Sorting Algorithms

Sorting Algorithms - Geeksforgeeks

Table for the complexity comparison:

Name	Best Case	Average Case	Worst Case	Memory	Stable	Method Used
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2 \log n)$	No	No	Partitioning
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Insertion & Merging
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Shell Sort	$O(n \log n)$	$O(n^{4/3})$	$O(n^{3/2})$	$O(1)$	No	Insertion
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Tree Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Insertion
Cycle Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Strand Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	Yes	Selection
Cocktail Shaker Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging

Name	Best Case	Average Case	Worst Case	Memory	Stable	Method Used
Comb Sort	$O(n \log n)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Exchanging
Gnome Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Odd–even Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging

Sorting Algorithms Overview

Sorting Algorithm	Average-Case Time Complexity	Worst-Case Time Complexity	Space Complexity
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(n + k)$

Merge Sort Explanation

- An array A with elements $A[1]$ to $A[n]$ is considered.
- The approach follows a divide and conquer strategy to reduce the problem size.
  - The array is divided into two halves, $A[1 \dots \frac{n}{2}]$ and $A[\frac{n}{2} + 1 \dots n]$.
- Then Merge the sorted array back

```
merge(A[1,...,n], B[1,...,m]) {
    C[1,...,n+m],
    i=j=k=1;
    while(i<=n && j<=m) {
        if(A[i] <= B[j]) {
            C[k] = A[i],
            i++, k++;
        } else {
            C[k] = B[j],
            j++, k++;
        }
    }
    if(i<=n) C[k:] = A[i:];
    if(j<=m) C[k:] = B[j:];
    return C;
}
```

- Time Complexity for merge function: O(n+m)
- Merge Sort Algorithm Complete Process:

```
merge_sort(A[...n]) {
    if (n < 2) return A;
    left = merge_sort(A[1 : n/2]);
    right = merge_sort(A[n/2 + 1 : n]);
    return merge(left, right);
}
```

- T(n) = 2T(n/2) + n
- **Recurrence Relation for Merge Sort:**
  - $T(n) = 2T(\frac{n}{2}) + n$
  - $a = 2, b = 2, c = 1$
  - $T(n) = \Theta(n \log n)$
- **Space Complexity for Merge Sort:**
  - $S(n) = \Theta(n)$
  - Note: Not in-place; needs an array to merge and copy over.

Derivation of Time Complexity:

- $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + \frac{n}{2^{\log n}}$
- Summed up equals: $n(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{\log n}})$

- Which simplifies to: $\Theta(n \log n)$

Quick Sort Algorithm

Quick Sort Overview

- Relies on **Partitioning**
- **Pivot** is a number in A
- A[1...n]

```
Quick-Sort(A[1...n]) {
    if (n < 2) return A;
    pivot_index = Partition(A[1...n]);
    Quick-Sort(A[1 : pivot_index-1]);
    Quick-Sort(A[pivot_index+1 : n]);
}
```

Time Complexity Analysis

- Best case: $T(n) = 2T(\frac{n}{2}) + n$, which simplifies to $\Theta(n \log n)$
- Worst case: $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$
- Bounds: $\frac{n \log n}{3} \leq T(n) \leq n \log \frac{3n}{2}$

Quick Sort Algorithm Continued

Partitioning in Quick Sort

```
Partition(A[1...n]) {
    Pivot = A[n];
    i = 0; // i = -1 if starting from index 0
    for (curr = 1; curr <= n-1; curr++) {
        if (A[curr] <= Pivot) {
            Swap(A[curr], A[i+1]);
            i++;
        }
    }
    Swap(A[n], A[i+1]);
    return i+1;
}
```

- Time Complexity for partition function: $T(n) = O(n)$

- Space Complexity for partition function: $S(n) = O(1)$

First Version of Partitioning

- Choose pivot as the last number.
- Procedure includes:
    i. If the current element is greater than the pivot, increment the current pointer.
    ii. If the current element is less than or equal to the pivot, swap with the element at the incrementing index.

