Pengjie Zhang
(352) 281-6813

Sorting Algorithms Analysis

Background

Sorting is given an unsorted list of things and sort them into a specific order based on a predefined equalityness. Say given numbers [9, 5, 1, 4, 3], sort the list in ascending order will get [1, 3, 4, 5, 9]; sort the list in descending order will get [9, 5, 4, 3, 1].

Nowadays there are many sorting algorithms designed and in play, with different time complexity and space complexity. Such as Bubble sort, its time complexity is O(n^2), space complexity is O(1).

Besides time complexity and space complexity, the easiness to understand/code is another important characteristic of a sorting algorithm.

Overview

In this lab, we are going to give a deep analysis of some sorting algorithms which are relatively easy to understand. We will focus on complexity analysis, and at the same time provide working code snippets in Java.

There are two metrics which are very important for measuring how effective a sorting algorithm is:

- number of comparisons
- number of swaps

We will explore the above two metrics for each sorting algorithm when sorting an array of numbers in ascending order (sort in descending order is the same complexity).

Analysis

Bubble sort

Bubble sort works iteratively, in each iteration, it moves the *current-largest* element backwards by doing compare-and-swap for each pair of elements, and finally settles the *current-largest* element to the correct position. The array will be sorted after all iterations are done. Ex sort [9, 5, 1, 4, 3] into ascending order:

```
Iteration 1:
        [(9, 5), 1, 4, 3] => [5, 9, 1, 4, 3] // a) Compare the first pair (9, 5), swap if needed.

                                             // b) Based on the previous result, now compare
        [5, (9, 1), 4, 3] => [5, 1, 9, 4, 3] //    second pair (9, 1), swap if needed.
                                             // c) Based on the previous result, now compare
        [5, 1, (9, 4), 3] => [5, 1, 4, 9, 3] //     third pair (9, 4), swap if needed.

                                             // d) Based on the previous result, now compare
        [5, 1, 4, (9, 3)] => [5, 1, 4, 3, 9] //     fourth pair (9, 3), swap if needed.

// After iteration 1, the largest element 9 is moved into the last position which is the correct
// position. Iteration 2 will continue doing the compare-and-swap but ignore the last element.
// We use 'X' to indicate not-care-anymore.

Iteration 2:
        [(5, 1), 4, 3, X] => [1, 5, 4, 3, X]
        [1, (5, 4), 3, X] => [1, 4, 5, 3, X]
        [1, 4, (5, 3), X] => [1, 4, 3, 5, X]
Iteration 3:
        [(1, 4), 3, X, X] => [1, 4, 3, X, X]
        [1, (4, 3), X, X] => [1, 3, 4, X, X]
Iteration 4:
        [(1, 3), X, X, X] => [1, 3, X, X, X]
Done: [1, 3, 4, 5, 9]
```

Java code

```java
public void bubbleSort(int[] a) {
    for (int iteration = 1; iteration < a.length; ++iteration) {
        for (int i = 0; i < a.length - iteration; ++i) {
            if (a[i] > a[i+1]) { // Swap
                int tmp = a[i];
                a[i] = a[i+1];
                a[i+1] = tmp;
            }
        }
    }
}
```

Complexity analysis

Apparently different original arrays will require different numbers of comparisons and swaps, there are three scenarios:

1. best case scenario: the array is already sorted in ascending order, say sort [1, 3, 4, 5, 9], which we don't swap at all.

2. worst case scenario: the array is in descending order, say sort [9, 5, 4, 3, 1], which we do many swaps.
3. average case scenario

Let's say there are n elements in the array to be sorted.

