

Bubble sort, insertion sort, selection sort, merge sort, quick sort, heap sort basic to advanced notes with code examples with clear documentation and output



Certainly! Here's an overview of several common sorting algorithms, including Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, and Heap Sort. Each algorithm is presented with basic to advanced explanations, code examples, clear documentation, and sample outputs.

1. Bubble Sort

Description: Bubble Sort is a simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

Algorithm:

1. Iterate over the array.
2. Compare each pair of adjacent elements.
3. Swap them if they are in the wrong order.
4. Repeat the process for the entire array until no swaps are needed.

Code Example (Python):

```
python Copy code

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 90]
print("Bubble Sort:", bubble_sort(arr))
```

Output:

```
less Copy code

Bubble Sort: [11, 12, 22, 25, 34, 64, 90]
```

2. Insertion Sort

Description: Insertion Sort builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into the correct position among the already-sorted items.

Algorithm:

1. Start from the second element (index 1).
2. Compare it with the elements before it and insert it into its correct position.
3. Repeat for all elements.

Code Example (Python):

```
python Copy code

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

```
return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Insertion Sort:", insertion_sort(arr))
```

Output:

less

Space complexity?

Sort algorithm comparison?

Copy code

Insertion Sort: [11, 12, 22, 25, 34, 64, 90]

3. Selection Sort

Description: Selection Sort divides the list into two parts: the sorted part and the unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part.

Algorithm:

1. Find the minimum element from the unsorted part.
2. Swap it with the first element of the unsorted part.
3. Move the boundary between the sorted and unsorted parts.

Code Example (Python):

```
python
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Selection Sort:", selection_sort(arr))
```

Output:

mathematica

Copy code

Selection Sort: [11, 12, 22, 25, 34, 64, 90]

4. Merge Sort

Description: Merge Sort is a divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves back together.

Algorithm:

1. Divide the array into two halves.
2. Recursively sort both halves.
3. Merge the sorted halves.

Code Example (Python):

```
python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
```

```
        i += 1
    else:
        arr[k] = R[j]
        j += 1
        k += 1

    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1

    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Merge Sort:", merge_sort(arr))
```

Output:

mathematica

Copy code

Merge Sort: [11, 12, 22, 25, 34, 64, 90]

5. Quick Sort

Description: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into elements less than the pivot and elements greater than the pivot. It then recursively sorts the partitions.

Algorithm:

1. Choose a pivot element.
2. Partition the array into elements less than and greater than the pivot.
3. Recursively sort the partitions.

Code Example (Python):

python

Copy code

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Quick Sort:", quick_sort(arr))
```

Output:

less

Copy code

Quick Sort: [11, 12, 22, 25, 34, 64, 90]

6. Heap Sort

Description: Heap Sort converts the array into a binary heap and then repeatedly extracts the maximum (or minimum) element from the heap to build a sorted array.

Algorithm:

1. Build a max heap from the input data.
2. Swap the root of the heap with the last element.
3. Reduce the size of the heap and heapify the root.
4. Repeat until the

Code Example (Python):

Code Example (Python):

```
python Copy code

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Heap Sort:", heap_sort(arr))
```

Output:

```
less Copy code

Heap Sort: [11, 12, 22, 25, 34, 64, 90]
```

Feel free to reach out if you need further explanations or additional details on any of these sorting algorithms!

Space complexity?

Sort algorithm comparison?

code documentation missing and timecomplexity



Certainly! Here's a more detailed explanation of each sorting algorithm, including time complexity and code documentation.

1. Bubble Sort

Description: Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Time Complexity:

- Best: $O(n)$ (when the array is already sorted)
- Average: $O(n^2)$
- Worst: $O(n^2)$

Space complexity?

Sort algorithm comparison?

Code Example (Python):

```
python Copy code

def bubble_sort(arr):
    """
    Sorts an array using the Bubble Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list.

    Time Complexity:
    - Best Case:  $O(n)$  - when the array is already sorted.
    - Average Case:  $O(n^2)$ .
    - Worst Case:  $O(n^2)$ .
    """
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```
n = len(arr)
for i in range(n):
    swapped = False
    for j in range(n-i-1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
            swapped = True
    if not swapped:
        break
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Bubble Sort:", bubble_sort(arr))
```

Output:

```
less Copy code

Bubble Sort: [11, 12, 22, 25, 34, 64, 90]
```

2. Insertion Sort [Space complexity?](#) [Sort algorithm comparison?](#)

Description: Insertion Sort builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into the correct position among the already-sorted items.