The notes also contain an example illustrating the partitioning process within an array, showing how elements are swapped based on their comparison with the pivot.

Example:
Array A = `[10, 2, 3, 7, 8, 9, 4, 6]`

Partitioning steps:

1. Pivot chosen is the last element, A[7] = 6.
2. Initialization: i points to the start of the array, curr is the current index starting from A[1].

During partitioning:

- Elements less than or equal to pivot (6) are moved to the left of the pivot.
- Elements greater than pivot are kept on the right.

After partitioning:

- The pivot (6) is placed after all elements smaller than or equal to it.
- The array is now partially sorted around the pivot.

Resulting array:

- Elements to the left of the pivot are less than or equal to 6.
- Elements to the right of the pivot are greater than 6.

Quick Sort Algorithm - Detailed Partition Function

Partition Function Code

```
Partition(A[1...n]) {
    Pivot = A[n];
    i = 0; // i = -1 if starting from index 0
    for (curr = 1; curr <= n-1; curr++) {
        if (A[curr] <= Pivot) {
            Swap(A[curr], A[i+1]);
            i++;
        }
    }
    Swap(A[n], A[i+1]);
    return i+1;
}
```

Time and Space Complexity for Partition Function

- $T(n) = \Theta(n)$
- $S(n) = \Theta(1)$
- Note: Quick Sort is an in-place sort.
- The notes emphasize the in-place nature of Quick Sort and provide detailed steps for the partition function, which is central to the Quick Sort algorithm. The function swaps elements in the array based on their comparison to the pivot value, and rearranges the array so that elements less than the pivot are on the left and elements greater than the pivot are on the right.

Heap Sort and Heap Data Structure

Heap Operations and Complexity

- Heap (Max)

Operation	Array Unsorted	Array Sorted	Binary Tree Sorted
Insert	$O(1)$	$O(n)$	$O(\log n)$
Get Max	$O(n)$	$O(1)$	$O(\log n)$
Delete Max	$O(n)$	$O(1)$	$O(\log n)$

Heap as a Dynamic Set of Numbers

- Represented in an array format, both physical and logical.
- "Almost" full binary tree.
- Heap property:
  - For Max heap: Parent $\geq$ Child

```
      100
     /   \
   50     30
  / \     / \
 20  40  15  10
```

  - For Min heap: Parent $\leq$ Child

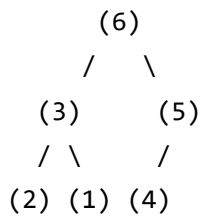
```
        10
      /    \
    15      20
   / \      / \
 40  50   30  100
```

Heap Structure Example

- A binary tree example is shown with nodes labeled and a corresponding array representation.
- To find the parent of a node at index i: $\text{index}/2$
- To find the children of a node at index i:
  - Left child: $2 \times \text{index}$
  - Right child: $2 \times \text{index} + 1$

Graphical Representation of Heapify Process

- The root node represents the largest element in a max heap.
- The `heapify` function compares the root with its children and swaps it with the largest of the two if the root is not the largest.
- After a swap, the `heapify` function is called recursively on the subtree rooted at the child that was swapped to ensure it too satisfies the max heap property.



In the array representation of the heap, the elements are stored as follows:

```
Array A = [6, 3, 5, 2, 1, 4]
```

The underscores represent that heap arrays are typically 1-indexed in educational contexts for easier arithmetic relating to parent and child indices:

- The parent index is given by `i/2`.
- The left child index is `2*i`.
- The right child index is `2*i + 1`.

Heap Sort Operations Code

Get Max Function

```
get_max() {
    return A[1];
}
```

Insert Function

```
insert(k) {
    n = n + 1;
    A[n] = k;
    i = n;
    while(parent(i) > 0 && A[i] > A[parent(i)]) {
        swap(A[i], A[parent(i)]);
        i = parent(i);
    }
}
```

Delete Max Function

```
delete_max() {
    swap(A[1], A[n]);
    n = n - 1;
    heapify(1);
}
```

Helper Function: Heapify

```
heapify(i) {
    largest = i;
    if (left(i) <= n && A[left(i)] > A[largest]) {
        largest = left(i);
    }
    if (right(i) <= n && A[right(i)] > A[largest]) {
        largest = right(i);
    }
    if (i == largest) return;
    swap(A[i], A[largest]);
    heapify(largest);
}
```

- The notes state that the time complexity for the heapify function is $O(\log n)$.