	Best case	Worst case	Average case
# of comparisons	(1+n)n/2	(1+n)n/2	(1+n)n/2
# of swaps	0	(1+n)n/2	< (1+n)n/2

Insertion sort

Insertion sort works iteratively, in each iteration, it inserts an *un-handled* element into the correct position of an already sorted sub-array, thus augmenting the size of the sorted sub-array. The array will be sorted after all iterations are done. Ex sort [9, 5, 1, 4, 3] into ascending order:

```
       [(9), 5, 1, 4, 3] // initially, the first element itself compose a sorted-subarray
                         // (which only has one element), which is the starting point.
Iteration 1:
       [(9), 5, 1, 4, 3] => [5, 9, 1, 4, 3] // insert 5 into the correct position of the sorted
                                            // subarray [9] we will get [5, 9]
Iteration 2:
       [(5, 9), 1, 4, 3] => [1, 5, 9, 4, 3] // insert 1 into the correct position of the sorted
                                            // subarray [5, 9] we will get [1, 5, 9]
Iteration 3:
       [(1, 5, 9), 4, 3] => [1, 4, 5, 9, 3] // insert 4 into the correct position of the sorted
                                            // subarray [1, 5, 9], we will get [1, 4, 5, 9]
Iteration 4:
       [(1, 4, 5, 9), 3] => [1, 3, 4, 5, 9] // insert 3 into the correct position of the sorted
                                            // subarray [1, 4, 5, 9], we will get
                                            // [1, 3, 4, 5, 9]
```

Java code

```java
public void insertionSort(int[] a) {
    // handleElement start from 1 because index 0 is already sorted
    // (single element is treated as sorted)
    for (int handleElement = 1; handleElement < a.length; ++handleElement) {
        int value = a[handleElement];
        int ind = handleElement - 1; // We compare forward.
        while (ind >= 0 && a[ind] > value) {
            a[ind + 1] = a[ind]; // Move elements backward if it is bigger than `value`.
            --ind;
        }
        a[ind + 1] = value; // Put the value into the correct position.
    }
}
```

Complexity analysis

Again three scenarios:

1. best case scenario: the array is already sorted in ascending order, say sort [1, 3, 4, 5, 9], which we don't swap at all.
2. worst case scenario: the array is in descending order, say sort [9, 5, 4, 3, 1], which we do many swaps.
3. average case scenario

Let's say there are n elements in the array to be sorted.

	Best case	Worst case	Average case
# of comparisons	n	(1+n)n/2	< (1+n)n/2
# of swaps	0	(1+n)n/2	< (1+n)n/2

Selection sort

Selection sort works iteratively, in each iteration, it finds the *current-largest* element by doing comparison, then settles it into the correct position (swapping). The array will be sorted after all iterations are done. Ex sort [9, 5, 1, 4, 3] into ascending order:

```
Iteration 1:
        [9, 5, 1, 4, 3] => [3, 5, 1, 4, 9] // Find current-largest element which is 9, its correct
                                            // position should be the last position, so swap
                                            // 9 with 3

// We use ‘X’ to represent elements that we don’t need to care about anymore.

Iteration 2:
        [3, 5, 1, 4, X] => [3, 4, 1, 5, X] // Find the current-largest element which is 5, its
                                            // correct position is before the first ‘X’, so swap
                                            // 5 with 4
Iteration 3:
        [3, 4, 1, X, X] => [3, 1, 4, X, X] // Find the current-largest element which is 4, its
                                            // correct position is before the first ‘X’, so swap
                                            // 4 with 1
Iteration 4:
        [3, 1, X, X, X] => [1, 3, X, X, X] // Find the current-largest element which is 3, its
                                            // correct position is before the first ‘X’, so swap
                                            // 3 with 1
Done: [1, 3, 4, 5, 9]
```

Java code

```java
public void selectionSort(int[] a) {
    for (int putToPosition = a.length - 1; putToPosition >= 1; --putToPosition) {
        int currentLargestIndex = 0;
        for (int i = 0; i <= putToPosition; ++i) {
            if (a[i] > a[currentLargestIndex]) {
                currentLargestIndex = i;
            }
        }

        int tmp = a[putToPosition];
        a[putToPosition] = a[currentLargestIndex];
        a[currentLargestIndex] = tmp;
    }
}
```

Complexity analysis

Again three scenarios:

1. best case scenario: the array is already sorted in ascending order, say sort [1, 3, 4, 5, 9], which we don't swap at all.
2. worst case scenario: the array is in descending order, say sort [9, 5, 4, 3, 1], which we do many swaps.
3. average case scenario

Let's say there are n elements in the array to be sorted.

