 with the second position.
6. Repeat steps 2-5 for the remaining elements of the array.

The sorted array would be: A = [1, 2, 4, 6, 7, 8, 9]

Complexity

Time Complexity: O(n^2)

Space Complexity: O(1)

▼ Bucket Sort

Bucket Sort is a out of place sorting algorithm which is used when the input is uniformly distributed over a range, and the range is known in advance. It is particularly efficient when the input is evenly distributed and the number of elements to be sorted is large. Bucket Sort is also used in parallel computing, as the buckets can be sorted independently on different processors.

Advantages

- It is a simple and easy-to-understand algorithm.
- It is very efficient when the input is uniformly distributed over a range.
- It can be easily parallelized, making it suitable for use in distributed computing environments.

Disadvantages

- It requires additional memory to hold the buckets, which can be a limiting factor for large inputs.
- It is not suitable for inputs that are not uniformly distributed, as this can lead to unbalanced buckets.
- It is out-of-place sorting algorithm, so it may not be suitable for use in memory-constrained environments.

Algorithm

The Bucket Sort algorithm works by dividing the input into a number of buckets, based on their values. Each bucket is then sorted using another sorting algorithm, or recursively using Bucket Sort. Finally, the sorted buckets are merged to produce the sorted output.

1. Choose the number of buckets to use and create them.

2. For each element in the input array, determine which bucket it belongs to and add it to that bucket.
3. Sort each bucket using another sorting algorithm, or recursively using Bucket Sort.
4. Concatenate the sorted buckets to produce the sorted output.

Example

Suppose the array is [54, 46, 83, 66, 95, 92, 43]

1. Determine the minimum and maximum values in the input array: minValue = 43 and
   maxValue = 95
2. Determine the range of values for each bucket: range = (95 - 43) / 5 = 10.4
3. Create empty buckets: buckets = [[], [], [], [], []]
4. Assign each element to the appropriate bucket:

```
for i = 0 to length(array) - 1
    bucketIndex = floor((array[i] - minValue) / range)
    append array[i] to buckets[bucketIndex]

buckets = [    [43, 46],
    [],
    [54],
    [66],
    [83, 95, 92]
]
```

5. Sort each bucket using the Insertion Sort algorithm:

```
for i = 0 to bucketCount - 1
    insertionSort(buckets[i])

buckets = [    [43, 46],
    [],
    [54],
    [66],
    [83, 92, 95]
]
```

6. Concatenate the sorted buckets to produce the sorted output:

```
result = []
for i = 0 to bucketCount - 1
    append all elements in buckets[i] to result

result = [43, 46, 54, 66, 83, 92, 95]
```

7. Thus, the sorted array is `[43, 46, 54, 66, 83, 92, 95]`.

Complexity

Time Complexity: O(n^2)

Space Complexity: O(n + k)

▼ Radix Sort

Radix Sort is a stable and out-of-place sorting algorithm which is generally used when the range of the input numbers is known in advance, and the numbers have the same number of digits. It is commonly used to sort integers or strings, where the elements have a fixed-length representation. Since Radix Sort is a non-comparative sorting algorithm, it is not affected by the distribution of the input data, and it can be faster than other comparison-based sorting algorithms for large datasets. However, it requires additional memory to store the temporary arrays used during the sorting process.

Advantages

- Radix Sort is a linear-time sorting algorithm, meaning that it can sort large datasets in a reasonable amount of time.
- It is not affected by the distribution of the input data, and it can be faster than other comparison-based sorting algorithms for large datasets.
- It is a stable sorting algorithm, meaning that it preserves the relative order of elements with equal keys during the sorting process.

Disadvantages

- Radix Sort requires additional memory to store the temporary arrays used during the sorting process, which can be a disadvantage for large datasets.
- It is only applicable for sorting fixed-length integers or strings, where the elements have a fixed-length representation.

Algorithm

1. Find the maximum element in the input array.
2. For each digit position i from the least significant to the most significant:
   a. Create a bucket array for each digit from 0 to 9.
   b. Traverse the input array and put each element in its corresponding bucket based on the value of its i-th digit.
   c. Merge the bucket arrays into the input array in the order 0 to 9.
3. The input array is now sorted.

The `getDigit` function takes a number and an index as input and returns the i-th digit of the number.

Example

Suppose the array is `arr = [170, 45, 75, 90, 802, 24, 2, 66]`

1. Find the maximum number in the array: `max_num = 802`
2. Loop through each digit of `max_num` from right to left:
   a. Create 10 buckets (0-9)
   b. Place each number in the array into the corresponding bucket based on the current digit being sorted (e.g., if sorting by the ones digit, put `170` in bucket 0, `45` in bucket 5, etc.)
   c. Place the numbers back into the array in order based on the bucket they were placed in
3. Repeat step 2 for the next digit to the left, until all digits have been sorted
4. The final sorted array is `[2, 24, 45, 66, 75, 90, 170, 802]`

Complexity

Time Complexity: O(nk)

Space Complexity: O(n + k)

▼ Binary Search

Binary search is an in-place and unstable searching algorithm that operates on a sorted list or array. It works by repeatedly dividing in half the portion of the list that could contain the desired value, until the value is found or determined to be not present. Because it eliminates half the remaining search space with each iteration.

Advantages

- It has a very fast average time complexity of O(log n), which makes it very efficient for searching large sorted arrays or lists.
- It is reliable because it always finds the target value if it is present in the sorted array or list, and will return "not found" otherwise.
- It is easy to understand and implement because it involves simple calculations and only requires a few variables to keep track of the search range.

Disadvantages

- It requires that the list or array to be searched is sorted. If the list is unsorted, then a separate sorting algorithm must be applied first, which adds to the time complexity.
- It is inefficient for dynamic or changing lists, because inserting or deleting elements requires the entire list to be re-sorted, which again adds to the time complexity.
- It requires additional memory to keep track of the pointers or indices, which can be an issue when searching very large arrays or lists.

Algorithm

1. Begin with the entire sorted array.
2. Set the left pointer to the beginning of the array, and the right pointer to the end of the array.
3. While the left pointer is less than or equal to the right pointer:
    - Calculate the middle index between the left and right pointers: middle = (left + right) / 2
    - If the middle value equals the target value, return the middle index as the location of the target value.
    - If the middle value is greater than the target value, move the right pointer to middle - 1.
    - If the middle value is less than the target value, move the left pointer to middle + 1.
4. If the target value is not found after the while loop, return "not found".

Complexity

Time Complexity: O(log n)

Space Complexity: O(1)