Time Complexity:

- Best: $O(n)$ (when the array is already sorted)
- Average: $O(n^2)$
- Worst: $O(n^2)$

Code Example (Python):

```
python Copy code

def insertion_sort(arr):
    """
    Sorts an array using the Insertion Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted array.

    Time Complexity:
    - Best Case:  $O(n)$  - when the array is already sorted.
    - Average Case:  $O(n^2)$ .
    - Worst Case:  $O(n^2)$ .
    """
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Insertion Sort:", insertion_sort(arr))
```

Output:

```
less Copy code

Insertion Sort: [11, 12, 22, 25, 34, 64, 90]
```

3. Selection Sort

Description: Selection Sort divides the list into two parts: the sorted part and the unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part.

Time Complexity:

- Best: $O(n^2)$
- Average: $O(n^2)$
- Worst: $O(n^2)$

Code Example (Python):



```
python
Space complexity?
Sort algorithm comparison?
Copy code

def selection_sort(arr):
    """
    Sorts an array using the Selection Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list.

    Time Complexity:
    - Best Case:  $O(n^2)$ .
    - Average Case:  $O(n^2)$ .
    - Worst Case:  $O(n^2)$ .
    """
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Selection Sort:", selection_sort(arr))
```

Output:

```
mathematica
Copy code

Selection Sort: [11, 12, 22, 25, 34, 64, 90]
```

4. Merge Sort

Description: Merge Sort is a divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves back together.



Time Complexity:

Space complexity?

Sort algorithm comparison?

- Best: $O(n \log n)$
- Average: $O(n \log n)$
- Worst: $O(n \log n)$

Code Example (Python):

```
python
Copy code

def merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list.

    Time Complexity:
    - Best Case:  $O(n \log n)$ .
    - Average Case:  $O(n \log n)$ .
    - Worst Case:  $O(n \log n)$ .
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)
```

```
def merge_sort(L):
    i = j = k = 0

    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1

    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Merge Sort:", merge_sort(arr))
```

Output:

```
mathematica
Merge Sort: [11, 12, 22, 25, 34, 64, 90]
```

5. Quick Sort

Description: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into elements less than the pivot and elements greater than the pivot. It then recursively sorts the partitions.

Time Complexity:

- Best: $O(n \log n)$
- Average: $O(n \log n)$
- Worst: $O(n^2)$ (when the pivot is the smallest or largest element)

Code Example (Python):

```
python
def quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list.

    Time Complexity:
    - Best Case:  $O(n \log n)$ .
    - Average Case:  $O(n \log n)$ .
    - Worst Case:  $O(n^2)$ .
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Quick Sort:", quick_sort(arr))
```

Output:

Quick Sort: [11, 12, 22, 25, 34, 64, 90]

6. Heap Sort

Description: Heap Sort converts the array into a binary heap and then repeatedly extracts the maximum (or minimum) element from the heap to build a sorted array.

Time Complexity:

- Best: $O(n \log n)$
- Average: $O(n \log n)$
- Worst: $O(n \log n)$



Code Example (Python)

Space complexity?

Sort algorithm comparison?

python

Copy code

```
def heapify(arr, n, i):
    """
    Ensures that the subtree rooted at index i in the array is a max heap.

    Parameters:
    arr (list): The list representing the heap.
    n (int): The size of the heap.
    i (int): The index of the root element of the subtree.

    Time Complexity:
    - Best Case:  $O(\log n)$ .
    - Average Case:  $O(\log n)$ .
    - Worst Case:  $O(\log n)$ .
    """
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[i]:
        largest = l
    if r < n and arr[r] > arr[l]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    """
    Sorts an array using the Heap Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list.

    Time Complexity:
    - Best Case:  $O(n \log n)$ .
    - Average Case:  $O(n \log n)$ .
    - Worst Case:  $O(n \log n)$ .
    """
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

# Example Usage
arr = [64, 34, 25, 12, 22, 11, 90]
print("Heap Sort:", heap_sort(arr))
```

Output:

less

Copy code

Heap Sort: [11, 12, 22, 25, 34, 64, 90]

Feel free to reach out if you have any more questions or need further assistance!



Space complexity?

Sort algorithm comparison?



Message ChatGPT



ChatGPT can make mistakes. Check important info.