	Best case	Worst case	Average case
# of comparisons	(1+n)n/2	(1+n)n/2	(1+n)n/2
# of swaps	0	n	< n

Shell sort

Shell sort also works iteratively.

In each iteration, the **whole** array is divided into multiple *equal* length subarray(s) (the last subarray might have smaller length due to not not being divisible). Say an array [9, 5, 1, 6, 8, 4, 3, 2, 7] and we want each subarray to have length of 4, the result of dividing would be [[9, 5, 1, 6], [8, 4, 3, 2], [7]]. Then *virtually(cognitively)* pick i***th*** element from each subarray to form a new array A, and sort A using **insertion sort**, this is another inner-iteration, i will loop from 0 to subArrayLength-1.

There are some heuristics in picking the length of the subarray for each iteration. Suppose the length of the whole array is n, the classic approach is first iteration the length of the subarray is n/2; second iteration the length of the subarray is n/4; third iteration the length of the subarray is n/8, ..., the last iteration the length of the subarray is 1.

Ex sort [9, 5, 1, 6, 8, 4, 3, 2, 7] into ascending order:

```
Iteration 1: // 9/2 = 4, i.e. the length of each subarray is 4
        [9, 5, 1, 6, 8, 4, 3, 2, 7] => [ [9, 5, 1, 6], [8, 4, 3, 2], [7] ]

                Inner-iteration-0: [ [9, 5, 1, 6], [8, 4, 3, 2], [7] ]
                                      => // now suppose you are going to insertion sort [9, 8, 7]
                                    [ [7, 5, 1, 6], [8, 4, 3, 2], [9] ]

                Inner-iteration-1: [ [7, 5, 1, 6], [8, 4, 3, 2], [9] ]
                                      => // now suppose you are going to insertion sort [5, 4]
                                    [ [7, 4, 1, 6], [8, 5, 3, 2], [9] ]

                Inner-iteration-2: [ [7, 4, 1, 6], [8, 5, 3, 2], [9] ]
                                      => // now suppose you are going to insertion sort [1, 3]
                                    [ [7, 4, 1, 6], [8, 5, 3, 2], [9] ]

                Inner-iteration-3: [ [7, 4, 1, 6], [8, 5, 3, 2], [9] ]
                                      => // now suppose you are going to insertion sort [6, 2]
                                    [ [7, 4, 1, 2], [8, 5, 3, 6], [9] ]

Iteration 2: // 9/4 = 2, i.e. the length of each subarray is 2
        [7, 4, 1, 2, 8, 5, 3, 6, 9] => [ [7, 4],  [1, 2],  [8, 5],  [3, 6],  [9] ]

                Inner-iteration-0: [ [7, 4],  [1, 2],  [8, 5],  [3, 6],  [9] ]
                                      => // insertion sort [7, 1, 8, 3, 9]
                                    [ [1, 4],  [3, 2],  [7, 5],  [8, 6],  [9] ]

                Inner-iteration-1: [ [1, 4],  [3, 2],  [7, 5],  [8, 6],  [9] ]
                                      => // insertion sort [4, 2, 5, 6]
                                    [ [1, 2],  [3, 4],  [7, 5],  [8, 6],  [9] ]

Iteration 3: // 9/8 = 1, i.e. the length of each subarray is 1
        [1, 2, 3, 4, 7, 5, 8, 6, 9] => [ [1], [2], [3], [4], [7], [5], [8], [6], [9] ]

                Inner-iteration-0: [ [1], [2], [3], [4], [7], [5], [8], [6], [9] ]
                                      => // insertion sort [1, 2, 3, 4, 7, 5, 8, 6, 9]
                                    [ [1], [2], [3], [4], [5], [6], [7], [8], [9] ]
```

As we can see, as the length of the subarray decreases exponentially iteration by iteration, the last iteration is effectively doing an insertion sort for the whole array. Another example to depict the mathematical calculation, suppose an array of length 134,

1. The first iteration the length of subarray is 134/2=67;
2. The second iteration the length of the subarray is 134/4=33;
3. The third iteration the length of the subarray is 134/8=16;
4. The fourth iteration the length of the subarray is 134/16=8;
5. The fifth iteration the length of the subarray is 134/32=4;

6. The sixth iteration the length of the subarray is 134/64=2;
7. The seventh iteration the length of the subarray is 134/128=1 (**effectively an insertion sort on the whole array**)

As we can see, the number of iterations should roughly equal to $log_2 134$, if the whole array is of length n, then the # of iterations should roughly equal to $log_2 n$.

