An Introduction Classical Sorting Algorithms (Implementations and Analysis of running times)

Piseth Tang

September 29, 2024

1 Introduction

It is only right for us to start with the definition of our problem, in particular, the sorting problem which can be stated as follows:

Sorting Problem Definition

Given an array A of size n whose elements belong to a totally ordered set[a], permute the elements in such a way that the result yields the following relation:

$$A[0] \leq A[1] \leq \ldots \leq A[n-1].$$

[a] A set is totally ordered if given any two distinct elements, it is always possible to establish which one of the two is larger than the other.

It is noteworthy to add that the above statement is very general, due to the fact that a set is a very abstract mathematical object. Therefore, sorting an array of strings falls within the same class of problems of sorting an array of integers or other very complex data types, which could be created by programming constructs such as *struct* or *class*. Imagine sorting, not just an integer, but a pair of integers (x, y).

The sorting problems have vast applications (it usages in a library management system to sort millions of books according to some criteria, in our own computer's file management system and etc.) Not withstanding the fact that sorting is a frequent operation, it can be used as a subroutine to speed-up other algorithm, for instance in the case of Dichotomic (Binary) Searching algorithm (which we will see later).

In this document, we will try to introduce the readers to the ideas behind each sorting algorithm and then perform best, worst and average [1] case analyses on most of them.

[1] It should be noted that average analysis is generally very desirable but just like anything that is desirable, it is not a trivial matter to perform a rigorous and complete average case analysis as it would go well beyond the scope of this document (see [1] for those who are interested)

2 Intuition behind each algorithm

2.1 Bubble Sort

Example: Consider a random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted.

Steps: 1. Compare the first two elements. If the first is greater than the second, swap them. 2. Move to the next pair of adjacent elements and repeat. 3. Continue until the end of the array. This completes one pass. 4. Repeat the process for the remaining elements until the array is sorted.

Visualization:

Bubble Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

Pass 1: 1. Compare 64 and 25, swap:

$$[25, 64, 12, 22, 11, 90, 37]$$

2. Compare 64 and 12, swap:

$$[25, 12, 64, 22, 11, 90, 37]$$

3. Compare 64 and 22, swap:

$$[25, 12, 22, 64, 11, 90, 37]$$

4. Compare 64 and 11, swap:

$$[25, 12, 22, 11, 64, 90, 37]$$

5. Compare 64 and 90: no swap.
6. Compare 90 and 37, swap:

$$[25, 12, 22, 11, 64, 37, 90]$$

Pass 2: 1. Compare 25 and 12, swap:

$$[12, 25, 22, 11, 64, 37, 90]$$

2. Compare 25 and 22, swap:

$$[12, 22, 25, 11, 64, 37, 90]$$

3. Compare 25 and 11, swap:

$$[12, 22, 11, 25, 64, 37, 90]$$

4. Compare 25 and 64: no swap.
5. Compare 64 and 37, swap:

$$[12, 22, 11, 25, 37, 64, 90]$$

Pass 3: 1. Compare 12 and 22: no swap. 2. Compare 22 and 11, swap:

$$[12, 11, 22, 25, 37, 64, 90]$$

3. Compare 22 and 25: no swap.
4. Compare 25 and 37: no swap.
5. Compare 37 and 64: no swap.

Pass 4: 1. Compare 12 and 11, swap:

$$[11, 12, 22, 25, 37, 64, 90]$$

Sorted Array:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.2 Selection Sort

Example: Consider the random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Selection Sort divides the array into a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region

and moves it to the end of the sorted region.

Steps: 1. Start with the first element as the minimum. 2. Compare it with the rest of the elements to find the smallest. 3. Swap it with the first element. 4. Repeat the process for the rest of the array.