Heap Sort Algorithm - Detailed Heapify Function

Heapify Function

```
// Helper Function: Heapify
heapify(i) {
    largest = i;
    if (left(i) <= n && A[left(i)] > A[largest]) {
        largest = left(i);
    }
    if (right(i) <= n && A[right(i)] > A[largest]) {
        largest = right(i);
    }
    if (i != largest) {
        swap(A[i], A[largest]);
        heapify(largest);
    }
}
```

- Time complexity: T(n) = T(n/2) + 1 = O(log n)

Heap Sort Algorithm

If we have a max-heap, keep calling delete function will sort the array.

```
make_heap(A[1...n]);

for (i = 1 to n-1) {
  delete_max();
}

// Building a max-heap
for (i = n/2 downto 1) {
  heapify(i);
}
```

Time complexity: T(n) = O(n)

```
// Inserting elements into the heap
for (i = 1 to n) {
    insert(n);
}
```

Time complexity: T(n) = O(n*log n)

The notes outline the process of heap sort, which involves building a max-heap and then repeatedly calling `delete_max` to sort the array. The `heapify` process is used during heap construction, and the `insert` method is used for adding elements, both with their respective time complexities.

The note also includes a graphical representation of the heap, but in markdown, we can only describe it textually.

```
Heap Structure:

            (10)
           /    \
         (9)    (3)
        /  \     / \
      (6)  (8) (2) (1)
     /  \   /

(4) (5) (7)
```

The heap is represented as a binary tree with nodes labeled from 1 to 9, indicating the positions in the array. The tree is almost complete, with all levels fully filled except possibly the last level, which is filled from left to right.

To maintain the max-heap property, the `heapify` function is called recursively to ensure that for every node `i` other than the root, the value in `i` is less than or equal to the value in its parent.

It shows a max-heap and indicates that the `heapify` function is called to maintain the max-heap property for the subtree.

Comparison-based Sorting and Counting Sort

Comparison-based Sorting

Any comparison-based sorting algorithm has a lower bound on its running time of $\Omega(n \log n)$.

This statement provides the lower bound for the running time of any algorithm that relies on comparisons to sort a list of items, such as Quick Sort, Merge Sort, or Heap Sort.

Counting Sort

Counting Sort Example

Given Array A

```
A = [2, 3, 1, 4, 1, 2]
```

Working Array B (Count Array)

```
B = [0, 2, 2, 1, 1]
```

This assumes the counts start at index 1 for the value 1, index 2 for the value 2, and so on.

Transformed Working Array B (Cumulative Count)

```
B = [0, 2, 4, 5, 6]
```

After accumulating counts; B[i] now contains the number of elements less than or equal to i.

Result Array C (Sorted Array)

```
C = [1, 1, 2, 2, 3, 4]
```

The sorted array, constructed using the counts from array B.

These arrays illustrate the intermediate steps in the Counting Sort algorithm, where `B` is used to count the occurrences of each number, and `C` is the output sorted array. The values in array `B` are transformed into a cumulative count to determine the positions of each element in the sorted array `C`.

Process:

1. Count occurrences of each number in A and store in B.
2. Modify B such that each index contains the sum of previous counts.
3. Place each number in the correct position in C by using the counts in B.

The notes provide an example of the Counting Sort algorithm, which uses an intermediate array to count occurrences and then sorts the array. This is particularly effective when the range of potential

values (k) is not significantly greater than the number of elements (n).