Java code

```java
public void shellSort(int[] a) {

    int n = a.length;
    for (int len = n / 2; len > 0; len /= 2) { // 'len' is the length of the subarray

        // we are going to do the insertion sort parallelly, see explanation below.
        for (int i = len; i < n; i += 1) {
            int temp = a[i];
            int j;
            for (j = i; j - len >= 0 && a[j - len] > temp; j -= len) {
                a[j] = a[j - len];
            }
            a[j] = temp;
        }
    }
}
```

Above code might confuse you in the parallel insertion sort part because it doesn't follow the previous step-by-step-guide 100%, step-by-step-guide below matches the java code.

```
Iteration 1: // 9/2 = 4, i.e. the length of each subarray is 4
       [9, 5, 1, 6, 8, 4, 3, 2, 7] => [ [9, 5, 1, 6], [8, 4, 3, 2], [7] ]

              Inner-iteration-0: [ [9, 5, 1, 6], [8, 4, 3, 2], [7] ]
                                   => // now suppose you are going to insertion sort [9, 8]
                                  [ [8, 5, 1, 6], [9, 4, 3, 2], [7] ]

              Inner-iteration-1: [ [8, 5, 1, 6], [9, 4, 3, 2], [7] ]
                                   => // now suppose you are going to insertion sort [5, 4]
                                  [ [8, 4, 1, 6], [9, 5, 3, 2], [7] ]

              Inner-iteration-2: [ [8, 4, 1, 6], [9, 5, 3, 2], [7] ]
                                   => // now suppose you are going to insertion sort [1, 3]
                                  [ [8, 4, 1, 6], [9, 5, 3, 2], [7] ]

              Inner-iteration-3: [ [8, 4, 1, 6], [9, 5, 3, 2], [7] ]
                                   => // now suppose you are going to insertion sort [6, 2]
                                  [ [8, 4, 1, 2], [9, 5, 3, 6], [7] ]

              Inner-iteration-4: [ [8, 4, 1, 2], [9, 5, 3, 6], [7] ]
                                   => // now suppose you are going to insertion sort [8, 9, 7]
                                  [ [7, 4, 1, 2], [8, 5, 3, 6], [9] ]
```

Actually, the result of Iteration 1 in this step-by-step-guide is the same as the previous step-by-step-guide, and if you look more carefully, we are just coding insertion sort parallelly.

Complexity analysis

Let us suppose the array length is exponential of 2 to simplify the analysis of the complexity.

1. When (len == n/2): the whole array is effectively divided into 2 parts: [part0, part1], each part has n/2 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap

    So total to n/2 (comparison or swap)

2. When (len == n/4): the total array is effectively divided into 4 parts: [part0, part1, part2, part3], each part has n/4 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap;
    b. each element in part2 will do at most 2 comparisons, at most 2 swaps;
    c. each element in part3 will do at most 3 comparisons, at most 3 swaps

    So total to n/4 + 2 * n/4 + 3 * n/4 = (1+3)3/2 * n/4 = 6 * n/4 (comparison or swap)

3. When (len == n/8): the total array is effectively divided into 8 parts: [part0, part1, part2, part3, part4, part5, part6, part7], each part has n/8 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap;
    b. each element in part2 will do at most 2 comparisons, at most 2 swaps;
    c. each element in part3 will do at most 3 comparisons, at most 3 swaps;
    d. each element in part4 will do at most 4 comparisons, at most 4 swaps;
    e. each element in part5 will do at most 5 comparisons, at most 5 swaps;
    f. each element in part6 will do at most 6 comparisons, at most 6 swaps;
    g. each element in part7 will do at most 7 comparisons, at most 7 swaps