Visualization:

Selection Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Find minimum (11):

$$[11, 25, 12, 22, 64, 90, 37]$$

2. Find minimum (12):

$$[11, 12, 25, 22, 64, 90, 37]$$

3. Find minimum (22):

$$[11, 12, 22, 25, 64, 90, 37]$$

4. Find minimum (25):

$$[11, 12, 22, 25, 64, 90, 37]$$

5. Find minimum (37):

$$[11, 12, 22, 25, 37, 90, 64]$$

6. Find minimum (64):

$$[11, 12, 22, 25, 37, 64, 90]$$

7. End:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.3 Insertion Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Insertion Sort builds the sorted array one element at a time by repeatedly taking the next element from the input and inserting it into the correct position in the sorted part.

Steps: 1. Start with the first element (consider it sorted). 2. Take the next element and compare it with the sorted part. 3. Shift the sorted elements if necessary and insert the current element.

Visualization:

Insertion Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Current (25):

$$[25, 64, 12, 22, 11, 90, 37]$$

2. Current (12):

$$[12, 25, 64, 22, 11, 90, 37]$$

3. Current (22):

$$[12, 22, 25, 64, 11, 90, 37]$$

4. Current (11):

$$[11, 12, 22, 25, 64, 90, 37]$$

5. Current (90):

$$[11, 12, 22, 25, 64, 90, 37]$$

6. Current (37):

$$[11, 12, 22, 25, 37, 64, 90]$$

7. End:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.4 Merge Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Merge Sort divides the array into halves, recursively sorts each half, and merges the sorted halves back together.

Steps: 1. Divide the array into two halves until each subarray contains one element. 2. Merge the subarrays back together while sorting.

Visualization:

Merge Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Split:

$$[64, 25, 12] \quad \text{and} \quad [22, 11, 90, 37]$$

2. Split again:

$$[64] \quad [25] \quad [12] \quad [22] \quad [11] \quad [90] \quad [37]$$

3. Merge:

$$[25, 64] \quad [11, 22, 90, 37]$$

4. Merge:

$$[12, 25, 64] \quad [11, 22, 37, 90]$$

5. Final Merge:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.5 Quick Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Quick Sort selects a 'pivot' element from the array and partitions the other elements into two subarrays according to whether they are less than or greater than the pivot.

Steps: 1. Choose a pivot element. 2. Partition the array into elements less than and greater than the pivot. 3. Recursively apply the same steps to the subarrays.

Visualization:

Quick Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Pivot (22):

$$[11, 12] \quad [22] \quad [64, 25, 90, 37]$$

2. Sort left:

$$[11, 12]$$

3. Sort right:

$$[25, 37, 64, 90]$$

4. Final:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.6 Shell sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Shell Sort is an optimization of Insertion Sort that allows the exchange of items that are far apart. The idea is to arrange the list of elements so that, taking elements at regular intervals, we can sort the array more efficiently.

Steps: 1. Start with a large gap and reduce the gap gradually. 2. For each gap, perform an insertion sort on the elements at that gap distance.

Visualization:

Shell Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Gap 3:

$$[11, 25, 12, 22, 64, 90, 37]$$

2. Gap 2:

$$[11, 12, 22, 25, 64, 90, 37]$$

3. Final:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.7 Stooge Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Stooge Sort is a recursive sorting algorithm. It is not efficient for large lists, but it serves as an interesting example of a naive sorting algorithm.

Steps: 1. If the first element is greater than the last, swap them. 2. If there are three or more elements, recursively sort the first two-thirds and the last two-thirds of the list.

Visualization:

Stooge Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Swap (64 and 11):

$$[11, 25, 12, 22, 64, 90, 37]$$

2. Sort first two-thirds:

$$[11, 12, 22, 25, 64, 90, 37]$$

3. Sort last two-thirds:

$$[11, 12, 22, 25, 37, 64, 90]$$

4. Final:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.8 Bogo Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Bogo Sort is a highly ineffective sorting algorithm based on generating random permutations of the input until it finds one that is sorted.

Steps: 1. Randomly shuffle the array. 2. Check if the array is sorted. 3. Repeat until sorted.

Visualization:

Bogo Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Shuffle:

$$[11, 22, 64, 25, 12, 90, 37]$$

2. Check sorted: Not sorted. 3. Shuffle:

$$[64, 12, 25, 90, 22, 11, 37]$$

4. Check sorted: Not sorted. 5. Shuffle:

$$[11, 12, 22, 25, 37, 64, 90]$$

6. Check sorted: Sorted!

2.9 Distribution Sorting Algorithms

2.10 Counting Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Counting Sort is a non-comparison-based sorting algorithm that counts the occurrences of each unique value.