Counting Sort Algorithm

```
counting_sort(A[1...n], k) {
    for i = 1 to n {
        B[A[i]] += 1;
    }
    for i = 2 to k {
        B[i] = B[i] + B[i-1];
    }
    for i = n downto 1 {
        C[B[A[i]]] = A[i];
        B[A[i]] -= 1;
    }
    return C;
}
```

- Counting part is $O(n)$
- Prefix sum part is $O(k)$
- Sorting part is $O(n)$
- Overall time complexity: $T(n) = O(n + k)$
- Counting Sort is a "stable" sort.

Counting Sort algorithm is a non-comparison-based sorting algorithm.

It is efficient when the range of input data (k) is not significantly greater than the number of elements to be sorted (n).

The algorithm consists of three main steps:

- counting the occurrences of each element,
- calculating the prefix sum to determine the positions of elements,
- and placing the elements in a sorted array.

The algorithm is noted for being stable, which means that it maintains the relative order of records with equal keys.

Review of Counting Sort

- Array `A` contains `n` integers where each `A[i]` is between `1` and `k`.

- The runtime complexity of counting sort is $O(n + k)$.
- For large `k`, Radix Sort or Digit Sort can optimize the process.

Example: 3-digit integers

- Original array: `145, 093, 888, 047, 012, 100, 099`
- Steps:
  - Compare last digits
  - Compare middle digits
  - Compare first digits

LSD (Least Significant Digit) to MSD (Most Significant Digit) Sort

- Sorting by each digit from LSD to MSD.
- For each phase, apply a stable sort (like counting sort) on the digit of interest.

The content describes the optimization of counting sort when the range `k` is large by using Radix Sort or Digit Sort, which involves sorting numbers by individual digits.

Counting Sort Algorithm Details

```
Input: A[1...n], k
for i = 1 to n do
    B[f(A[i])] += 1;

for i = 2 to k do
    B[i] = B[i] + B[i - 1];

for i = n downto 1 do
    C[B[f(A[i])]] = A[i];
    B[f(A[i])] -= 1;

return C;
```

- Function `f(x)` is used to map a number to its corresponding digit:
  - For the last digit: `f(x) = x % 10`
  - For the middle digit: `f(x) = (x / 10) % 10`
  - For the first digit: `f(x) = (x / 100) % 10`

This section provides the pseudo-code for the Counting Sort algorithm and describes the function `f(x)` that extracts the relevant digit from the number for sorting.

Understanding the Counting Sort Algorithm and Its Complexity

- The range of the counting sort algorithm is defined as 0 to k.
- The function `f(A[i])` maps the element $A[i]$ to the correct index in the count array B.
- The time complexity of the counting sort algorithm is $O(d(n+k))$:
  - d is the number of digits in the numbers being sorted.
  - n is the length of the input array.
  - k is the range of the input values (e.g., the max value).

Example of Counting Sort on 3-digit Numbers

For an array of 3-digit numbers, the intermediate steps would look like this:

- Input array: $[002, 011, 321, 175]$ with $k = 999$.
- Bit operations might be used to isolate each digit of the numbers.

The notes describe the counting sort's range and the function `f(A[i])` for mapping elements to indices in the count array. It also introduces bit operations for digit isolation, which can be helpful in radix sort implementations.

Counting Sort vs. Radix Sort Complexity

Counting Sort and Radix Sort do not strictly have a linear running time; it depends on the specific use case.

Complexity Comparison

- **Counting Sort**: $O(n+k)$
- **Radix Sort**: $O(d(n+k))$

Use Cases

- Counting Sort is suitable for smaller ranges of integers or when k is not significantly larger than n.
- Radix Sort is more efficient for larger ranges or when the numbers have more digits, as it processes each digit separately.

Example Use Case Analysis

- For 10 million 3-digit numbers, Counting Sort is efficient.
- For 10 million 32-bit integers, Radix Sort is a better choice due to the larger range of values.

The notes explain the scenarios where Counting Sort and Radix Sort are most suitable, highlighting that the choice of algorithm depends on the range of the numbers to be sorted and the length of the input array.