    So total to n/8 + 2 * n/8 + 3 * n/8 + 4 * n/8 + 5 * n/8 + 6 * n/8 + 7 * n/8 = 28 * n/8

4. ...
5. When (len == 1): the total array is effectively divided into n parts:

Actually for each of above iteration

1. # of comparison = $(\frac{1}{2}x^2 - \frac{x}{2}) \times \frac{n}{x}$ where x is equal to either 2, 4, 8, 16, ...
2. # of swaps = $(\frac{1}{2}x^2 - \frac{x}{2}) \times \frac{n}{x}$ where x is equal to either 2, 4, 8, 16, ...

The total # of comparisons/swaps is $n^2 - \frac{1}{2}nlogn$

However, above analysis doesn't **take previous iteration-effect into account**, now let's take a deeper analysis:

1. When (len == n/2): the whole array is effectively divided into 2 parts: [part0, part1], each part has n/2 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap.

    So total to n/2 (comparison or swap)

2. When (len == n/4): the total array is effectively divided into 4 parts: [part0, part1, part2, part3], each part has n/4 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap;
    b. each element in part2 will do at most 2 comparisons, at most 1 swap;
    c. each element in part3 will do at most 3 comparisons, at most 2 swaps.

    So total to n/4 + 2 * n/4 + 3 * n/4 = (1+3)3/2 * n/4 = 6 * n/4 comparisons

    total to n/4 + n/4 + 2 * n/4 = 4 * n/4 swaps

3. When (len == n/8): the total array is effectively divided into 8 parts: [part0, part1, part2, part3, part4, part5, part6, part7], each part has n/8 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap;
    b. each element in part2 will do at most 2 comparisons, at most 1 swap;
    c. each element in part3 will do at most 3 comparisons, at most 2 swaps;
    d. each element in part4 will do at most 3 comparisons, at most 2 swaps;
    e. each element in part5 will do at most 4 comparisons, at most 3 swaps;
    f. each element in part6 will do at most 4 comparisons, at most 3 swaps;
    g. each element in part7 will do at most 5 comparisons, at most 4 swaps;

    So total to (1+2+3+3+4+4+5) * n/8 = 22 * n/8 comparisons

    total to (1+1+2+2+3+3+4) * n/8 = 16 * n/8 swaps

4. When (len == n/16): the total array is effectively divided into 16 parts: [part0, part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11, part12, part13, part14, part15], each part has n/16 elements.
    a. each element in part1 will do at most 1 comparison, at most 1 swap;
    b. each element in part2 will do at most 2 comparisons, at most 1 swap;
    c. each element in part3 will do at most 3 comparisons, at most 2 swaps;
    d. each element in part4 will do at most 3 comparisons, at most 2 swaps;

e. each element in part5 will do at most 4 comparisons, at most 3 swaps;
f. each element in part6 will do at most 4 comparisons, at most 3 swaps;
g. each element in part7 will do at most 5 comparisons, at most 4 swaps;
h. each element in part8 will do at most 5 comparisons, at most 4 swaps;
i. each element in part9 will do at most 6 comparisons, at most 5 swaps;
j. each element in part10 will do at most 6 comparisons, at most 5 swaps;
k. each element in part11 will do at most 7 comparisons, at most 6 swaps;
l. each element in part12 will do at most 7 comparisons, at most 6 swaps;
m. each element in part13 will do at most 8 comparisons, at most 7 swaps;
n. each element in part14 will do at most 8 comparisons, at most 7 swaps;
o. each element in part15 will do at most 9 comparisons, at most 8 swaps;

So total [(1+2+3+4+5+6+7+8+9) + (1+2+3+4+5+6+7+8) - 1 - 2] * n/16 = 78 * n/16 comparisons

Total to [(1+2+3+4+5+6+7+8) + (1+2+3+4+5+6+7)] * n/16 = 64 * n/16 swaps

5. ...
6. When (len == 1): the total array is effectively divided into n parts:

Actually for each of above iteration

3. # of comparison = $(\frac{1}{4}x^2 + x - 2) \times \frac{n}{x}$ where x is equal to either 2, 4, 8, 16, ...
4. # of swaps = $\frac{1}{4}x^2 \times \frac{n}{x}$ where x is equal to either 2, 4, 8, 16, ...