Steps: 1. Count each element's occurrence. 2. Calculate the position of each element in the output array. 3. Place elements in the correct position based on the count.

Visualization:

Counting Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Count occurrences:

$$[11:1, 12:1, 22:1, 25:1, 37:1, 64:1, 90:1]$$

2. Position array:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.11 Bucket sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Bucket Sort distributes the elements into several buckets and then sorts these buckets individually.

Steps: 1. Create buckets for ranges of values. 2. Sort each bucket (using another sorting algorithm). 3. Concatenate the sorted buckets.

Visualization:

Bucket Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Buckets:

$$B_1 = [11, 12] \quad B_2 = [22] \quad B_3 = [25] \quad B_4 = [37] \quad B_5 = [64] \quad B_6 = [90]$$

2. Sort each bucket:

$$B_1 = [11, 12] \quad B_2 = [22] \quad B_3 = [25] \quad B_4 = [37] \quad B_5 = [64] \quad B_6 = [90]$$

3. Concatenate:

$$[11, 12, 22, 25, 37, 64, 90]$$

2.12 Radix Sort

Example: Consider the same random array: $A = [64, 25, 12, 22, 11, 90, 37]$

Idea: Radix Sort processes the digits of numbers in a positional manner.

Steps: 1. Sort the array based on the least significant digit. 2. Then sort by the next significant digit and so on.

Visualization:

Radix Sort Steps

Initial Array:

$$[64, 25, 12, 22, 11, 90, 37]$$

1. Sort by unit place:

$$[11, 12, 22, 25, 64, 37, 90]$$

2. Sort by tens place:

$$[11, 12, 22, 25, 37, 64, 90]$$

3 Implementation with counting on the number of elementary operations

3.1 Bubble Sort

```python
def bubble_sort_with_m sure(a: list) -> tuple:
    n = len(a)
    swaps, comparisons = 0,0
    # a = copy.deepcopy(a)
    if n <= 1: return a, swaps, comparisons # Nothing to sort
    # Outer loop includes i = 1
    for i in range(n-1, 0, -1): # Comparison: i <= 1 (or is i == 1)
    ?
        for j in range(0,i):    # Comparison: j >= i (or is j == i)
    ?
            comparisons = comparisons + 1 # If this line is reached
    , 3 comparisons must have been performed.
            if a[j] > a[j+1]:   # Comparison: as is
                a[j], a[j+1] = a[j+1], a[j] # Swap: only when a[j]
    > a[j+1]
                swaps = swaps + 1
    return a, swaps, comparisons
```

Listing 1: Python example

3.2 Insertion Sort

```python
def insertion_sort(t):
    comps = 0
    moves = 0
    if len(t) <= 1:
        return t, 0, 0
    for i in range(1, len(t)):
        j = i
        # swap until right position found...
        comps += 1
        while j >= 1 and t[j - 1] > t[j]:
            (t[j - 1], t[j]) = (t[j], t[j - 1])
            moves += 3
            j = j - 1
            comps += 1
    return t, comps, moves
```

Listing 2: Insertion Sort

3.3 Merge Sort

```python
def merge_with_metrics(l1, l2):
    merged_list = []
    i, j = 0, 0
    comparisons, moves = 0, 0

    while i < len(l1) and j < len(l2):
        comparisons += 1
        if l1[i] <= l2[j]:
            merged_list.append(l1[i])
            i += 1
        else:
            merged_list.append(l2[j])
            j += 1
        moves += 1

    while i < len(l1):
        merged_list.append(l1[i])
        i += 1
        moves += 1