So in worst case

1. Total # of comparisons = $\frac{1}{2}n^2 + nlogn - 2n$
2. Total # of swaps = $\frac{1}{2}n^2$

Again three scenarios:

1. best case scenario: the array is already sorted in ascending order, say sort [1, 3, 4, 5, 9], which we don't swap at all.
2. worst case scenario: the array is in descending order, say sort [9, 5, 4, 3, 1], which we do many swaps.
3. average case scenario

Let's say there are n elements in the array to be sorted.

	Best case	Worst case	Average case
# of comparisons	(1+n)n/2	1/2n^2+nlogn-2n	< 1/2n^2+nlogn-2n

# of swaps	nlogn - n	1/2n^2	< 1/2n^2

Emmm, still $O(n^2)$ time complexity overall.

Radix sort

Radix sort is only usable for a limited number of scenarios, such as sorting numbers. It works iteratively. Each iteration it will try to move elements to its correct *relative* position based on current compared digits. Ex sort [345, 671, 735, 143, 847, 452, 559] in ascending order, we start from least significant digits

```
Iteration 1: // Create array of double-linked-list,
        For loop each element in the array, get the least significant digit of the element, push
        the element into the right double-linked-list (digit value matched the index of the
        doubly-linked-list)
                [0]
                [1] -> 671
                [2] -> 452
                [3] -> 143
                [4]
                [5] -> 345 -> 735
                [6]
                [7] -> 847
                [8]
                [9] -> 559
        Read elements in the array of doubly linked list (use pop) and write them back to the
        original array start from index 0, we will get [671, 452, 143, 345, 735, 847, 559]

Iteration 2: // Reuse array of double-linked-list,
        For loop each element in the array again, get the second to the least significant digit of
        the element, push the element into the right double-linked-list (digit value matched the
        index of the doubly-linked-list)
                [0]
                [1]
                [2]
                [3] -> 735
                [4] -> 143 -> 345 -> 847
                [5] -> 452 -> 559
                [6]
                [7] -> 671
                [8]
                [9]
        Read elements in the array of doubly linked list (use pop) and write them back to the
        original array start from index 0, we will get [735, 143, 345, 847, 452, 559, 671]

Iteration 3: // Reuse array of double-linked-list,
        For loop each element in the array again, get the third to the least significant digit of
        the element, push the element into the right double-linked-list (digit value matched the
        index of the doubly-linked-list)
                [0]
                [1] -> 143
                [2]
```

```
            [3] -> 345
            [4] -> 452
            [5] -> 559
            [6] -> 671
            [7] -> 735
            [8] -> 847
            [9]
     Read elements in the array of doubly linked list (use pop) and write them back to the
     original array start from index 0, we will get [143, 345, 452, 559, 671, 735, 847]
```

Java code

```java
public void radixSort(int[] arr) {

    LinkedList<Integer>[] tool = new LinkedList<Integer>[10];
    for (int i = 0; i < tool.length; ++i) {
        tool[i] = new LinkedList<Integer>();
    }

    int mod = 1;
    for (int round = 1; round <= 3; ++round) {

        // Add numbers to the doubly linked lists.
        for (int i = 0; i < arr.length; ++i) {
            int a = arr[i];
            int interestedDigit = (a % (mod * 10)) / mod;
            tool[interestedDigit].add(a);
        }

        // Read numbers back from doubly linked lists to the array.
        int position = 0;
        for (int i = 0; i < tool.length; ++i) {
            while (!tool[i].IsEmpty()) {
                arr[position++] = tool[i].poll();
            }
        }

        mod *= 10;
    }
}
```

Complexity analysis

Radix sort only usable in certain scenario, and its time complexity is *numOfDigitsInLargestElement * n*

Appendix

Wiki about sorting algorithm