```

```
    while j < len(l2):
        merged_list.append(l2[j])
        j += 1
        moves += 1

    return merged_list, comparisons, moves

def merge_sort_with_metrics(lst):
    if len(lst) <= 1:
        return lst, 0, 0

    mid = len(lst) // 2
    left_half, comp_left, moves_left = merge_sort_with_metrics(lst
    [:mid])
    right_half, comp_right, moves_right = merge_sort_with_metrics(
    lst[mid:])

    merged_list, comp_merge, moves_merge = merge_with_metrics(
    left_half, right_half)
    total_comparisons = comp_left + comp_right + comp_merge
    total_moves = moves_left + moves_right + moves_merge

    return merged_list, total_comparisons, total_moves
```

Listing 3: Mergesort (Not in-place)

3.4 Quick Sort

```
def quick_choose_pivot(t, begin, end):
    return (begin+end)//2


def quick_sort_partition_in_place(t, begin, end):
    pivot_pos = quick_choose_pivot(t, begin, end)
    moves = 3
    t[begin], t[pivot_pos] = t[pivot_pos],t[begin]
    left = begin+1
    right = end-1
    comps = 0
    while right >= left:
        comps += 1
        if t[left] < t[begin]:
            left += 1
        else:
            comps += 1
            if t[right] >= t[begin]:
                right -= 1
            else:
                moves += 3
                t[left], t[right] = t[right], t[left]
                left += 1
                right -= 1
    moves += 1
    t[begin], t[right] = t[right], t[begin]
    return t, right, comps, moves

# this partition is not in place!
def quick_sort_partition(t, begin, end):
    pivot_pos = quick_choose_pivot(t, begin, end)
    l = []
    r = []
    comps = 0
    moves = 0
```

```
    for i in range(begin, pivot_pos):
        comps += 1
        moves += 1
        if t[i] < t[pivot_pos]:
            l.append(t[i])
        else:
            r.append(t[i])
    for i in range(pivot_pos+1, end):
        comps = 0
        moves = 0
        if t[i] < t[pivot_pos]:
            l.append(t[i])
        else:
            r.append(t[i])
    pivot = t[pivot_pos]
    moves += 1
    for i in range(0, len(l)):
        moves += 1
        t[begin+i] = l[i]
    moves += 1
    t[begin+len(l)] = pivot
    for i in range(0, len(r)):
        moves += 1
        t[begin+i+len(l)+1] = r[i]
    return t, begin+len(l), comps, moves
```

Listing 4: Quick Sort (Not in-place)

3.5 Counting Sort

```
def counting_sort(t):
    minimum = min(t)  # linear
    maximum = max(t)  # linear
    counters = [0]*(maximum-minimum+1)
    # count number of times an element is the array
    for e in t:
        counters[e-minimum] += 1
    current_pos = 0
    for i in range(0, len(counters)):
        for j in range(0, counters[i]):
            t[current_pos] = i+minimum
            current_pos += 1
    return t
```

Listing 5: Counting Sort (Not in-place)

3.6 Radix Sort

```
def counting_sort(t):
    minimum = min(t)  # linear
    maximum = max(t)  # linear
    counters = [0]*(maximum-minimum+1)
    # count number of times an element is the array
    for e in t:
        counters[e-minimum] += 1
    current_pos = 0
    for i in range(0, len(counters)):
        for j in range(0, counters[i]):
            t[current_pos] = i+minimum
            current_pos += 1
    return t
```

Listing 6: Radix Sort (Not in-place)

4 Characteristics of sorting algorithms

We have decided to placed this section last to not let the readers be bogged down with definitions that may obscure the whole purpose behind this document. As we have already examined the computing for the mentioned sorting algorithms, other criteria may determine the choice of the sorting algorithm **in practice.** The first is obviously space, especially if the array to be sorted has a very large size. In fact, the model of computation **R.A.M** (Random-Access Machine) [5] assumes that the data that each of our algorithm can process **fits** into the main/physical memory. However, again, in practice, this is not always the case where we have to move either some or all of the data, in this case - to be sorted, into the hard disk. Then, we will have to do all of the analyses above within the framework of a different model of computation. This, of course, goes beyond the scope of algorithms and data structures. The interested readers will have to find books and resources on *Computer Architecture and Organization.*

4.1 In-place

Definition

A sorting algorithm is said to be in-place if it uses just $\Theta(1)$ memory.

In some cases, especially when we are not dealing with a set of data that is fixed in time, but rather with a dynamical database where new elements may be added on-the-go, it is more efficient to employ algorithms that do not have to restart the sorting process from zero after each insertion. This leads to the following definition.

4.2 Online

Definition

A sorting algorithm is said to be online if it can be executed as data drive without waiting for all the elements to be known.

An algorithm is one that will work if the elements to be sorted are provided one at a time, with the understanding that the algorithm must keep the element sorted as more elements are added. In contrast, offline algorithms must be given the whole problem data from the beginning, as they presume that they know all the elements in advance. [see [2]]

4.3 Stability

Definition

A sorting algorithm is said to be stable if it preserves the relative order of equal elements.

The concept of stability does not make much of a sense for simple/primitive data since the sorting key is the value of the element itself. If an array of integers

contains two or more repeated values, the order in which these repeated values appear in the final sorted array does not make a difference (at least from the naked eyes).

To illustrate the idea of stability in sorting, let us consider a list of records (or objects) that contain both a key (value) and an associated name. We will sort this list based on the key while maintaining the relative order of the names.

Example Data Let's take the following list of records, where each record consists of a tuple with an integer key and a name:

$$\text{data} = \{(3, "Alice"), (1, "Bob"), (2, "Charlie"), (3, "David"), (2, "Eve")\}$$

Unstable Sort Example Using an unstable sorting algorithm (like the default quicksort), sorting the data by the first element may yield a different relative order for equal keys:

1. **Original Order**:
    - (3, "Alice")
    - (1, "Bob")
    - (2, "Charlie")
    - (3, "David")
    - (2, "Eve")
2. **Sorted Using Unstable Algorithm (Result)**:
    - (1, "Bob")
    - (2, "Eve") **(changed order for equal key)**
    - (2, "Charlie")
    - (3, "David") **(changed order for equal key)**
    - (3, "Alice")

In this example, the output shows that the relative order of "Eve" and "Charlie" as well as "David" and "Alice" has changed, demonstrating that the algorithm is unstable.

Stable Sort Example Using a stable sorting algorithm (like Merge Sort), sorting the same data will preserve the order of the names with equal keys:

1. **Original Order**:
    - (3, "Alice")
    - (1, "Bob")
    - (2, "Charlie")
    - (3, "David")
    - (2, "Eve")

2. **Sorted Using Stable Algorithm (Result)**:

   - (1, "Bob")
   - (2, "Charlie") **(order preserved)**
   - (2, "Eve") **(order preserved)**
   - (3, "Alice") **(order preserved)**
   - (3, "David") **(order preserved)**

In this sorted list, the order of "Charlie" and "Eve" is maintained, as well as the order of "Alice" and "David".This concept is particularly important in applications such as sorting databases or maintaining user preferences where the order of equal elements is significant.

5 Summary

We summarize each of the sorting algorithm listed above with its time complexity into a compact table below:

Algorithm	**Best Case**	**Average Case**	**Worst Case**	**Space**	**Stable**	**In-Place**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	No
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Yes
Shell Sort	$O(n \log n)$	$O(n^{3/2})$	$O(n^2)$	$O(1)$	No	Yes
Stooge Sort	$O(n^{2.709})$	$O(n^{2.709})$	$O(n^{2.709})$	$O(1)$	Yes	No
Bogo Sort	$O(n)$	$O(\infty)$	$O(\infty)$	$O(n)$	Yes	No
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	Yes	No
Bucket Sort	$O(n)$	$O(n+k)$	$O(n^2)$	$O(n)$	Yes	No
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	Yes	No

6 Reference

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 4th edition. MIT Press, 2022. Chapter 5: Probabilistic Algorithms.

[2] A good place to start reading about online and offline algorithms

[3] Goodrich, M. T., & Tamassia, R. (2001). *Algorithm Design and Foundations*. Chapter 4: Sorting, Sets, and Select (p. 242).

[4] Donald E. Knuth. *The Art of Computer Programming Volume 3 - Sorting and Searching*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[5] The RAM model of computation.

[6] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986.